CS240 Part III: Complexity theory

Introduction

In part I of the course we saw that some problems are algorithmically unsolvable. Eg:

- the halting problem (will a given TM halt on a given input?)
- deciding the truth of an arbitrary statement about arithmetic.

But there are wide variations in difficulty amongst the solvable problems. In practice it's no use to know that a problem is solvable, if all solutions take an inordinately long time to run. So we need to refine our view of the solvable problems. In part III we will classify them according to difficulty: how long they take to solve. Note: the problems in part III are solvable by an algorithm; but they may not be solvable in a reasonable time.

Earlier, we formalised the notion of a *solvable problem* as one that can be solved by a Turing machine (Church's thesis). We did this to be able to reason about algorithms in general. We will now formalise the *complexity* of a problem, in terms of Turing machines, so that we can reason in general about the varying difficulty of problems.

We will classify problems into four levels of difficulty or *complexity*. (There are many finer divisions).

- The class *P* of *tractable* problems that can be solved efficiently (in polynomial time: *p-time*).
- The *intractable* problems. Even though these are algorithmically solvable, any algorithmic solution will run in exponential time (or slower) in the worst case. Such problems cannot be solved in a reasonable time, even for quite small inputs, and for practical purposes they are unsolvable for most inputs, unless the algorithm's average case performance is good. The exponential function dwarfs technological changes (Fig. 6.1), so hardware improvements will not help much.
- The class *NP* of problems. These form a kind of half-way house between the tractable and intractable problems. They can be solved in p-time, but by a *non-deterministic* algorithm. Could they have p-time deterministic solutions? This is the famous question 'P=NP?' — is every NP-problem a P-problem? The answer is thought to be *no,* though no-one has proved it. So these problems are currently intractable, but haven't been proved so.
- The class *NPC* of *NP-complete problems.* In a precise sense, these are the hardest problems in NP. Cook's theorem (§12) shows that NP-complete problems exist (e.g., 'PSAT'); examples include the Hamiltonian circuit and travelling salesman problems we saw in §7–8, and around 1,000 others. All NP-complete problems reduce to each other in polynomial time (see §8.6). So a fast solution to any NP-complete problem would immediately give fast solutions to all the others — in fact to all NP problems. Because of this, most people believe NP-complete problems have no fast deterministic solution.

Why study complexity? It is useful in practice. It guides us towards the tractable problems that are solvable with fast algorithms. Conversely, NP-complete problems occur frequently in applications. Complexity theory tells us that when we meet one, it might be wise not to seek a fast solution, as many have tried to do this without success.

On a more philosophical level, Church's thesis defined an algorithm to be a Turing machine. So two Turing machines that differ even slightly represent two different algorithms. But if each reduces quickly to the other, as all NP-complete problems do, we might wish to regard them as the *same* algorithm — even if they solve quite different problems! So the notion of fast reducibility of one problem or algorithm to another gives us a higher-level view of the notion of algorithm.

So in part III we will:

- define the *run time function* of a Turing machine
- introduce *non-deterministic Turing machines* and define their run time function also
- formalise *fast reduction* of one problem to another
- examine *NP- and NP-complete problems.*

9 Basic complexity theory

We begin by introducing the notions needed to distinguish between tractable and intractable problems. The classes NP and NPC will be discussed in §10 and §12.

9.1 YES/NO PROBLEMS

We will only deal with 'yes/no problems', so that we can ignore the output of a Turing machine, only considering whether it halts and succeeds or halts & fails. This simplification will be especially helpful when we consider non-deterministic Turing machines (§10).

9.1.1 Definition

A *yes/no problem* is one with answer yes or no. Each yes/no problem has a set of *instances* — the set of valid inputs for that problem. The *yes-instances* of a problem are those instances for which the answer is 'yes'. The others are the no-instances.

9.1.2 Examples

Many problems can be put in yes/no form:

Problem	**instances**	**yes-instances**	**no-instances**
Is w a prime number?	binary representations of numbers	binary representations of primes	binary representations of non-primes
Halting problem: Does M halt on input w?	all pairs (*code*(M),w) where M is a standard TM, and w a word of C	all those pairs (*code*(M),w) such that M halts & succeeds on w	the pairs (*code*(M),w) such that M doesn't halt & succeed on w
HCP	all (finite) graphs	graphs with a Hamiltonian circuit	graphs with no Hamiltonian circuit
TSP	all pairs (G,d), where G is a weighted graph, and $d \geq 0$	all pairs (G,d) such that G has a Hamiltonian circuit of length $\leq d$	all pairs (G,d) such that G has no Hamiltonian circuit of length $\leq d$

Even if we ignore its output, a Turing machine can still 'communicate' with us by halting & succeeding ('yes'), or halting & failing ('no'), so it can answer yes/no problems. Of course, we may have to code the instances of the problem as words, so they can be input to a Turing machine. For HP, *code*(M) is given to the TM, as M itself is a machine, not a word. The coding of instances into words should be *reasonable:* we do not allow unary representation of numbers, and cheating (such as coding all yes-instances as 'yes' and all no-instances as 'no') is not allowed.

9.1.3 Acceptance, rejection, solving

1. A Turing machine M is said to *accept* a word w of its input alphabet if M halts and succeeds on input w.
2. M is said to *reject* w if M halts and fails on input w.

3. A Turing machine M is said to *solve* a yes/no problem A if:
   - every instance of A is a word of the input alphabet of M (or can be coded as one in a reasonable way);
   - M accepts all the yes-instances of A;
   - M rejects all the no-instances of A.

9.1.3.1 Example

1. In §3.3.4 we saw a Turing machine that halts and succeeds if its input word is a palindrome, and halts & fails if not. This machine solves the yes/no problem *is* $w \in I^*$ *a palindrome?*, where I is its input alphabet.
2. The universal Turing machine U does <u>not</u> solve the halting problem. If we give it *code*(M)*w for some standard M and word $w \in C^*$, then U does halt & succeed on the yes-instances (see the table above). <u>But</u> it does <u>not</u> halt & fail on all the no-instances: if M runs forever on w, U never halts on *code*(M)*w.

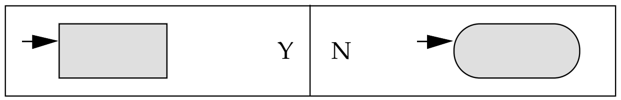
9.1.3.2. Our problems must have infinitely many y- and n-instances

<u>We do not consider yes/no problems with only finitely many yes-instances, or only finitely many no-instances</u>. They are too easy. E.g., if the yes-instances of a yes/no problem X are just $y_1,\ldots,y_n$, a Turing machine M can 'solve' X by checking to see if the input word w is one of the y_i. (The y_i are 'hard-wired' into M; we can do this as there are only finitely many of them.) If it is, M halts and succeeds; otherwise it halts and fails. No 'calculation' is involved. E.g., 'Is 31 prime?' has no instances at all, and is solved by the trivial Turing machine whose initial state is halting.

This may seem odd. For example, one of the Turing machines in Fig. 9.1 (both with input alphabet C, say) solves the yes/no problem:

'Is (a) Goldbach's conjecture true, and (b) w = w?'

The instances of this problem are all words w of C. The yes-instances are those w such that (a) Goldbach's conjecture is true, and (b) w=w. The no-instances are the rest. So if the conjecture is true, every $w \in C^*$ is a yes-instance, so Y solves it (Y halts & succeeds on any input). If not, every $w \in C^*$ is a no-instance, so N solves it (N halts & fails on any input). Of course, we don't know which! <u>But the problem is solvable</u> — either by Y or by N.



9.1 Which machine solves Goldbach's conjecture?

For a problem to be solvable according to our definition, we are only concerned that a Turing machine solution exists, not in how to find it. So

Goldbach's conjecture really is too easy for us! We are only interested in problems with infinitely many yes- and infinitely many no-instances.

9.2 POLYNOMIAL TIME TURING MACHINES

As for algorithms in §6, we want to define how long a Turing machine takes to run. We consider the worst case only. The warnings in §6 about doing this still apply: e.g., M may usually be very quick, taking the maximum time on only a few inputs.

9.2.1 Run time function of Turing machines

Let $M = (Q,\Sigma,I,q_0,\delta,F)$ be a Turing machine. For any n = 0,1,2,…, M can run for different numbers of steps on different words w of length n. We write $time_M(n)$ for the the length of the longest run of M on any input of size n (we want the worst case).

We can call the function $time_M : \{0,1,2, \ldots\} \to \{0,1,2, \ldots, \infty\}$ the *run time function of M.*

9.2.2 p-time Turing machines

A Turing machine M is said to *run in polynomial time (p-time)* if there's a polynomial $p(n) = a_0+a_1n+a_2n^2+ \ldots+a_kn^k$, where the coefficients $a_0,\ldots,a_k$ are non-negative whole numbers, such that:

$$time_M(n) \le p(n) \text{ for all } n = 0,1,2,\ldots$$

That is, no run of M on any word of length n lasts longer than p(n) steps.

9.2.2.1. Remark

Turing machines that run in p-time are considered to be *fast.* This is a broad but still useful categorisation, sometimes called the Cook-Karp thesis (cf. Church's thesis).

9.2.2.2. Note

Note that we do not use the O-notation in the definition of M running in p-time (e.g., by saying '$time_M(n)$ is $O(n^k)$' for some k). We require that $time_M(n)$ should be at most p(n), not just at most c.p(n) for some constant c. This is no restriction because c.p(n) is a polynomial anyway. But further, we require that $time_M(n) \le p(n)$ for all n, however small, so that we are sure what happens for all n. For example:

9.2.2.3. p-time Turing machines always halt

If M is p-time then for some polynomial p(n), M takes at most p(n) steps to run on any word w of length n. But p(n) is always finite, for any n. So M always halts (succeeding or failing) on any input; it can't run forever.

The following exercise shows that insisting on a firm polynomial bound on run time for all n is not really a restriction.

9.2.2.4. Exercise

Let $f : I^* \to \Sigma^*$ be any partial function. Show that the following are equivalent:

(i) $f = f_M$ for some Turing machine M running in p-time

(ii) $f = f_{M^*}$ for some Turing machine M* such that $time_{M^*}(n)$ is $O(n^k)$ for some k.

Hint: for (ii)⇒(i), tabulate all 'short' inputs for M* as a look-up table.

9.3 TRACTABILITY OF PROBLEMS

Our main interest is in when a problem is *tractable:* when it has a reasonably fast solution. Intractable problems often have exponential run time function (or worse), and are effectively no better than unsolvable ones for even moderately large inputs. Here we define tractability formally, and look at some examples. We also discuss the main obstacle to finding a fast solution: the need to conduct an exhaustive search for it.

9.3.1 Tractable problems

A yes/no problem is said to be *tractable* if it can be solved by a Turing machine running in p-time. An algorithm is said to be tractable if it can be implemented by a Turing machine that runs in polynomial time.

9.3.1.1 Discussion

Tractable problems can be solved in a reasonable time for instances of reasonable size. The graph in Fig. 6.1 showed that polynomials ($100n^2$ etc.) grow at a manageable rate. The *degree* of the polynomial (k in §9.2.2) can usually be massaged down to 5 or better. As $\log_2(n) \le n$ for all $n \ge 1$ (exercise: prove it!), even run time functions such as $n.\log_2(n)$ are bounded by a polynomial (here, n^2).

The problems we saw in §7–8 are all solvable in polynomial time, except (probably) the Hamiltonian circuit and travelling salesman problems, HCP and TSP. Al-Khwārazmi's algorithms for arithmetic are tractable, as are most sorting and other algorithms in common use.

So, rather as in Church's thesis, we equate the *tractable* problems (a vague notion) with the problems *solvable in p-time* (a precise one). Doing so is sometimes called the *Cook-Karp thesis,* after S. Cook (of whom more later), and R. Karp. As we said, this is crude but still useful, and seems to reflect a 'genuine fact about nature'. But this time, not everyone agrees with it.

9.3.2 The class P

We write *P* for the class of tractable yes/no problems: those that are solvable by a Turing machine running in polynomial time.

9.3.3 P is closed under complementation

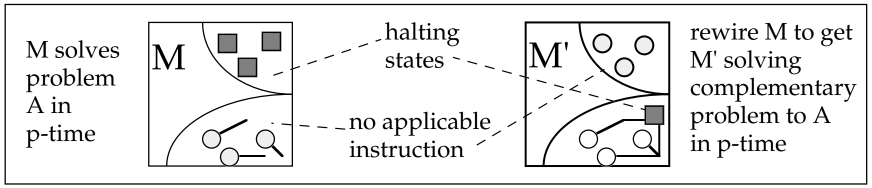
The class P has some nice properties. We look at one now, to practice using P. We will see more nice properties of P in §11.

9.3.3.1 Complements of yes-no problems

The *complement* of a yes/no problem is got by exchanging the answers *yes* and *no*. What were the yes-instances now become the no-instances, and vice versa. Eg. the complement of "is n prime?" is "is n composite?". If S is a class of problems (eg. P), we write *co-S* for the class consisting of the complements of the problems in S. Clearly, S = co-co-S.

9.3.3.2 P is closed under complementation

That is, if A is in P, the problem complementary to A is also in P. For if A is a yes/no problem solvable in p-time by a Turing machine M, we can rewire M so that (a) the halting states are no longer halting states, so that entering one now causes a halt and fail, and (b) whenever M is in a 'halt and fail' situation (no applicable instruction), control passes to a new state, which is halting. The rewired machine M' also runs in p-time. As M' accepts exactly the words that M rejects, it solves the complementary problem to A, which is therefore in P.



10.3 Sketch of why P is closed under complementation

9.3.3.3 Exercise

Show that co-P = P.

9.3.4 Intractable problems

A yes/no problem is said to be *intractable* if (i) it is algorithmically solvable, but (ii) it is not tractable. All solutions necessarily use intractable

algorithms. An algorithm is said to be *intractable* if it can't be implemented by a p-time Turing machine.

9.3.5 Seemingly intractable problems

Many common problems have not been proved to be either tractable or intractable! Typical examples are HCP and TSP. All known algorithms to solve these problems are intractable, but it is not known if the problems are themselves intractable. Maybe there's a fast algorithm that everyone's missed. For reasons to be seen in §12, this is not thought likely.

Besides TSP and HCP, problems in this category include:

9.3.5.1 Primality and non-primality testing

Primality problem: 'given a whole number x, is it prime?' Non-primality problem: 'given a whole number x, is it composite (i.e., not prime)?'

One solution is to check all possible factors y of x (so $1<y\leq\sqrt{x}$). If x is given in binary, then an input of length n could represent an x of up to about 2^n, which has $\sqrt{(2^n)} = (\sqrt{2})^n$ possible factors. This approach will therefore take exponential time.

There is no known p-time algorithm for primality or for non-primality testing, though people are hopeful that one will be found. A great deal of work has been done in this area, and some superb probabilistic algorithms have been devised, but as it is a crucial field for cryptography (see Harel's book), some of the work is probably not published.

9.3.5.2 Propositional satisfaction (PSAT)

Here we consider formulas of *propositional logic.* They are written using an alphabet I with *atoms,* p1, p2, p3,…,[1] connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies) and $\leftrightarrow$ (iff), and brackets), (. This is as for arithmetic, but there are no quantifiers this time. Any formula is a word of I, and can be given as input to a Turing machine.

We can assign varying truth values (true or false) to the atoms. We write h for a particular assignment: so e.g., h(p1) = true, h(p2) = false is possible. Then we can recursively work out the truth value of any formula A:

h($\neg$A) = true iff h(A) = false
h(A$\wedge$B) = true iff h(A) = h(B) = true
h(A$\vee$B) = true iff at least one of h(A), h(B) are true
h(A$\rightarrow$B) = true if either A is false or B is true or both
h(A$\leftrightarrow$B) = true iff h(A) = h(B).

So for example, if h makes p = p1 true and q = p2 false, then:

h(p$\rightarrow$q) = false

1 As in §5, p1 is 2 symbols, p and 1. So I has p and the numbers 0-9 say, so is finite.

$h(((p\to q)\to p)\to p)$ = true
$h(p\wedge q \vee (\neg p\wedge\neg q)) = h(p\leftrightarrow q)$ = false

Problem *(PSAT, propositional satisfiability):* Given a formula A, is there some assignment h to the atoms of A such that h(A) = true? That is: is A *satisfiable?*

A Turing machine could check whether A is satisfiable by checking every valuation for the atoms in A. We want to find out how long this might take. For each atom p, there are 2 possible values of h(p), so for n atoms there are 2^n possible valuations. So we need to estimate how many atom A has, in the worst case.

To simplify, we assume that every atom has the same length (say 1) as a word. Whilst this will be false in general (e.g., the 1,000,000th atom, $p_{1000000}$, will have length 8), it will be true e.g., for up to 9999 atoms if we use base 10,000 (say) for arithmetic; and solving PSAT for an arbitrary formula with 10,000 different atoms is currently unthinkable.

Under this assumption, for any r>0 we can easily find a formula with r atoms that has length at most 6r. We can prove this by induction on r. If r = 1 then certainly both p and ¬p have length ≤6. If it's true for r, let A have r atoms and be of length ≤6r. Choose an atom p not occurring in A. Then (p∧A), (p→A), (¬A∧¬p) etc. have r+1 atoms. Their length is the length of A, plus 1 for p (thanks to our simplifying assumption), plus 2 for (,) 1 for the connective (∧, →, etc.), and at most 2 for possible ¬'s. The total is ≤ (length of A) + 6 ≤ 6r + 6 = 6(r+1). QED.[2]

Hence for any n (divisible by 6) there are formulas of length n with at least n/6 atoms. So in the worst case, there are at least $2^{n/6}$ different valuations h for A of length n. So the run time on an input of length n is at least $2^{n/6} = (\sqrt[6]{2})^n$ in the worst case — exponential. There are more sophisticated methods (tableaux etc.), but no known tractable one. PSAT is NP-complete — see below, §12.

9.3.6 Exhaustive search in algorithms

Broadly, the yes/no problems we've seen fall into two types:

(∃) *Is there a needle in the haystack?* We want to show that there is a solution amongst the many possibilities. One way to do so (we cannot rule out the possibility that there are other ways) is to actually find a solution — e.g., a factor of a composite number, a (short) Hamiltonian circuit (HCP and TSP), or a valuation making a formula true (PSAT).

(∀) *Is there no needle in the haystack?* We want to show that none of the possible solutions is in fact a solution. *Complements* of type (∃) problems are of this type. E.g., to show n is prime, we want to establish that there's no number between 2 and √n that divides n.

Exhaustive search (try all possible solutions) can be used in (∃) and (∀), but it leads to intractability, as the number of possible solutions tends to rise at least exponentially with the input size. It could be called exhausting search. We very much want to avoid it.

But at least, for any possible solution that the search throws up, we generally have a fast (p-time) way of checking that it actually is a solution. This is certainly so in all the cases we've seen. For example, it's fast to check whether a given possible factor of a number really is a factor. It's fast to check whether a possible Hamiltonian circuit (= a listing of the nodes in some order) really is a Hamiltonian circuit, and whether a given Hamiltonian circuit has total length < d. It's fast to check whether a given valuation of the atoms of a formula actually makes the formula true.

So the real barrier to efficient solution of these problems is the search part.

Now for some problems, a clever search strategy has been invented, rendering them tractable. For example, consider the following yes-no problem.

Problem	**instances**	**yes-instances**	**no-instances**
Spanning tree weights	all pairs (G,d), where G is a connected weighted graph, and d>0 is a number.	those pairs (G,d) where G has a spanning tree of weight <d	those pairs (G,d) where all spanning trees of G have weight ≥d

This can be solved by finding a MST (§8) and comparing its total weight with d. Not only is the length comparison is fast, but there's also a fast (log linear) algorithm to find a MST. It's as though we have a metal detector that guides us to the needle if there is one, so we don't need to examine the whole haystack.

For other problems such as HCP and TSP (type (∃)), no clever search strategy has yet been found, and no tractable solutions are known[3].

So problems subdivide further:

(∃1) Problems of type (∃) for which a clever (i.e., p-time) search strategy is available.

(∃2) Problems of type (∃) for which no clever search strategy is known.

[2] In fact if B, C have s, t atoms (all different) and are of length ≤ 6s-5, 6t-5 respectively, then (B∧C), (B→C), (~B∨~C) etc. have s+t = r atoms and are of length ≤ 6s-5+6t-5+5 = 6r-5. This gives even more formulas.

[3] There are fast probabilistic and genetic 'solutions' to TSP that are sometimes very effective, but they are not guaranteed to solve the problem.

The type (∀) problems subdivide similarly. E.g., *is every spanning tree of length >d?* is type (∀1), as we can find a MST and see if it weighs in at more than d. The table gives more examples.

	∃) Is there a needle?	**∀) Is there no needle?**
1) a fast search strategy is known	'Does the weighted graph G have a spanning tree of weight < d?'	'Is every spanning tree of the weighted graph G of total weight ≥ d?'
2) no fast search strategy is known as yet	TSP, HCP, PSAT, 'is n composite?' [and all NP-complete problems]	'Is n prime?' 'Is every Ham. circuit of length >d?' 'Is the given formula unsatisfiable?'

The point is that we know where to look for a tree of length ≤d if there is one. We can narrow down the search space to a small size that can be searched tractably, given that we have a fast algorithm to check that a given possible solution to the problem is actually a solution.

Now, importantly, if we can narrow down the search space in this way, then it's just as easy to find a solution as to check that there isn't a solution! Both involve going through the same shrunken search space. So type (∃1) and (∀1) problems are equally easy (they are tractable). This really follows from the fact that P is closed under complement (§9.3.3.2). Once we know a type (∃) problem is tractable (in P), its complement, a type (∀) problem, will also be tractable.

The type (2) problems seem to be intractable, but the only (!) source of intractability is our inability to find a clever strategy to replace exhaustive search. They would become tractable if we had a good search strategy.

Which problems become tractable if we discount[4] the cost of exhaustive search? This is a kind of science fiction question: what would it be like if …?

The answer is: over a thousand commonly encountered ones: the *NP problems*. They are of type (∃); their complements, of type (∀), would simultaneously become tractable, too. And could there really be a clever search strategy for these problems, one that we've all missed? Most people think not, but no-one is sure. We explore these interesting questions in the next section, using a new kind of Turing machine.

[4] There are several ways of doing the discounting, depending on what kind of information we want from the exhaustive search. For example, type (∀) problems intuitively seem harder than type (∃), just as finding an algorithm for a given problem is easier than proving there's no algorithm that works. But often it's not known whether they really are harder, in the sense that the time complexity of a Turing machine solution is necessarily higher. The simplest way is to discount the cost of search in type (∃) problems — those involving simply seeing whether there exists (∃) a solution among many possibilities — and this is the approach we will take in §10.

10 Non-deterministic Turing machines

A *non-deterministic Turing machine* is one that can make choices of which 'instruction' to execute at various points in a run. Such a machine gives us an exhaustive search for free, because by using a sequence of choices it can simply guess the solution (which part of the haystack to check). We don't specify which choices are made, or how, because we are interested in solving problems when we're given a search for free, not in the mechanism of the search. We can view the non-deterministic parts of N as 'holes', waiting to be filled by a clever search strategy if it's ever invented. In the meantime we can still study the behaviour of N — by studying non-determinism itself.[5]

So: a non-deterministic Turing machine is like an ordinary one, but more than one instruction may be applicable in a given state, reading a given symbol. If you like, the instruction table δ can have more than one entry for each pair $(q,a) \in Q \times \Sigma$. In this situation the machine can choose which instruction to execute. This is why these machines are called non-deterministic: their behaviour on a given input is not determined in advance.

10.1 Definition of non-deterministic TM

Formally, a *non-deterministic Turing machine* (NDTM) is a 6-tuple N = $(Q,\Sigma,I,q_0,\delta,F)$ as before, but now:

$$\delta : (Q \setminus F) \times \Sigma \to 2^{Q \times \Sigma \times \{\pm 1, 0\}} \text{ is a (total[6]) function.}$$

Here, 2^X is the set of all subsets of X. E.g., if X = {1,2} then 2^X = {∅,{1},{2},{1,2}}. If X has n elements then 2^X has 2^n elements, which explains the notation 2^X.

[5] At first sight, a guessing machine may allow even more for free than exhaustive searches. We'll see in §10.3.3 that in fact it doesn't!

[6] We can take δ to be total, rather than partial, because if we want there to be no applicable instruction in state q when reading symbol a, we define δ(q,a) = ∅ (empty set).

10.1.1 Operation of NDTM

A non-deterministic Turing machine N has a one-way infinite tape and a single head, as usual. (We can consider 3-tape variants etc., if we want.) N begins in state q_0 with its head in square 0. In state q and reading symbol a, N works like this:

- If $q \in F$ then N halts and succeeds.
- Otherwise, N can go into state q', write symbol a', and move the head in direction $d \in \{\pm 1, 0\}$, for any $(q',a',d) \in \delta(q,a)$.
- N has free choice as to which $(q',a',d) \in \delta(q,a)$ to take.
- $\delta(q,a) = \varnothing$ means that there is no applicable instruction. In this case N halts and fails. N also halts and fails if its head tries to move left off the tape.

Of course, there are NDTMs N such that $\delta(q,a)$ always either contains a single triple (q',a',d) or is empty. Such an N will behave like an ordinary Turing machine — deterministically. So the ordinary Turing machine is a special case of a non-deterministic Turing machine. We have again generalised the definition of a Turing machine, as we did with the n-tape Turing machine in §3 (the n-tape model generalises the ordinary one, as n could be 1).

10.1.2 Input & output of NDTM

As usual, the input of a NDTM N is taken to be the contents of the tape before the run, up to but not including the first blank.

However, we have a problem in defining the output of N for a given input. This is because N may well have more than one successful run (one in which it halts and succeeds) starting with input w. For each such run we may get a different 'output' — so which one is the real output of N on w?

We could define N's output to be the set of all words x of Σ such that after some successful run of N, the tape contents (up to the first blank) are x. I.e., $f_N(w)$ would be the set of all possible outputs of N on w. So f_N would be a (total[7]) function $f_N : I^* \to 2^{\Sigma^*}$.

This is getting complicated. Again we simplify matters by considering only yes/no problems. With these, N's 'output' is just *yes* or *no*, according to whether N accepts or rejects the input. So we don't need to consider its output word(s) at all.

Therefore we do not define the output of a NDTM. But we do need to revise Definition 9.1.3 (acceptance/rejection of input) to cover NDTMs.

[7] If there is no successful run of N on w, then $f_N(w) = \varnothing$. So again we can take f_N to be a total function.

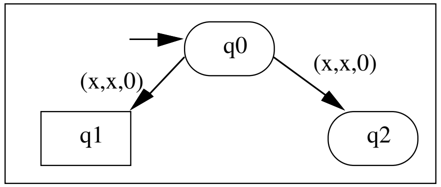
10.1.3 Accepting & rejecting for NDTMs

A Turing machine N (non-deterministic or otherwise) is said to *accept* the input $w \in I^*$ if there exists a successful run of N (one in which N halts and succeeds) given w as input. If all runs on input w end in a halt-and-fail, N is said to *reject* w. This agrees with our previous definition when N is deterministic[8].

The definition of *solving* a yes/no problem is the same as for deterministic Turing machines (§9.1.3): N should accept the yes-instances and reject the no-instances.

10.1.3.1 Exercise

What's wrong with this argument: the non-deterministic Turing machine in Fig. 10.1 solves any yes/no problem, because given a yes-instance, it can move to state q1, and given a no-instance it can move to state q2. As it's non-deterministic we don't need to say how it chooses! [Look at the definition of *solving*.]



10.1 A TM to solve any problem??

10.2 Examples of NDTMs

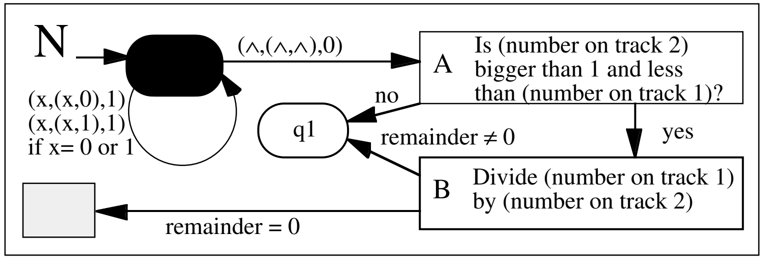
Let's see some examples. In each one, notice how the machine first guesses a solution, and then checks that it really is a solution. Both processes are fast.

10.2.1 Non-primality testing

Given a number n, is it composite? We can make a 2-track NDTM N that:

(i) guesses a number m with 1<m<n;
(ii) divides n by m (deterministically, in p-time);
(iii) if there's no remainder then it halts and succeeds ('yes'); otherwise it halts and fails ('no').

[8] If N is deterministic then it has only one run on w. So by definition, N accepts w if N halts and succeeds on input w, and rejects w if it halts & fails on w. This is just as before (§9.1.3).



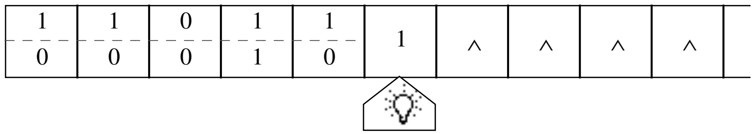
10.2 A NDTM solving non-primality problem

The input is a binary number, on track 1. N non-deterministically writes out a number on track 2. Notice how two instructions are applicable in state q0 when reading 0 or 1 on track 1: N can write either 0 or 1 in track 2. But if it's reading a $\wedge$ in track 1, it must write $\wedge$ and go to box A. This means that it writes 0's and 1's until the end of the input: it can't run forever. It ends with a $\wedge$ and goes to box A.

Now it halts and succeeds iff the number it wrote was not 0, 1 or $\geq$ the input (subroutine A), and there's no remainder on dividing the input by the guessed number (subroutine B). A and B can be done deterministically in p-time, using ordinary long division, etc.

Following Fig. 10.2 through, the only way that N can halt and succeed is by finding a factor. If the input is composite, then N could possibly halt and succeed — it has only (!) to guess the right factor. But if the input is prime, N cannot halt and succeed. It can only halt and fail. Hence N solves the non-primality yes/no problem.

In Fig. 10.3, N (in state q0) has guessed '10?' (in binary) as a factor of 55 (110111). It's about to guess the last digit, '?'. If it chooses 0, leaving '100' = 4, it will halt & fail after B. But it could pick 1, leaving '101' = 5. Then it will halt and succeed. So it can possibly halt & succeed on this input, so 55 is not prime.



10.3 Will N be lucky?

Exercise: why doesn't this approach work for the problem 'is n prime?'

10.2.2 Travelling salesperson problem

An instance consists of the distances between the cities, and the bound d, all coded in some sensible manner (no unary notation, etc.). The answer is 'yes' if there is a route of length $\leq$ d, and 'no' otherwise.

We can design a NDTM N that:

(i) non-deterministically guesses a route around the cities;
(ii) works out its length (it can do this deterministically, in p-time);
(iii) if the length is $\leq$ d then it halts and succeeds ('yes'); otherwise it halts and fails ('no').

Clearly, if there is a route of length $\leq$ d then N could possibly halt and succeed — it only has to guess right in (i). But if there is no such route, N cannot halt and succeed. It can only halt and fail. Hence N solves TSP.

10.2.3 PSAT

Problem: Given a formula A of propositional logic, is A satisfiable?

Using a NDTM we can guess an assignment v, and then easily check in p-time whether or not v(A) = true.

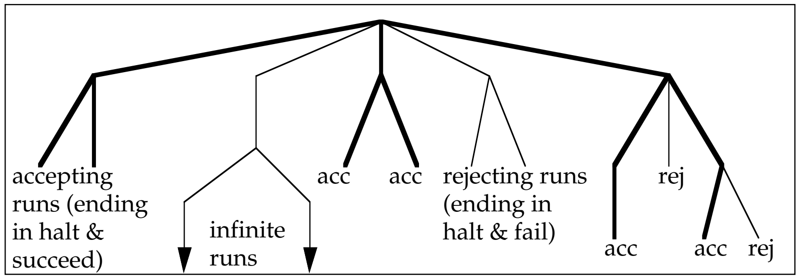
10.2.4 Exercise

How can we solve the Hamiltonian circuit problem using a fast NDTM?

10.3 SPEED OF NDTMS

What does it mean to say that a non-deterministic Turing machine N runs in p-time? To answer this we have to define $t_N(w)$ and $time_N(n)$, as we did for deterministic TMs in §9.2.1. But because N can guess, there are now many possible runs of N on any given input $w \in I^*$. The runs are of three kinds:

(i) successful (*accepting*) runs that end in a halt & succeed;
(ii) *rejecting runs* (ending in a halt & fail);
(iii) infinite runs in which N never halts.



10.4 Tree of runs of a NDTM on some input

There are several ways of defining the run time of N.[9] We will take the simplest:

10.3.1 Run time function for NDTMs

$\text{time}_N(n)$ is the length of the longest possible run (i.e., the depth of the tree) that N could make on any word of length n. So $\text{time}_N(n) \leq \infty$.

Again, this definition reduces to that of §9.2.1 when N is deterministic, as it then has only one run on any input.

10.3.1.1 Exercise for mathematicians (cf. König's tree lemma)

Is this well-defined? What is $\text{time}_N(n)$ if N has arbitrarily long finite runs on words of length n, yet no infinite run? Then there'd be no longest run. But in fact this can't happen! Show that if for all r≥0 there is a run of N of length at least r, then N must also have an infinite run.

10.3.2 p-time NDTMs

We say that a non-deterministic Turing machine N *runs in polynomial time (p-time)* if there is a polynomial p(n) with positive coefficients such that $\text{time}_N(n) \leq p(n)$ for all n≥0. I.e., no run of N on any input of length n lasts longer than p(n) steps. This is as before. As for deterministic Turing machines, a p-time non-deterministic Turing machine always halts (cf. §9.2.2.3).

10.3.3 The class NP of problems

We define *NP* to be the class of all yes/no problems A such that there is some NDTM N of p-time complexity that solves A: N accepts all the yes-instances of A, and rejects all the no-instances.

The class NP is very important. It is the class of 'type (∃) problems' (see §9.3.6) that would succumb to a clever search strategy. It contains many commonly occurring problems of practical importance. For instance, each NDTM described in the examples of §10.2 has p-time complexity, so PSAT, TSP, HCP, and compositeness testing are all in NP. So is primality testing.

We said that an NDTM is a Turing machine with 'holes' waiting to be filled by a clever search strategy. In effect, all the hoped-for strategy needs do is to search the tree of Fig. 10.4, to find an accepting run. Thus the remark in footnote 5 is justified!

[9] We could follow Rayward-Smith and ignore the no-inputs. For most purposes (eg. Cook's theorem) the choice doesn't matter very much.

10.3.4. P=NP?

Clearly P ⊆ NP (because p-time deterministic TMs are a special case of p-time NDTMs). It is not known whether P=NP — this is probably the most famous open question of computer science. Most computer scientists believe that P≠NP, but they may all be wrong.

Unlike Church's thesis, the question P=NP is precisely stated. Whether P=NP or not shouldn't be a matter of belief. We want to prove it, one way or the other. A lot of work has been done, but no-one has yet published a proof.

10.4 SIMULATION OF NDTMS BY ORDINARY TMS

We now show that although NDTMs seem faster, they can't solve any more Y/N problems than deterministic Turing machines. This gives our last hefty chunk of evidence for Church's thesis.

The idea is very simple. We saw that NDTMs are just a quick way of doing an exhaustive search. So now we'll do the exhaustive search in full — deterministically.

As usual, we check the easy part first: NDTMs can do anything that ordinary TMs can. This is true because a deterministic Turing machine is a special case of a NDTM.

Formally, given an ordinary Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$, we can convert it into a NDTM, $N = (Q,\Sigma,I,q_0,\delta',F)$ as follows. For each $q \in Q$ and $a \in \Sigma$:

- If $\delta(q,a)$ is defined then set $\delta'(q,a) =_{\text{def.}} \{\delta(q,a)\}$;
- If $\delta(q,a)$ is undefined, set $\delta'(q,a) =_{\text{def.}} \varnothing$.

Then N behaves exactly as M does: deterministically! So N solves the same problem as M.

Now we check the hard part.

10.4.1 Theorem

Any yes/no problem solvable by a NDTM can also be solved by a deterministic Turing machine.

Proof sketch: We will show that any NDTM N can be simulated by a 3-tape deterministic Turing machine M. M will accept/reject the same input words as N.

The idea is that M simulates all possible runs of N on the given input. At each point of the run of N there is possibly a fork, when N chooses which of several instructions to execute. So the runs of N form the branches of a tree (see Fig. 10.4). M will construct and search this tree in a breadth first fashion, halting and succeeding as soon as it finds a branch that ends in a 'halt & succeed' for N.

The search must be breadth-first because branches corresponding to runs of N that never terminate (e.g., loops) are infinite. A depth-first search would take M off down such a branch forever, never to return. But the next branch along might represent a 'halt & succeed' run, which is what M is looking for. Breadth-first searches are expensive in memory usage. Fortunately, Turing machines have lots of memory!

1. Each node of the tree corresponds to the *configuration* of N when 'at' that node, namely the following information:
   - the current state of N
   - the current contents of N's tape
   - the position of N's head on its tape.

The configuration determines the possible next moves of N, and so determines which nodes lie immediately below the current node in the tree. Note that the tree is finitely branching (i.e., there are only finitely many children of each node), because N has only finitely many choices in any configuration.

2. We can represent a configuration of N by a word s*t, where:
   - s is a word representing the current state of N (e.g., the decimal state number)
   - * is a delimiter
   - t is a '2-track word' with N's current tape contents (up to the last non-blank character) in track 1, and an X in track 2 marking the current head position (cf. §3). Note that t could be arbitrarily long but is always finite.

If q is a state of N, w a word of Σ, and $k \geq 0$ a number, let us write *config(q,w,k)* for the word s*t corresponding to the configuration where N is in state q, w is on the tape, and the head of N is over square k.

3. Any level of the tree of runs of N can be represented by a finite sequence of configurations of the form s*t, each separated from the next by another delimiter, say **. Using this data storage method, M can simulate N.
4. Initially, N's input word w is on tape 1 of M. M replaces it with config(q_0,w,0), using tape 2 for scratch work (copying etc.).
5. After n cycles, the tree has been explored to depth n (root = depth 0). Tape 1 of M will hold all those labelled configurations s*t attached to nodes in level n of the tree.
6. Now, for each step of N, M updates tape 1.
7. First M checks to see if any configuration so far on tape 1 is a 'halt & succeed' for N (i.e., it involves a halting state for N). If M finds any, it also halts and succeeds: the search is over.

8. Otherwise, M moves successively through the configurations s*t on tape 1. Each s*t corresponds to a node at level n of the tree. It may have several *child nodes* at level n+1: there will be one for each possible move of N in the configuration s*t. For each such possible move of N in turn, M calculates the configuration of the corresponding 'child' node: the new label s'*t'. M can do this because it 'knows' the instruction table of N.[10] If s'*t' would involve a negative head position, it is invalid and is ignored.[11] Otherwise, M appends it to the end of tape 2. Tape 3 is used for rough work.
9. So M works out all possible moves of N in each configuration s*t, one after the other. If there are no valid children (e.g., if N has no applicable instruction in this configuration) then no change is made to tape 2.
10. Having done all children of this configuration s*t, head 1 moves to the next configuration on tape 1, and the process repeats, the new children being appended to tape 2. And so on, through all configurations on tape 1.
11. When every configuration on tape 1 has been dealt with in this way, tape 2 holds the configurations corresponding to level n+1 of the tree. M can copy tape 2 back over tape 1 and go on to the next cycle (step 7 above). If tape 2 is empty, this means that there are no valid children. The tree has no level n+1, so M halts and fails.

Now if N accepts w then there is some successful run of N on w. So somewhere in the tree there's a configuration of the form

(†) *config(q,w,m)* , where q is a halting state of N.

M will eventually find it, and will also halt and succeed. So M also accepts w.

On the other hand, if N rejects w, then every run of N on w ends in a halt-and-fail. So (cf. §10.3.1.1) the tree will be finite, of depth n, say, with no configurations of the form (†). M will eventually try to construct level n+1 of the tree, which is empty. At that point, M halts & fails. So M rejects w too.

So N and M solve the same yes/no problem. This completes the proof.

[10] Alternatively we could use the technique of §4 and provide M with code(N) on another tape. This would allow M to simulate any NDTM with the right alphabet!

[11] NB. M does not halt and fail as soon as it finds a single run of N that halts and fails, because other runs of N may be successful.