4 An Introduction to Randomised Computation

4.1 Some examples

Suppose we were able to design an algorithm which may at certain points make a decision based on the value of a random bit but is otherwise deterministic, and which has the following behaviour: when presented with a Boolean formula φ, the algorithm runs in time polynomial in the size of φ and

(i) if φ is satisfiable, outputs "Yes" with overwhelmingly high probability (greater than $1 - 2^{-100}$, say);

(ii) if φ is not satisfiable, always outputs "No".

Then we could claim that, to all intents and purposes, SAT is tractable: surely, the above *randomised* (or *probabilistic*) algorithm is no more prone to error than is any physical computing device.

Unfortunately, no such algorithm is known for SAT or any other NP-complete problem. However, the development for other problems of randomised algorithms which are more efficient than the best known deterministic counterparts has given impetus to research in this area. A further reason for studying randomised computation is the additional insight which it affords into P =? NP and related questions: this stems from the fact that randomisation is in some sense intermediate between determinism and non-determinism, as we shall see.

Before introducing a formal framework for studying randomised algorithms, we discuss some examples:

Example 4.1. Perhaps the most influential probabilistic algorithms are those for primality testing due to Rabin (1976) and Solovay and Strassen (1977). Both depend on identifying a predicate $W(i, n)$ with the following properties:

(i) for $1 \leq i < n$, $W(i, n)$ can be computed in time polynomial in $\log n$;

(ii) if n is prime, then $W(i, n)$ is false for all i;

(iii) if n is composite, then $W(i, n)$ is true for at least half the integers i in the range $1 \leq i < n$.

Thus any i for which $W(i, n)$ holds may be regarded as a *witness* to the compositeness of n. The key fact is that many witnesses always exist, which immediately leads to the following algorithm (where n denotes the input, presented in binary):

> select m independent random integers $i_1, \ldots, i_m$ in the range $[1, n-1]$;
> **if** $W(i_j, n)$ holds for some $1 \leq j \leq m$
>   **then** output "n is composite"
>   **else** output "n is prime"

By property (i), this algorithm runs in time polynomial in $\log n$ for any fixed m. When n is prime, it always produces the correct answer since property (ii) ensures that no witness i satisfying $W(i, n)$ will be found. When n is composite, the algorithm gives an incorrect answer only if *all* m selected integers fail to be a witness. Since each integer fails to be a witness with probability at most 1/2 (by property (iii)), and the integers are chosen independently, this happens with probability at most 2^{-m}, which can be made negligibly small by suitable choice of m (e.g., $m = 100$).

The achievement in designing these algorithms lies in finding a predicate W with the desired properties. In the case of Rabin's algorithm, $W(i, n)$ takes the form:

(i) $i^{n-1} \neq 1 \pmod{n}$; **or**

(ii) there exists k such that $r = \dfrac{n-1}{2^k}$ is integral, and $1 < \gcd(i^r - 1, n) < n$.

Testing these conditions in time polynomial in $\log n$ is straightforward, using repeated squaring for the powering operations. The proof that composite numbers have many witnesses requires some fairly sophisticated number theory which we will not present here.

The above algorithms are of considerable practical value, enabling very large primes (having thousands of bits) to be certified with high probability in a matter of seconds.

Example 4.2. We look now at a simpler example which is easy to analyse in detail. Consider the following (rather artificial) problem:

INPUT: $n \times n$ real matrices A, B, C;

OUTPUT: "Yes" if $AB = C$, "No" otherwise.

The best known deterministic procedure for this problem is the obvious one involving matrix multiplication, which requires $\mathrm{O}(n^{2.81})$ arithmetic operations if Strassen's algorithm is used. (Faster methods for multiplying matrices do exist in theory but are of little practical value.) By contrast, the following elementary probabilistic algorithm due to Freivalds (1979) requires only matrix-vector multiplications and hence runs in time $\mathrm{O}(n^2)$:

select m independent random n-vectors $x_1, \ldots, x_m$ with entries from $\{1, -1\}$;
if $A(Bx_j) \neq Cx_j$ holds for some $1 \leq j \leq m$
  then output "No"
  else output "Yes"

The algorithm works in similar fashion to that of Example 4.1, by attempting to find a witness $x \in \{1, -1\}^n$ to the fact that $AB \neq C$. It is easy to verify that, for fixed m, the runtime of this procedure is $O(n^2)$, and that the correct answer will always be given in the case $AB = C$.

To see that the error probability is small when $AB \neq C$, consider the matrix $D = AB - C$, which must contain some row d with at least one non-zero entry, say the ith. Note that the equality $A(Bx) = Cx$ can hold only if the scalar product $d \cdot x = 0$. Now we claim that, for a random vector $x \in \{1, -1\}^n$, the probability that $d \cdot x = 0$ is at most $1/2$. For if $d \cdot x = 0$, then the vector $\hat{x}$ obtained from x by flipping the sign of its ith entry clearly *cannot* satisfy $d \cdot \hat{x} = 0$. Hence $d \cdot x = 0$ for at most half the possible vectors x. As a result, we see that $A(Bx) = Cx$ with probability at most $1/2$. Since the algorithm performs this test for m independent random vectors x, the overall probability of error is only 2^{-m}.

Example 4.3. Our third example, which is due to Schwartz (1980), has applications in the rapidly expanding area of computer algebra. Suppose we are given two polynomial expressions Q and R, each involving the n variables $x_1, \ldots, x_n$. We might well wish to know whether Q and R both denote the same polynomial, i.e., whether $Q \equiv R$.[2] Note that we are not assuming that Q and R are presented in any kind of standard form, so the answer will in general be far from obvious.

Clearly, any such question can be formulated as a test for $Q \equiv 0$ for some polynomial Q, so we focus on this problem. We assume also that Q has real coefficients. As a concrete example, consider the polynomial

$$Q(x_1, \ldots, x_n) = \prod_{\substack{i<j \\ i,j \neq 1}} (x_i - x_j) - \prod_{\substack{i<j \\ i,j \neq 2}} (x_i - x_j) + \cdots + (-1)^n \prod_{\substack{i<j \\ i,j \neq n}} (x_i - x_j). \quad (3)$$

How should we go about testing whether Q is identically zero?[3] The only way to be absolutely sure appears to involve expanding all the products in (3) by brute force, collecting terms and testing whether all coefficients in the resulting expansion are zero. Clearly such an algorithm is out of the question even for quite small values of n, as the number of terms in the expansion grows exponentially with n.

A randomised approach to the problem is based on the following idea. Suppose we select values for the variables x_i randomly and evaluate Q at this point. This can be done efficiently by substitution into (3). If the result is non-zero then we can be certain that $Q \not\equiv 0$; if the result is zero then *either* $Q \equiv 0$ *or* we have been rather unlucky in selecting a point which happens to be a zero of Q. This is so because, as we shall see below, if $Q \not\equiv 0$ then its zeros are very sparsely distributed in $\mathbb{R}^n$.

More precisely, the algorithm operates as follows, where d denotes the degree of Q:

[2]The symbol $\equiv$ here denotes identity, i.e., $Q \equiv R$ iff $Q(x_1, \ldots, x_n) = R(x_1, \ldots, x_n)$ for all values of the variables $x_1, \ldots, x_n$.

[3]This happens to be true for this example for all values of n. However, some (presumably non-mechanisable) insight is required to see why.

select n random integers $a_1, \ldots, a_n$ from the set $\{0, 1, \ldots, 2d-1\}$;
if $Q(a_1, \ldots, a_n) \neq 0$ **then** output "No" **else** output "Yes"

It is clear that this algorithm runs in time polynomial in the size of the representation of Q. (Note that a reasonable upper bound on d can be found by inspection.) Moreover, it always behaves correctly when Q is identically zero. In order to analyse its behaviour when $Q \not\equiv 0$, we require the following fact:

Proposition 4.1. *Suppose $Q(x_1, \ldots, x_n)$ is a polynomial, of degree at most d, that is not identically zero, and let S be any set of real numbers. Then the number of zeros of Q that lie in the set S^n is at most $d|S|^{n-1}$.*

We'll prove Proposition 4.1 presently, but first we'll use it to bound the error probability of the algorithm when $Q \not\equiv 0$. Proposition 4.1 tells us that if $Q \not\equiv 0$ then Q has at most $d|S|^{n-1}$ zeros in S^n; hence the probability that a randomly chosen point in S^n is a zero of Q is bounded above by $d|S|^{n-1}/|S|^n = d/|S|$. If we take $S = \{0, 1, \ldots, 2d-1\}$, as in the algorithm, this probability is at most $1/2$. Hence the algorithm will incorrectly assert that Q is zero with probability at most $1/2$. As in Examples 4.1 and 4.2, by choosing m random points rather than just one we can make the error probability less than 2^{-m}.

Proof of Proposition 4.1. We use induction on n. The statement holds trivially for $n = 1$, since any polynomial in one variable of degree d has at most d zeros. Now suppose $n > 1$ and view Q as a polynomial in x_n (whose coefficients are polynomials in $x_1, \ldots, x_{n-1}$) as follows:

$$Q(x_1, \ldots, x_n) = Q'(x_n) = \sum_{i=0}^{d} Q_i(x_1, \ldots, x_{n-1})\, x_n^i.$$

Since Q is not identically zero, there must be at least one i with $0 \leq i \leq d$ such that $Q_i \not\equiv 0$; let k be the largest such i. Note that the degree of Q_k is at most $d-k$. Now consider under what circumstances a point $(a_1, \ldots, a_n) \in S^n$ can be a zero of Q. There are two cases:

(i) $Q_k(a_1, \ldots, a_{n-1}) = 0$. In this case $Q'(x_n)$ could be zero for *all* values of x_n. But, applying the induction hypothesis to Q_k, the number of values $(a_1, \ldots, a_{n-1})$ for which this case can occur is at most $(d-k)|S|^{n-2}$. Hence the number of zeros of Q arising from this case is at most $(d-k)|S|^{n-1}$.

(ii) $Q_k(a_1, \ldots, a_{n-1}) \neq 0$. In this case $Q'(x_n)$ is a non-zero polynomial of degree k in x_n, and so has at most k zeros. The number of zeros of Q arising from this case is therefore at most $k|S|^{n-1}$.

Putting together cases (i) and (ii), we see that the total number of zeros of Q is bounded by $d|S|^{n-1}$, which completes the inductive proof. $\square$

4.2 Probabilistic Turing machines and the class RP

To make our discussion of probabilistic algorithms more precise, we now extend the basic Turing machine model to include a source of randomisation. A *probabilistic Turing machine* (PTM) is a Turing machine which possesses certain distinguished "coin-tossing" states. The operation of the machine is deterministic except that there are precisely two possible moves from any configuration containing a coin-tossing state, the choice of move being made according to the fall of an unbiased coin. (Realists may prefer to think in terms of a Zener diode, or some other source of "perfectly random" bits.) Thus the coin tosses induce a probability distribution on the computations of a PTM on any given input. All other features of the definition remain as before. In particular, a PTM is $T(n)$ time-bounded if every computation on inputs of size n halts within $T(n)$ steps.

We are now in a position to define a randomised analogue of the class P of tractable problems, which we call RP ("Randomised Polynomial time"). A language $L \subseteq \Sigma^*$ belongs to RP iff there exists a polynomial time-bounded PTM M such that, for all input strings $w \in \Sigma^*$,

(i) if $w \in L$ then $\Pr(M \text{ accepts } w) \geq 1/2$;

(ii) if $w \notin L$ then $\Pr(M \text{ accepts } w) = 0$.

Thus M always answers correctly on strings not in L, and does so with probability at least $1/2$ on other inputs.

Incidentally, it is possible to define RP without reference to probabilistic Turing machines and coins. Call a non-deterministic TM *balanced* if it has precisely two possible next moves from every non-halting configuration and all computations on any input have the same (finite) length. Then $L \in$ RP iff it is accepted by a polynomial time-bounded balanced non-deterministic TM M with the additional property that, whenever M accepts an input, at least half of its computations are accepting.

The algorithms of Examples 4.1, 4.2 and 4.3 demonstrate that the language COMP of composite numbers, the language of matrix triples $\langle A, B, C\rangle$ with $AB \neq C$, and the language of polynomial expressions which are not identically zero all belong to RP.

If a language lies in RP then it is tractable in a very strong sense, as we see from the following "powering" result:

Lemma 4.2. *Let $L \subseteq \Sigma^*$ belong to* RP*, and q be any polynomial. Then there exists a polynomial time-bounded PTM M such that, for all inputs $w \in \Sigma^*$,*

(i) *if $w \in L$ then* $\Pr(M \text{ accepts } w) \geq 1 - 2^{-q(|w|)}$;

(ii) *if $w \notin L$ then* $\Pr(M \text{ accepts } w) = 0$.

Proof. Let M_0 be a PTM recognising L with one-sided error probability at most $1/2$ as in the definition of RP. Then the desired machine M, on input w, simply performs $q(|w|)$ simulations of M_0 on w and accepts iff at least one of the simulations accepts. The probability that M_0 errs on *all* simulations is at most $2^{-q(|w|)}$. □

Note that the algorithms of Examples 4.1, 4.2 and 4.3 use precisely this device to reduce the error probability.

To illustrate the practical consequences of powering, iterating an RP-machine just 100 times suffices to reduce the error probability to 2^{-100}, which is presumably smaller than the probability of error due to hardware failure in most physical devices. Furthermore, using a powering argument as in Lemma 4.2 it is easy to see that the bound of $1/2$ in the definition of RP is essentially arbitrary, and can be replaced by any fixed constant ξ in the range $0 < \xi < 1$ without altering the class.

Exercise 4.1. Verify this.

Now let us consider where RP lies in relation to the other classes we know about. Since a PTM can be viewed as a non-deterministic TM, it should be obvious that

$$\mathrm{P} \subseteq \mathrm{RP} \subseteq \mathrm{NP}\,.$$

We might naturally ask whether either of these inclusions is strict. There is a lack of persuasive evidence either for or against RP = P. If the classes *are* unequal, then a reasonable candidate for membership in RP \ P is the language of polynomial (in)equivalences studied in Example 4.3. (The language COMP is no longer a candidate, since Agrawal, Kayal and Saxena showed COMP ∈ P.)

Turning to the other inclusion in the above chain, we would not expect that RP = NP since, as we have argued, RP corresponds to a plausible notion of effective tractability while we believe that NP contains intractable languages like SAT. As usual, the rôle of NP-completeness is central here, as it can easily be shown that

$$\mathrm{RP} = \mathrm{NP} \iff \mathrm{RP}\text{ contains some NP-complete language.}$$

Exercise 4.2. Verify this.

The next section will add independent justification to our claim that problems in RP are tractable. Even so, astonishingly, no one has been able to separate RP from *exponential* deterministic time; one problem here that it is impossible to diagonalise over randomised TMs, as was done for deterministic TMs in the proof of the Time-hierarchy Theorem.

Exercise 4.3. A language $L \subseteq \Sigma^*$ is said to belong to the class ZPP if there exists a polynomial time-bounded PTM M which, on input $w \in \Sigma^*$, may either accept, reject, or respond with "?" (meaning "don't know"), and

(i) $\Pr(M$ responds with "?"$) \leq 1/2$;

(ii) if $w \in L$ then M never rejects w;

(iii) if $w \notin L$ then M never accepts w.

Show that ZPP = RP $\cap$ co-RP.

The class ZPP defined in the last exercise is so called because the probability of an erroneous answer being produced is zero: the machine never accepts or rejects unless it is sure. Algorithms with this property are sometimes referred to as *Las Vegas* algorithms, to distinguish them from the weaker *Monte Carlo* algorithms corresponding to RP.

4.3 Randomisation and circuit complexity

This section is devoted to an elegant and remarkable result, which says that all languages in RP are tractable in the sense of having polynomial size circuits.

Theorem 4.3 (Adleman, 1978). *If $L \subseteq \{0,1\}^*$ belongs to* RP *then L has polynomial size circuits.*

The proof is simplified by the observation that L has polynomial size circuits if it can be recognised by a polynomial time-bounded deterministic TM equipped with a small extra piece of "advice" for each input size. (Note that this effectively makes the TM non-uniform.) We make this precise in a lemma.

Lemma 4.4. *Suppose there exists a polynomial time-bounded deterministic TM $\widehat{M}$ having the following property: for each n there is an "advice string" $a_n \in \{0,1\}^*$ of polynomially bounded length such that, for all $w \in \{0,1\}^n$,*

$$w \in L \iff \widehat{M} \text{ accepts } w\$a_n\,.$$

Then L has polynomial size circuits.

Proof. By Theorem 3.1, $\widehat{M}$ can be simulated by polynomial size circuits. Now, rather than treating the entire string $w\$a_n$ as part of the input, we *hard-wire* the advice string a_n into the circuit using constant input nodes. This can be done because a_n is fixed for all inputs of size n. The resulting circuits accept precisely the strings $w \in L^{(n)}$. □

Exercise 4.4. Prove that the converse of Lemma 4.4 is also true, i.e., that L has polynomial size circuits *if and only if* it is accepted by a polynomial time-bounded deterministic TM with polynomially bounded advice strings. (Hint: The advice is the circuit itself.)

Proof of Theorem 4.3. By Lemma 4.4, it suffices to find polynomially bounded advice strings a_n for L. Let M be an RP-machine recognising L, and p a polynomial bounding the time complexity of M. By Lemma 4.2, we may assume that the error probability of M on inputs of size n is at most $2^{-(n+1)}$. By adding artificial branching steps as necessary, we may identify computations of M on any input of length n with the set of binary strings of length $m = p(n)$ in an obvious way. The advice string a_n will be a computation of M which *covers* $L^{(n)}$, i.e., which is an accepting computation for every string in $L^{(n)}$. We prove the existence of such a computation using a probabilistic argument.

Consider a computation y selected uniformly at random from $\{0,1\}^m$. We will show that y is quite likely to cover $L^{(n)}$. For any $w \in L^{(n)}$ we have

$$\Pr(y \text{ is a non-accepting computation of } M \text{ on } w) \leq 2^{-(n+1)}$$

since y is chosen uniformly at random. Applying this inequality to each $w \in L^{(n)}$ in turn, we get

$$\Pr(y \text{ does not cover } L^{(n)}) \leq |L^{(n)}| \times 2^{-(n+1)} \leq 1/2.$$

Now, since y was chosen randomly, we conclude that *some* computation y which covers $L^{(n)}$ must exist. (Indeed, at least half of the possible computations cover $L^{(n)}$.)

Finally, the advice string a_n is simply a computation y which covers $L^{(n)}$, whose existence we have just established. The length of a_n is $p(n)$, a polynomial function of n. We can then design a TM $\widehat{M}$ satisfying the conditions of Lemma 4.4 as follows: on input $w\$a_n$, with $w \in \{0,1\}^n$, $\widehat{M}$ simply checks whether a_n corresponds to an accepting computation of M on w and accepts iff this is so. $\square$

Note that the construction given above for a_n is *not effective*: thus, although we know that suitable advice strings for L exist, we have no efficient way of finding them deterministically, or even of deciding whether a given string has the required advice property. Of course, this is just another manifestation of the non-uniform nature of the circuit model.

4.4 Approximability

The most stunning developments in computational complexity in the last decade have been in the area of approximation algorithms for optimisation problems. At first sight this topic seems to have no connection with the others considered in the chapter. It turns out, however, that the familiar class NP has an an alternative probabilistic formulation, and that this alternative formulation is the key to understanding why some optimisation problems are difficult even to approximate. The equivalence of the alternative definition of NP to the usual one is a deep result, and we shall not attempt even to sketch the proof. However, we shall be able to examine the implications of the equivalence for approximability.

The classical theory of NP-completeness deals with languages or, equivalently, decision problems. To force them into this framework, optimisation problems are conventionally re-expressed as decision problems: instead of asking "what is the largest clique?" in a graph G we specify a bound b and ask "is there a clique of size at least b?". At one level, this change of view makes no difference; if we could answer the decision problem, then, by binary search, we could solve the optimisation problem. In this section, however, we are interested in how well an optimisation problem can be approximated, so we cannot any longer reduce everything to a straightforward decision problem.

The practical motivation for this study is clear. The clique problem—in common with most other optimisation problems—is NP-complete, so we should not waste too much time looking for a polynomial-time algorithm that finds an optimum (i.e., largest) clique in an arbitrary graph G. On the other hand, what about a polynomial-time algorithm that finds a clique who size is at least half that of the optimum? Such an algorithm is not ruled out by the classical theory of NP-completeness, and might be good enough in certain practical situations. So it is of interest to know, for each NP-complete optimisation problem, how closely the optimum solution can be approximated in polynomial time.

The general setting is as follows. For each instance x of an optimisation problem, there is an associated set $S(x)$ of feasible solutions. Each solution $y \in S(x)$ has an associated integer cost $c(y)$. The optimum cost is then $\mathrm{OPT}(x) = \min\{c(y) : y \in S(x)\}$ for a minimisation problem, or $\mathrm{OPT}(x) = \max\{c(y) : y \in S(x)\}$ for a maximisation problem. (For a maximisation problem, "benefit" or "profit" might be a more appropriate word than "cost," but we shall stick with "cost" for the sake of uniformity.) For example, the instance x might be the encoding of an undirected graph G, the feasible solutions $S(x)$ might be the set of all cliques in G, and the cost of a feasible solution (clique) the number of vertices in the clique. The sort of problem we want to address in this section is the following. Fix a number $0 < \alpha < 1$. Given an instance x, find a feasible solution $y \in S(x)$ such that $c(y) \geq \alpha\,\mathrm{OPT}(x)$. (This assumes we are dealing with a maximisation problem.) The number α defining the quality of the solution is sometimes called the *performance ratio.*

Let us study the issues in the context of MAXSAT, the optimisation version of SAT: given a CNF formula φ, find an assignment of truth values to the variables of φ that makes as many as possible of the clauses of φ true.

Exercise 4.5. Set the problem MAXSAT within the general framework for optimisation problems just given. What are the set of feasible solutions and the cost function c?

There is a simple polynomial-time randomised approximation algorithm that solves MAXSAT with expected performance ratio $\frac{1}{2}$. Let the variables of the formula φ be $x_1, x_2, \ldots, x_n$. For each variable x_i independently, set x_i true with probability $\frac{1}{2}$ and false with probability $\frac{1}{2}$. Focus attention on a particular clause C

of φ, and suppose that C has k literals. The probability that C is made true by the assignment is $1 - 2^{-k} \geq \frac{1}{2}$. Thus the expected number of clauses made true is at least $m/2$ where m is the number of clauses. Since $\mathrm{OPT}(\varphi)$ is certainly bounded by m, the expected performance ratio is at least $\frac{1}{2}$. Note that if we are given the additional information that every clause has at least k literals then the performance ratio rises to $1 - 2^{-k}$, e.g., to $\frac{7}{8}$ for "MAX=3SAT," the restriction of MAXSAT to instances in which every clause contains *exactly* three distinct literals.

Exercise 4.6. Present a *deterministic* polynomial-time approximation algorithm for MAXSAT that achieves the same performance ratio $\frac{1}{2}$. (Hint: since the *expected* number of clauses made true by a random assignment is at least $m/2$, there must be at least one assignment that actually makes at least $m/2$ clauses true. Try setting the variables $x_1, \ldots, x_n$ to true or false in sequence.) How about a *deterministic* polynomial-time approximation algorithm for MAX=3SAT, again achieving performance ratio $\frac{7}{8}$? (This is harder; if you crack this, you will have rediscovered the "method of conditional expectations.")

The method just described (set the variable to true or false at random) is very naive, and one feels it should be possible to do better. Goemans and Williamson recently introduced an important new technique based on "semidefinite programming relaxation," and showed how it could be used to achieve performance ratio 0·758 for MAXSAT. Subsequently, Trevisan, Sorkin, Sudan and Williamson combined this technique with another idea to achieve performance ratio 0·801 for "MAX3SAT" (every clause has *at most* three literals).

How close to 1 can the performance ratio for MAXSAT be pushed? Could it be the case that, for every $\varepsilon > 0$, there is a polynomial time algorithm that achieves performance ratio $1 - \varepsilon$? (This situation is described by the phrase *Polynomial Time Approximation Scheme* or PTAS.) We shall see presently that the answer is no: there is a definite upper bound on the performance ratio that can be achieved by any polynomial-time approximation algorithm (provided, as usual, $\mathrm{P} \neq \mathrm{NP}$).

Recall the characterisation of NP from Lemma 2.1: a language L is in NP iff there exists a polynomial p and a polynomial-time (deterministic) TM V such that

$$L = \big\{x : \text{there exists } y \in \{0,1\}^{p(|x|)} \text{ such that } V \text{ accepts } \langle x, y\rangle\big\}. \tag{4}$$

For convenience, y ranges over *binary* words, which is clearly not restrictive. A word y that causes V to accept the pair $\langle x, y\rangle$ can be though of as a *certificate* of x's membership in L, and the TM V itself can be thought of as a *verifier* (for certificates).

It is possible to restrict the power of the verifier V without affecting the language class defined by (4). For example, if we insist that V is log-space bounded, then (4) still defines the class NP. The astonishing fact is that the power of V can be restricted until it is computationally very weak indeed, and still (4) continues to define the class NP. In fact, we shall consider verifiers that look at only a constant number of bits of the certificate before coming to a decision! Naturally, because the verifier does not look at the whole certificate it will sometimes be misled into accepting a false certificate. However, the verifier will be right "most of the time." Implicit in this statement is the notion that V must be a probabilistic TM.

Now for some detail. We define a $(\log n, 1)$-*restricted verifier* to be a probabilistic TM V which on input $\langle x, y\rangle$ behaves as follows:

- V creates a string r of $\lceil c \log |x| \rceil$ random bits (say, by flipping a fair coin $\lceil c \log |x| \rceil$ times), where c is a constant. From this point on, V behaves entirely deterministically.
- V computes a set $Q(x, r) = \{i_1, \ldots, i_k\}$ of integers, where k is some constant.
- V now performs some polynomial-time computation based on x, r and $y_{i_1} \ldots y_{i_k}$, at the end of which it either accepts or rejects.

The "$\log n$" part of the title refers to the number of random bits used (i.e., $O(\log n)$), and the "1" to the number of bits of the certificate y that are examined (i.e., $O(1)$ or constant). We say that V is a $(\log n, 1)$-*restricted verifier for language L* iff

- for all $x \in L$ there exists a word $y \in \{0,1\}^{p(|x|)}$ such that V always accepts $\langle x, y\rangle$;
- for all $x \notin L$ and for any word $y \in \{0,1\}^{p(|x|)}$, V rejects $\langle x, y\rangle$ with probability at least $\frac{1}{2}$.

The key theorem is:

Theorem 4.5 (Arora, Lund, Motwani, Sudan and Szegedy, 1992). *A language L is in* NP *iff it has a* $(\log n, 1)$*-restricted verifier.*

Armed with this alternative characterisation of NP, we can prove a non-approximability result for MAXSAT.

Theorem 4.6. *There is a constant $\alpha > 0$ such that there is no polynomial-time approximation algorithm for* MAXSAT *achieving performance ratio $1 - \alpha$, unless* P = NP.

Proof. Consider any NP-complete language L, and let V be a $(\log n, 1)$-restricted verifier for L. We shall demonstrate that the existence of a polynomial-time approximation algorithm for MAXSAT achieving performance ratio $1 - \alpha$, for $\alpha > 0$ sufficiently small, would imply $L \in \mathrm{P}$, and hence $\mathrm{P} = \mathrm{NP}$.

So suppose that we have such an approximation algorithm for MAXSAT, and let x be a word we wish to test for membership in L. Consider the operation of V on input $\langle x, y\rangle$ and random bits r. (Although x is fixed, the random bits r and certificate y will of course vary.) The combination of x and r picks out a set of k indices $Q(x, r)$, as specified in the definition of $(\log n, 1)$-restricted verifier. The subsequent behaviour of V depends, aside from x and r, only on k bits $(y_i : i \in Q(x, r))$ of the certificate y. The question of whether V accepts can be expressed as a boolean function, say $\varphi_r(y)$, of these variables. Thus, for given random bits r, the function $\varphi_r(y)$ is true iff V accepts the certificate y. Note that although we have written $\varphi_r(y)$ as a function of y, it in fact depends only on k bits of y. Since $\varphi_r(y)$ has only k variables—recall that k is a constant—it can be expressed as a CNF formula of bounded size. In fact, the number of clauses in $\varphi_r(y)$ is at most $K = 2^k$, another constant.

Now consider the formula $\Phi(y) = \bigwedge_r \varphi_r(y)$, where r ranges over all binary words of length $\lceil c \log |x| \rceil$. (So instead of choosing r at random, we are exhaustively testing all possibilities.) Since r has length $O(\log n)$, the number of possible choices for r is polynomial in n. Thus $\Phi(y)$ has only a polynomial (in n) number of clauses, and each clause is of bounded size (independent of n).

There are two cases to consider. If $x \in L$, then, by definition of $(\log n, 1)$-restricted verifier, there is a certificate y such that V accepts $\langle x, y\rangle$ whatever random bits r are selected. This certificate y makes every formula $\varphi_r(y)$ true, and hence $\Phi(y)$ itself true. Note that $\Phi(y)$ is in CNF, i.e., is a conjunction of clauses, and that *every* clause is made true by y. On the other hand, if $x \notin L$, then, again by definition of $(\log n, 1)$-restricted verifier, for every certificate y, the verifier V rejects $\langle x, y\rangle$ on at least half the random bit sequences r. Equivalently, for every y, at least half the subformulas $\varphi_r(y)$ are false. Since at least half the subformulas $\varphi_r(y)$ contribute at least one false clause to $\Phi(y)$, the proportion of false clauses in $\Phi(y)$ is at least $1/2K$. So we have a dichotomy: if $x \in L$ then we can make all clauses in Φ true; whereas if $x \notin L$ we must fail to satisfy a proportion $1/2K$ of them.

The key point is this: to determine whether $x \in L$ we do not need to know whether Φ is satisfiable; it is enough to know if it is "nearly satisfiable." Recall that we postulated at the outset a polynomial-time approximation algorithm for MAXSAT achieving performance ratio $1 - \alpha$. Now choose $\alpha = 1/3K$. If $x \in L$ then Φ is satisfiable, and the approximation algorithm must return an assignment satisfying at least a proportion $1 - 1/3K$ of the clauses. On the other hand, if $x \notin L$ then the approximation algorithm must return an assignment satisfying at most a proportion $1 - 1/2K$ of the clauses (simply because there are no assignments that do better than this). Note that by counting the number clauses satisfied by the output of the approximation algorithm we can easily determine whether $x \in L$. Thus we would have a polynomial-time algorithm for an NP-complete problem, and hence P = NP. □

What is the constant α in Theorem 4.6? To answer this question, we would need to know the constants implicit in the big-O expressions in the definition of $(\log n, 1)$-restricted verifier. By refining the basic techniques in ingenious ways, Håstad (1997), has shown that any $\alpha < 1/8$ will do, even for the restricted version MAX=3SAT: a major intellectual achievement. This is the correct answer, in the sense that there does exist a polynomial-time approximation algorithm for MAX=3SAT achieving performance ratio 7/8.

Theorem 4.6 can be viewed as a strengthening of Cook's Theorem. By analogy with Cook's Theorem we might hope to obtain further non-approximability results from Theorem 4.6 by exhibiting reductions from MAXSAT. But we have to exercise care: polynomial-time reductions do not in general preserve performance ratios. (Alternatively, polynomial-time reductions do not always preserve the gap that was created in Theorem 4.6 between the situations "all clauses satisfiable" and "only $1 - \alpha$ of the clauses satisfiable.") However, some reductions *do* preserve performance ratios. As an example, let's reconsider the SAT to CLIQUE reduction from CS3 *Computability and Intractability*, in the light of what we have recently learnt.

Let MAXCLIQUE be the optimisation problem whose input is an undirected graph G, whose feasible solutions are the cliques in G, and whose cost function is cardinality (i.e., the size of a clique in terms of the number of vertices).

Theorem 4.7. *There is a constant $\alpha > 0$ such that there is no polynomial-time approximation algorithm for* MAXCLIQUE *achieving performance ratio $1 - \alpha$, unless* P = NP.

Proof. We re-use the the reduction from CS3 *Computability and Intractability.* Let φ be a formula in CNF. We shall construct a graph G such that the size of the largest clique in G is equal to the maximum number of clauses in φ that can be satisfied simultaneously.

Let $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_r$, where each clause C_i is given by $C_i = (\lambda_{i1} \vee \lambda_{i2} \vee \cdots \vee \lambda_{i,s_i})$, and each λ_{ij} is a literal. Say that a pair of literals is *complementary* if it consists of the negated and un-negated forms of the *same* variable. (E.g., x and $\neg x$ form a complementary pair.) Our reduction maps the formula φ to the graph $G = (V, E)$ with vertex set

$$V = \big\{\lambda_{ij} : 1 \leq i \leq r \text{ and } 1 \leq j \leq s_i\big\}$$

and edge set

$$E = \big\{\{\lambda_{ij}, \lambda_{hk}\} : i \neq h, \text{ and the pair } \lambda_{ij},\, \lambda_{hk} \text{ is } \textit{not} \text{ complementary}\big\}.$$

The use of λ_{ij} to denote both a literal in φ and a vertex of G is a slight abuse of notation, but it does have the virtue of pointing up an intentional correspondence between vertices of G and literals in φ. Informally, two vertices (literals) are connected by an edge iff they occur in different clauses in φ and are not complementary. This completes the description of the reduction, which is clearly polynomial-time computable.

We now show, using the above reduction, that the existence of polynomial-time approximation algorithm for MAXCLIQUE, achieving performance ratio $1-\alpha$, entails the existence of a like approximation algorithm (with the same performance ratio) for MAXSAT. The theorem then follows from Theorem 4.6 (with the same α in both cases).

Suppose $\mathrm{OPT}(\varphi) = \hat{r}$, i.e., there is an assignment of truth values to the variables of φ that makes $\hat{r}$ clauses in φ true, and there is no assignment making $\hat{r}+1$ clauses true. Consider any assignment that makes $\hat{r}$ clauses true, and select one true literal from each true clause ($\hat{r}$ literals in all). The $\hat{r}$ vertices of G that correspond to this choice of literals must form a $\hat{r}$-clique. (Since all $\hat{r}$ literals are true, no pair of them can be complementary.)

If we run the postulated approximation algorithm on G we shall obtain a clique of size at least $\tilde{r} = \lceil (1-\alpha)\hat{r} \rceil$. Consider the literals corresponding to the vertices of this clique: Each of the $\tilde{r}$ literals must occur in a different clause of φ; moreover, no pair of the $\tilde{r}$ literals is complementary. Therefore, we can construct an assignment of truth values to the variables of φ which makes all $\tilde{r}$ literals true, and hence at least $\tilde{r}$ clauses of the formula φ true. But $\tilde{r}/\hat{r} \geq 1-\alpha$, so we have an approximation algorithm for MAXSAT with performance ratio $1-\alpha$. $\square$

Although Theorem 4.6, as we remarked, comes close to characterising the difficulty of approximating MAXSAT, Theorem 4.7 can be hugely strengthened. Indeed Håstad (1996) has shown (under some reasonable complexity-theoretic assumption) that for any constant $\varepsilon > 0$, however small, there is no polynomial-time approximation algorithm for MAXCLIQUE with performance ratio $n^{-1+\varepsilon}$, a devastating negative result.