1 The Complexity of Enumeration

1.1 Basics of complexity

The basic notions of computational complexity are now familiar concepts in most branches of mathematics. One of the main purposes of the theory is to separate tractable problems from the apparently intractable. Deciding whether or not $P = NP$ is a fundamental problem in theoretical computer science. We will give a brief informal review of the main concepts.

We regard a computational problem as a function, mapping inputs to solutions, (graphs to the number of their 3-vertex colourings for example). A function is *polynomial time computable* if there exists an algorithm which computes the function in a length of time (number of steps) bounded by a polynomial in the size of the problem instance. The class of such functions we denote by FP. If A and B are two problems we say that A is *polynomial time Turing reducible* to B, written $A \propto B$, if it is possible with the aid of a subroutine for problem B to solve A in polynomial time, in other words the number of steps needed to solve A (apart from calls to the subroutine for B) is polynomially bounded.

The difference between the widely used P and the class FP is that, strictly speaking, both P and NP refer to *decision* problems.

A typical member of NP is the following classical problem known as *SATISFIABILITY*, and often abbreviated to *SAT*.

SATISFIABILITY

Instance: Boolean formula in conjunctive normal form, $C_1 \wedge \ldots \wedge C_m$, where the C_i are clauses in the literals $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$.

Question: Is there an assignment of truth values to the variables x_i which satisfies the formula?

It is clear that *verifying* that a given assignment of values satisfies the clauses can be done in time which is a polynomial function of the input size.

This is the essence of NP; it is the class of decision problems which have solutions which can be *checked* in polynomial time. A formal definition will be given later but perhaps the easiest way to think of it and one which is particularly pertinent to counting problems is as follows.

Visualise a *nondeterministic Turing machine (ndtm)* as an ordinary Turing machine with an additional input which can contain a *certificate* or *witness*. This certificate will be used in the verification process, which is an

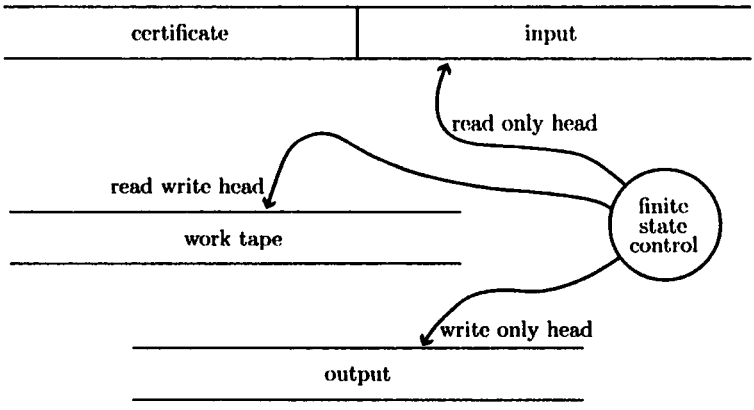

Figure 1.1. A nondeterministic Turing machine

ordinary Turing machine computation except that it is allowed to use the information in the certificate. This is the model used in Garey and Johnson (1979) and illustrated in Figure 1.1.

Thus in the case of *SATISFIABILITY* if the input x is a Boolean formula F, the certificate y might be a particular assignment of truth values and the working of the Turing machine is just the checking that with these values F is satisfied.

As another example, consider the language of composite integers. Formally this might be described by

COMPOSITE INTEGERS

Instance: Integer n expressed in binary.

Question: Is n composite?

The witness here might be a factorisation of n. The process of checking would be to verify that these factors multiplied together gave n, and this can be done in time which is bounded by a polynomial function of the size of the input, which in this case is $\lceil \log n \rceil$. Thus *COMPOSITE INTEGERS* belongs to NP. It is not obvious that the complementary language of *PRIME INTEGERS* belongs to NP. That it does, follows from a very nice paper of V. Pratt (1975) with the appealing title "Every prime has a succinct certificate".

The fundamental theorem of Cook (1971) is that NP has the hardest problems, known as the NP-complete languages. They are characterised by,

(i) $L \in NP$,

(ii) if there exists a polynomial time algorithm for deciding L, then $NP = P$.

In other words, L is NP-complete if it belongs to NP and for every other

$L' \in NP$, $L' \propto L$. There are now many hundreds of examples of NP-complete languages, including *SATISFIABILITY* and most of the difficult problems of graph theory such as deciding whether a graph G is k-colourable ($k \geq 3$) or has a Hamilton circuit.

A problem π is *NP-hard* if the existence of a polynomial time algorithm for π would mean there exists a polynomial time algorithm for some NP-complete language. Thus a language L is NP-complete if it is NP-hard and it belongs to NP.

Polynomial space, usually denoted by *PSPACE* consists of all languages recognisable by deterministic Turing machines which use an amount of space bounded by some polynomial function of the input size. It clearly contains NP, just run through all possible certificates, and the containment is thought to be strict.

Between NP and *PSPACE* is what is known as the *polynomial hierarchy*. This is the computational analogue of the Kleene arithmetic hierarchy of recursive function theory. A formal definition of it is found in Garey and Johnson (1979) but we can capture the set of languages in the polynomial hierarchy with the following definition. Define the class of languages PH by,

$$PH = \bigcup_{j=1}^{\infty} \Sigma_j^p$$

where the Σ_j^p are defined recursively by,

$$\Sigma_0^p = P,$$

$$\Sigma_{k+1}^p = NP^{\Sigma_k^p}$$

and where we are using standard notation for oracles. For a class Y of languages, NP^Y denotes the class of languages accepted by a nondeterministic polynomial time Turing machine with an oracle for any language in the class Y.

The containment relationships among these classes is shown in Figure 1.2, where *EXPTIME* denotes the class of decision problems solvable in time bounded above by $2^{p(n)}$ for some polynomial p.

Apart from knowing that P is a proper subset of *EXPTIME*, none of the other containments shown in Figure 1.2 is known to be strict.

1.2 Counting problems

The type of counting problem with which we shall be mainly dealing is of the following type.

For a given "universe", whether it be a d-dimensional lattice, graph, group or whatever, and a particular well defined "object" such as path, colouring, polygon, or automorphism on that universe, how many objects of given size are present?

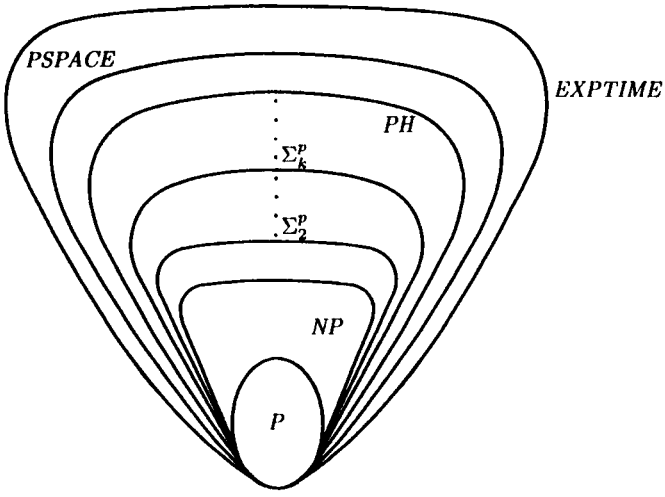

Figure 1.2.

In particular we shall be concerned with whether the associated counting problem can be done in time which is a polynomial function of the size of the universe or whether the problem is provably hard in some well defined sense.

We will not be concerned with problems such as the following.

(i) How many topologies are there on a set of size n?

(ii) How many nonisomorphic graphs are there on n vertices?

Nor will we be much concerned with the sort of enumeration problems covered in the monograph of Stanley (1986) where in most cases there is some hope of an answer in closed form.

Rather, we shall be concerned with the class $\#P$ of enumeration problems in which the objects being counted can be recognised in polynomial time. Since, if objects cannot be recognised it does not seem wise to try and count them we regard this property as justifying our calling $\#P$ the class of "sensible" counting functions.

Example. Let f be the function which maps any Boolean formula to its number of satisfying assignments. Then f is a member of $\#P$.

To see this, construct the ndtm M which, on input of a Boolean formula x, checks that a given assignment (or *witness* or *certificate*) y is an assignment of variables which makes x satisfiable. Thus M is a machine which can be interpreted as a function $g : \Sigma^* \times \Sigma^* \to \{0,1\}$, in such a way that

$$g(x,y) = \left\{ \begin{array}{ll} 1 & \text{if } y \text{ is a satisfying assignment to } x \\ 0 & \text{otherwise;} \end{array} \right\}$$

and also g is computable in polynomial time. □

In other words $\#P$ is the class of *functions* that count the accepting paths of nondeterministic polynomial time Turing machines.

Valiant's original definition of $\#P$ in 1979 was as follows.

A *counting Turing machine* is a standard nondeterministic Turing machine with an auxiliary output device that (magically) prints, in binary notation, on a special tape the number of accepting computations induced by the input. It has (worst-case) *time complexity* $t(n)$ if the longest accepting computation induced by the set of all inputs of size n takes $t(n)$ steps (when the Turing machine is regarded as a standard non-deterministic machine with no auxiliary device).

(1.2.1) Definition: $\#P$ is the class of functions which can be computed by counting Turing machines of polynomial time complexity.

Another (equivalent) definition is given below. Its structure is appealing because it highlights the structural similarities between $\#P$ and the decision classes NP and RP (random polynomial time).

Consider first the following definition of nondeterministic polynomial time NP.

(1.2.2) Definition: A language $L \in NP$ if there exists a polynomial p and a polynomial time algorithm, which computes for each input x and each possible certificate (= guess) (= witness) y of length $p(|x|)$ a value $R(x, y) \in \{0, 1\}$ such that

(i) if $x \notin L$ then $R(x, y) = 0$ for all y,

(ii) if $x \in L$ then $R(x, y) = 1$ for at least one possible certificate y.

(1.2.3) Definition: The class RP = *random polynomial time* is defined by replacing "at least one" in (ii) by "at least half of the possible certificates".

Then we can define $\#P$ to be the class of *functions* $f : \Sigma^* \to \mathbf{N}$, such that there exists $L \in NP$ and associated p, R as defined above, such that

$$f(x) = |\{y : R(x, y) = 1\}| \qquad x \in \Sigma^*.$$

Yet another formulation of $\#P$ is the following.

(1.2.4) Definition: Given the computation tree of a nondeterministic polynomial-time machine M, let $\mathrm{acc}_M(x)$ be the number of accepting computations of M on input x. Then

$$\#P = \{f : \Sigma^* \to \mathbf{N} \text{ such that } f = acc_M \text{ for some polynomial time non deterministic machine } M\}.$$

The language class $P^{\#P}$

We emphasise that $\#P$ is a class of functions, whereas the more familiar classes P and NP are classes of languages or sets. Thus, strictly speaking, $\#P$ is not directly comparable with the more familiar classes P and NP. Hence it is often more convenient to consider the associated class of languages $P^{\#P}$ defined by

$$P^{\#P} = \{L : L \text{ can be recognised in polynomial time by a Turing machine equipped with a } \#P \text{ oracle}\}.$$

Until the recent results of Toda (1989), to be discussed below, not much more was known about the class $P^{\#P}$ than that:

(1.2.5) $P^{NP} \subseteq P^{\#P} \subseteq PSPACE$.

Proof: An oracle which counts the number of satisfying assignments of a Boolean formula is obviously at least as strong as an oracle which decides whether a Boolean formula is satisfiable. Thus $P^{NP} \subseteq P^{\#P}$. To show that $P^{\#P} \subseteq PSPACE$ we note that an exhaustive search and check of all possible satisfying assignments can be carried out in polynomial space. □

1.3 # P-complete problems

We now define the class of $\#P$-hard (complete) functions in a way which is completely analogous to the familiar concepts NP-hard (complete).

Recall that FP is the class of functions which can be computed by deterministic polynomial time Turing machines.

(1.3.1) Definition: A problem π is *$\#P$-hard* if $\#P \subseteq FP^{\pi}$; it is *$\#P$-complete* if it is $\#P$-hard and it belongs to $\#P$.

Having defined $\#P$ and the concept of completeness and hardness for this class, it remains to exhibit some natural members. An obvious approach is to mimic the way in which NP-complete (hard) languages were found; this works, though as we shall see, a certain care has to be taken.

The strategy is to start with the counting counterpart of the generic NP-complete problem introduced by Cook (1971) in his seminal paper, and then to proceed in the same way as in NP-reductions but to ensure that the reductions in question preserve the number of solutions.

Accordingly we define:

 # TM-COMP: (Turing machine computation.)

Instance: Polynomial time nondeterministic Turing machine M, together with input $x \in \Sigma^*$.

Question: How many accepting computations has $M(x)$?

(1.3.2) Theorem: *# TM-COMP* is $\#P$-complete.

Proof. First show # *TM-COMP* $\in \#P$. To do this we need to exhibit a nondeterministic machine N such that given any pair $\langle M, x\rangle$, with M a ndtm and x an input to M, will verify that M accepts x. This is clear, since N just simulates the computation $M(x, y)$ where y is a witness to the acceptability of x. Since M is polynomially bounded, the length of the computation is bounded by a polynomial in $|\, x\,|$.

Now consider any other function in $\#P$, call it g. Then by definition there exists a nondeterministic machine M_g such that for all $x \in \Sigma^*$,

$$|\{y : M_g(x, y) = 1\}| = g(x).$$

Thus clearly

$$\#P \subseteq FP^{\#TM\text{-}COMP}.$$

□

We can now quickly build up a collection of $\#P$-complete problems as follows.

Take one of the standard NP-complete problems such as *SATISFIABILITY (SAT)* or *HAMILTON CIRCUIT* and define # *SAT* and # *HAMILTON CIRCUIT* as follows:

SAT

Instance: Boolean formula F in conjunctive normal form.

Question: How many satisfying assignments has F?

HAMILTON CIRCUIT

Instance: Graph G.

Question: How many Hamilton circuits does G have?

(1.3.3) # SAT is $\#P$-complete.

Proof: It clearly belongs to $\#P$ and Cook's original reduction which showed that SAT is NP-complete is easily checked to preserve the number of solutions. □

As pointed out first by Simon (1977) and Valiant (1979a), it seems to be the case that between most of the standard NP-complete languages there exist polynomial time transformations which preserve the number of solutions. In this way the original NP-completeness transformations developed by Karp (1972) show that the "natural" associated counting problem is also $\#P$-complete. Thus by tracing these transformations (and occasionally modifying them) we get for example:

(1.3.4) # *HAMILTON CIRCUIT* is $\#P$-complete.

(1.3.5) $\#k$-*COLOURINGS* is $\#P$-complete for $k \geq 3$.

We illustrate the proof technique by showing the easy reduction between *SAT* and *3SAT*-(instances of *SATISFIABILITY* in which each clause has 3 literals).

(1.3.6) *# 3-SAT* is $\#P$-complete.

Proof: Suppose F has a clause with $i \geq 4$ literals. In this clause replace any two of these literals, say $x_j, \bar{x}_k$, by a new variable y and form

$$F' = F \wedge C[(x_j \vee \bar{x}_k) \equiv y]$$

where $C[(x_j \vee \bar{x}_k) \equiv y]$ is the conjunctive normal form for $(x_j \vee \bar{x}_k) \equiv y$.

Then F' has exactly the same number of solutions as F and the added clauses will have at most 3-literals per clause. The result follows by induction. □

Thus, the reduction above is a *parsimonious* reduction, in that it preserves the number of solutions unlike the easier Turing reduction between *SAT* and *3-SAT* given in Cook's original paper.

A slightly tedious, but probably worthwhile exercise is to go through the reductions between the classical NP complete problems and to check whether they preserve the number of solutions.

For those which are not, it is normally not too difficult to produce modified reductions which have this property. As a result we are able to make the following slightly vague claim.

(1.3.7) The counting versions of the classical NP-complete problems seem usually to be $\#P$-complete.

However we urge caution here, there is no canonical definition of "classical" nor of the concept of a "natural" NP-complete problem. It is certainly not known to be true "that the counting counterpart of all natural NP-complete problems are $\#P$-complete".

As a further note of caution we emphasise that the idea of "natural counting problem associated with an NP-complete language" is *not* a well defined concept; consider, for example, the decision problem *HAMILTON CIRCUIT*.

The "natural" nondeterministic Turing machine M for testing whether G is Hamiltonian will use as a certificate of membership a Hamilton circuit of G and check its existence.

However an equally good nondeterministic machine M' will take a path of length say $n - [\log n]$ in G (where $n = |V(G)|$) and check (by exhaustive search) whether it can be extended to a Hamilton circuit. This also can be done in polynomial time and (apart from at the intuitive level) it is not easy to say why M' is less natural than M.

Despite this caveat we shall (wherever we think there is no confusion) adopt the notation $\#(A, x)$ to denote the number of "natural" witnesses to

x belonging to the language A. Thus for example, $\#(SAT, F)$ will denote the number of assignments to the variables of F which make it satisfiable.

At this stage we formally define a notion which has been pervasive in this discussion.

(1.3.8) Definition: A *parsimonious transformation* from problem A to problem B is a polynomial time transformation f such that if $\#(A, x)$ denotes the number of solutions of problem A with instance x, then $\#(A, x) = \#(B, f(x))$.

Berman and Hartmanis (1977) have used padding arguments to show the existence of parsimonious transformations between most of the natural search problems associated with the classical NP-complete sets. As mentioned earlier Simon (1977) and Valiant (1979a) noted that the reductions between the early classical NP-complete sets were (or could be easily modified so as to be) parsimonious. However, as we shall show in a later section, it is *not* true that there are parsimonious reductions between all pairs of natural NP-complete languages.

It turns out that very few of the counting problems which arise naturally are known to have polynomial time algorithms. Indeed more can be said, in that most counting problems seem to be hard (or complete) for the class $\#P$. Thus they will not have polynomial time algorithms unless $\#P = FP$ and this would be much more surprising than $NP = P$.

We close this introduction by listing some of the, relatively few, nontrivial counting problems which are known to have P-time algorithms.

(1.3.9) Spanning trees of a graph. Counting the number of spanning trees in an undirected graph G reduces to evaluating any cofactor of the Kirchhoff matrix of G.

(1.3.10) Spanning arborescences of a graph. A *rooted arborescence* of a digraph Γ is a subgraph H which when undirected is a tree and which has a distinguished vertex v, the *root*, such that all edges of H are directed away from v. The number of spanning arborescences originating at any root of Γ is given by $T(\Gamma) = \det K_r(\Gamma)$ where r is any vertex and $K_r(\Gamma)$ is the Kirchhoff matrix of Γ.

(1.3.11) Eulerian cycles. The number of Eulerian cycles in a digraph Γ with vertex set $\{1, \ldots, m\}$ is given by

$$C(\Gamma) = \prod_{j=1}^{m} (d_j^+ - 1)! T(\Gamma)$$

where d_j^+ is the outdegree of vertex j and $T(\Gamma)$ is the number of spanning arborescences originating at a vertex.

(1.3.12) Perfect matchings of a planar graph. Kasteleyn (1967) gives a polynomial-time algorithm which computes the number of perfect matchings of any planar graph. This is in terms of a Pfaffian and is based on his earlier solution of the dimer problem in classical statistical physics.

The method also extends to the classes of graphs a) not containing $K_{3,3}$ as a minor and b) what are described as Pfaffian graphs. However, since it is not known how to decide if G is Pfaffian this extension has limited use. For a beautiful discussion of this problem see Lovász and Plummer (1986).

1.4 Decision easy, counting hard

It is not difficult to find examples of problems where finding a solution is very easy but counting the number of solutions is $\#P$-hard. Here are some examples of varying degrees of interest.

 # UNSAT

Instance: Boolean formula F in conjunctive normal form.

Question: How many assignments of the variables make F false?

(1.4.1) $\#UNSAT$ is $\#P$-complete.

Proof: It is clearly in $\#P$. But if there are n variables in F, the number of unsatisfying assignments when subtracted from 2^n gives an algorithm for $\#SAT$. □

A less trivial example is the following:

 # MONOTONE BOOLEAN FORMULA

Instance: Monotone Boolean formula F (that is it contains no negated variable).

Question: How many satisfying assignments has F?

Clearly finding a satisfying assignment to a monotone Boolean formula is trivial. However the counting problem is hard as we now see.

(1.4.2) *# MONOTONE BOOLEAN FORMULA* is $\#P$-complete.

Proof. Any Boolean formula F may be rewritten in the form

$$F = G \wedge H$$

where G and H are both monotone.

Then, counting the number of solutions of $G \wedge H$ and of G gives the number of solutions of F.