TASK QUARTERLY 9 No 3, 273–291

LOGIC, PRIMES AND COMPUTATION: A TALE OF UNREST

STEFANO LEONESI AND CARLO TOFFALORI

Dipartimento di Matematica e Informatica,
Università di Camerino,
via Madonna delle Carceri 9, 62032 Camerino, Italy
{*stefano.leonesi, carlo.toffalori*}*@unicam.it*

(Received 25 May 2005)

Abstract: The early connections between Mathematical Logic and Computer Science date back to the thirties and to the birth itself of modern Theoretical Computer Science, and concern computability. This survey wishes to emphasize how alive and fruitful this relationship has been since then, and still is.

Keywords: computability, feasibility, primality, satisfiability

1. Logic and Computation

It is undeniable that Mathematical Logic and Computer Science closely interacted in the past and are still closely interacting. Indeed it is generally agreed that the birth itself of the modern Theoretical Computer Science directly springs from Mathematical Logic. Several contributions to [1], most notably [2] and [3] discuss and investigate this topic in full detail; in particular, Beeson's paper [2] explicitly, and perhaps provocatively claims that *Logic is Computation* and, conversely, *Computation is Logic*. Actually we have to say that we do not feel so absolute. Anyway nobody can deny that there do exist strong historical roots supporting this point of view and in any case the relationship between Mathematical Logic and Computer Science. Perhaps it is worth summarizing them once again. In some sense the affair began in the early thirties, when several logicians, and among them Gödel, Kleene, Church, Turing, were interested in singling out a precise abstract mathematical definition of what a computation formally is, and consequently of what is computable and who is entitled to compute. Some deep motivations originated this scientific curiosity. In fact in 1931 the celebrated Gödel Incompleteness Theorems had underlined the human impossibility of proving deductively all the statements true in natural numbers, and had emphasized in this way the problem of what Man, or Machine, can really compute and solve. Besides that, the logical and the mathematical communities were just facing at that time some problems whose solution looked very far from being obtained, and had been conjectured impossible. This is the case, for instance, of the

famous Hilbert Tenth Problem, but also of a crucial logical question, again proposed by Hilbert, called in German *Entscheidungsproblem* (so Decision Problem) and lying in finding a procedure recognizing the true statements in first order logic and distinguishing them from the false ones. The difficulties arising in answering these questions did produce a sort of pessimistic feeling, tending to exclude any positive solution and, after all, raising the general problem of realizing what can be effectively computed. So the basic abstract question became, as said: What is computable? And what does computing mean?

Several answers were proposed in those years and especially in 1936, by Gödel, Kleene and Church (via the notion of recursive function), Church (λ-calculus), Turing (Turing machines); all of them were proved equivalent to each other. But indeed Turing's model and Turing's machine were the most convincing ones, and still are the most popular ones. Accordingly, one can agree that intuitive computation just corresponds to Turing computation, meaning that:

what is computable is exactly what a Turing machine can compute.

This is the (rough) content of the famous Church-Turing Thesis. By the way, it is also worth emphasizing the fact that Turing machines provided a model *ante litteram* of modern computers, but preceded almost ten years the first electronic computer, Von Neumann's ENIAC.

In conclusion there is no doubt that Mathematical Logic did contribute to the birth of Computer Science, as the program itself of defining computations and computers was firstly debated and in some sense solved in Mathematical Logic, by logicians and in order to answer logical questions. Indeed, on the basis of Church-Turing thesis, the *Entscheidungsproblem* and, some decades later, in 1970, the Hilbert Tenth Problem itself at last received a surprising negative solution, saying that no effective procedure can handle them because no Turing machine can do it.

Problems which were proved to be algorithmically unsolvable in this way include several other notable examples, both from Logic and general Mathematics. We would like to recall here, for instance, the decision problem of the theory of the addition and multiplication of natural numbers $(\mathbf{N}, +, \cdot)$ – a result of Tarski closely related to Gödel's Incompleteness Theorems. Functions from $\mathbf{N}$ to $\mathbf{N}$ which cannot be algorithmically computed (because no Turing machine can do it) are also known; let us mention here as a nice and amusing example the Rado Σ function, growing so fast to overcome asymptotically any Turing computable function, such as $n \mapsto 2^n$, $n \mapsto 2^{2^n}$, $n \mapsto 2^{2^{2^{\cdots^n}}}$, and so on: [4] discusses in more detail the properties of Σ, which are also described and updated in the websites www.drb.insel.de/∼heiner/BB/index.html and grail.cba.csuohio.edu/∼somos/bb.html.

Key basic features of the Turing model include:

- discreteness,
- determinism.

In fact, Turing machines deal with discrete (indeed finite) inputs, like natural numbers or more generally *finite length* words on *finite* alphabets, and run through successive steps in a *discrete* time. Moreover, their computations may be infinitely long, so diverge on some particular inputs; but, when a computation halts, its output is

unique, indeed any step of any computation is uniquely determined by the instructions of the machine.

These peculiarities of discreteness and determinism sound quite reasonable. Even today computers are usually expected to deal in practice with *finite* strings on *finite* alphabet and to run in a *finite* and *discrete* time. Even from a theoretical point of view, this choice of referring to a discrete framework, so ultimately to natural numbers, may boast of some strong and prestige support, like the famous opinion of Kronecker according to which *natural numbers are the only created by God* and so there is no reason to imagine and involve real or complex numbers, or continuous settings.

But in the years after Turing, his model and Church-Turing Thesis had also to meet with some reservations and criticism. Indeed one can reasonably claim that Turing machine is a possible model of computation, but it is not the only one; in fact it cannot fit some alternative approaches to computation, such as:

(i) natural computation, aiming at studying and approximating the behaviour of a nervous system,
(ii) or nanocomputation, considering microscopical phenomena,
(iii) or also some aspects of the celebrated quantum computation.

In all these settings noise, uncertainty, errors, faults, damage may occur, which suggests new mathematical models and in particular:

- continuity,
- undeterminacy

instead of discreteness and determinism.

Accordingly new horizons have been arising in computation theory, privileging a *continuous* (rather than discrete) approach, so *real* (rather than integer) numbers, and looking for *analog* (instead of *digital*), or also *hybrid* computers. By the way Logic has been playing a crucial and critical role also in these alternative settings, accompanying and helping their developments; for instance, we might mention here *quantum logics* and their support to quantum mechanics, since the pioneer contribution of Birkhoff and Von Neumann in 1936 [5].

These different approaches to computation have been debated on their turn, and still are. Even today one may meet people claiming (just like Kronecker) that real numbers do not exist, hence considering continuous models or analog computers makes no sense in practice.

Anyway our aim in this paper is not to contribute to this discussion. We found it right to mention shortly these alternative ways to computation, but we wish to continue to refer to the Turing approach, so to Turing Machine and to Church-Turing Thesis, as the most popular model (although not the only possible one). This is the setting we will stay in, and where we want to outline and discuss some relevant developments this model has been having since the early sixties, and new interests and perspectives sprouting from its interior. Our purpose is also to emphasize that Mathematical Logic always intervenes as the core of all this and still interacts in a crucial way with the progress of Theoretical Computer Science, just as at the time of Turing.

We assume that our reader is neither familiar with Logic, nor with Computational Complexity Theory. So we will try to introduce our main topics without any hurry and to keep our exposition as clear and simple as possible. In this perspective we will frequently refer to primes. In fact, Number Theory and the fascinating world of primes provide a wonderful tool for our purposes, as a matter classical but still lively and rich of new and beautiful developments. A much wider treatment of Computational Complexity can be found, for instance, in classical textbooks such as [6, 7] or in the more recent [8]. [1] discusses in detail the relationship between Logic and Computability.

2. Computable or Feasible? Gauss' opinion

Let us start with a provocative question and ask:

Is "computability" the right notion?

As said, we are assuming the quite orthodox point of view of classical computability, the one inspiring the model of Turing and leading to the Church-Turing Thesis. So our question may sound quite strange and surprising.

Anyway, to illustrate what we mean, let us propose a simple example, actually a twofold example: the problems PRIMES and FACTORING among natural numbers. Accordingly we are given an integer $N \geq 2$ as an input, and we are asked:

1. (PRIMES) to decide whether N is prime or composite,
2. (FACTORING) to decompose N into its prime factors.

So we are considering a couple of problems having little to do at least in principle with Logic (don't worry, Mathematical Logic will arrive and play its role soon), and yet both classical and well-known to everybody in Mathematics, hence easy to understand.

Let us mention the opinion Gauss expressed in 1801, in Article 329 of his *Disquisitiones Arithmeticae*, about these questions:

> *The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in Mathematics.*

This opinion, although authoritative, may sound a little surprising. Indeed, the two problems have an easy and well known procedure dating back to the ancient Greeks and perhaps before, so more that two thousand years old. This algorithm proceeds as follows. Take all the integers $1 < d < N$ (actually $d \leq \sqrt{N}$ suffices) and try to divide N by d.

- If some division is successful, then declare N *composite*, indeed you have got also information about the factoring of N, through d and the quotient $\frac{N}{d}$.
- Otherwise, if no division gives an exact quotient with remainder 0, then output N *prime*.

As said, this algorithm is very simple and Gauss did know it very well. And yet let us read Gauss again:

> *The dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and celebrated.*

Why this urgency so many centuries after the Greeks? Let us hear once again Gauss' explanations:

> *The techniques that were previously known would require intolerable labour even for the most untiring calculator.*

In other words what Gauss was regretting is the lack of any *easily practicable* primality or factoring algorithm at his time. For instance, the elementary procedure of the Greeks may require up to $\sqrt{N}$ divisions, but $\sqrt{N}$ is an exponential function of the length of N (which is approximately its logarithm) and everyone agrees that:

> *exponentially many attempts require too much labour.*

Indeed what this example emphasizes is that (Turing) computable might not imply feasible and practicable.

In fact, there do exist some cases of problems having some algorithmic solution, but no feasible algorithmic solution. A famous example, this time coming from Mathematical Logic, is provided by a result of Fischer and Rabin [9] concerning the first order theory of the real addition $(\mathbf{R}, +)$. It is well-known that this theory has a decision procedure, indeed Tarski showed in the late thirties that the first order theory of the real addition and multiplication, that is of the real field $(\mathbf{R}, +, \cdot)$, is decidable. But the theorem of Fischer and Rabin says that no fast algorithm can handle it: there is a constant $c > 0$ such that, for every Turing machine deciding the first order real addition, there is a positive integer n_0 such that, for every integer $n \geq n_0$, there exists a sentence ϕ_n on the real addition such that:

- ϕ_n has length $\leq n$,
- the machine requires at least 2^{cn} steps before answering about the truth of ϕ_n in $(\mathbf{R}, +)$

(and an exponential time is to be supposed unfeasible, as said).

So we might ask:

Question 1. *Is "feasible" – rather than "algorithmically computable" – the right notion, i.e. the right way of intending "effectively computable"?*

But, of course, one should preliminarily clarify:

Question 2. *What does "feasible" mean?*

Indeed we might have some intuitive idea about that, but we should fix it in a precise way. Hence, let us first deal with Question 2. Of course, several quite natural criteria of interpreting and measuring feasibility come to mind:

- the *time* a computation requires,
- the *space/memory* it needs,
- its *cost* (money),
- the *energy* it requires

as well as further, subtler criteria, like *randomness* or *interactivity* (we will illustrate them in more detail later).

However the most natural option (but not the only one!) is *time*. Accordingly feasible might mean *fast* under this perspective.

A corresponding theory is easily sketched: For every Turing machine M, one considers the *complexity function of* M, taking any positive integer n to the maximal number of steps of convergent computations of M on inputs of length $\leq n$ (if any). Unless excluding machines diverging almost everywhere (and consequently having no practical interest), one can reasonably assume that these functions are defined for every sufficiently large n, increasing and unbounded.

The complexity functions are then compared *asymptotically* using the O (big o) preorder relation, defined as follows: for f and g complexity functions, $f = O(g)$ holds if and only if f asymptotically dominates g up to a constant positive factor c, meaning that $f(n) \leq c \cdot g(n)$ for every sufficiently large positive integer n.

By the way, let us introduce here a slight variation on this O notation, *i.e.* $\widetilde{O}$:

$$f = \widetilde{O}(g) \quad \Leftrightarrow \quad f = O(g \cdot (\log^{O(1)} g))$$

(where $O(1)$ denotes a positive constant); note that, if $f = \widetilde{O}(g)$, then f is asymptotically bounded in the O-relation by some power of g. Also, observe that, in this framework, it does not matter with respect to which basis > 1 logarithm is computed; for, any two of these logarithmic functions are clearly O each other. Accordingly 'log' will mean from now on logarithm with respect to any fixed basis $a > 1$ (for instance, with respect to $a = 2$).

So let us assume that time is our referring parameter in measuring feasibility. At this point we should agree what *feasible*, hence *fast* means in this framework. There is a proposal about that, arising since the sixties, basically due to Edmonds (1965 [10–12]), but also expressed in some preliminary form by Von Neumann (1953 [13]), Rabin (1963 [14]) and Cobham (1964 [15]) and confirmed by Cook and Karp in the early seventies. So let us call it *Edmonds-Cook-Karp Thesis* (a name directly reminding Church-Turing Thesis). As a (rough) slogan, it might be stated as follows:

fast = polynomial.

More precisely: the *fastly* solvable problems are exactly those that can be solved by a Turing machine with complexity $O(f(n))$ for some polynomial f (with integer coefficients and positive values). This class is usually denoted P (where P means polynomial, of course).

This proposal is still largely debated. Indeed there are some puzzling questions concerning it. For instance, everyone knows that polynomials may have arbitrarily large (and even titanic) degree and leading coefficient. So assume to deal with polynomials $f(n)$ of the form:

$$n^{2^{2^{2^{\cdots}}}}$$

or

$$2^{2^{2^{2^{\cdots}}}} \cdot n;$$

both are monomials, and the latter is even linear. But how fast an algorithm having this complexity and so, basically, this running time is?

On the other hand, theoretically speaking, can we propose anything better when considering the time criterion? For instance, for k a fixed positive integer, why to accept, say, a running time n^k and to exclude n^{k+1}? Or why to allow $k \cdot n$

and to forbid $(k+1)\cdot n$? Is there any natural evidence supporting these choices? It seems no. So, in some sense, the Induction Principle itself comes to the assistance of Edmonds-Cook-Karp Thesis.

In conclusion let us accept, at least momentarily, this thesis, just for laziness if not for belief.

3. Satisfying Gauss' expectations

The class P is not so crowded as one would like it to be. But it had a relevant *new entry* in the latest times. Indeed the problem PRIMES (that of distinguishing prime and composite numbers) was shown in P only in 2002. So the Gauss expectations and dreams were at last satisfied. Three indian researchers (Agrawal, Kayal, Saxena) did it. Their algorithm, named AKS after their initials, is founded on the following nice characterization of primes:

Theorem 1. (Agrawal-Kayal-Saxena, 2002, [16, 17]) *Let N be an integer ≥ 2. Let r be a positive integer $< N$ such that N has period $> (\log_2 N)^2$ modulo r. Then N is prime if and only if the following conditions hold:*

(a) *N is not a perfect power;*
(b) *N does not have any prime factor $\leq r$;*
(c) *for every positive integer $a \leq \sqrt{r}\cdot \log_2 N$ the polynomials $(x+a)^N$ and $x^N + a$ are congruent modulo $\langle N, x^r - 1\rangle$.*

As said, this is a beautiful characterization of primality. In fact, all the conditions (a), (b), (c) are easily understood; indeed (a) and (b) are trivial, and (c) directly refers to the classical *Fermat Little Theorem* (we will deal again with it in the next sections). The proof of the theorem is also simple, although ingenious; basically it requires some combinatorics, elementary group theory and finite field theory. Which is more relevant for our purposes is that, on its basis, one can eventually show:

Theorem 2. (Agrawal-Kayal-Saxena, 2002, [16, 17]) PRIMES *is in P.*

In fact Theorem 1 does suggest an algorithm checking primality deterministically in at most polynomial running time. Indeed procedures recognizing perfect powers, hence testing (a), even in (almost) linear time were known well before 2002 (see [18]). The conditions (b) and (c) clearly depend on r. But one sees that what they require can be checked in time $\widetilde{O}(\sqrt{r^3}\cdot \log^3 N)$. So the point is to find r as small as possible. As $r \geq (\log_2 N)^2$, the best running time we can expect is $r = \widetilde{O}(\log^6 N)$. Agrawal, Saxena and Kayal found in their proof $r = \widetilde{O}(\log^5 N)$, which ensures a total running time $\widetilde{O}(\log^{10.5} N)$. Subsequent further improvements gave $r = \widetilde{O}(\log^3 N)$ and then provided a lower running time $\widetilde{O}(\log^{7.5} N)$.

Quite recently Lenstra and Pomerance [19] at last reached the bound $\widetilde{O}(\log^6 N)$ via an approach still inspired by [16], but slightly different, and using some old Gauss ideas about the problem of building regular polygons.

Needless to say, and due the definition of $\widetilde{O}$, a running time bounded by some power of $\log N$ with respect to $\widetilde{O}$ is also polynomially bounded by $\log N$ in O.

4. A Millennium Problem and a Question of Logic

So PRIMES is in P. But what about FACTORING? With respect to this point, let us recall here the famous story of the mathematician Frank Cole. In a 1903 American Mathematical Society meeting in San Francisco, he proposed to the audience the following factorization:

$$2^{67}-1 = 147573952589676412927 = 193707721 \cdot 761838257287.$$

Of course one may ask why this was so interesting and relevant. The answer is that the number factored in this way is a Mersenne number, more precisely is the 67-th element in the list of Mersenne numbers, and everybody knows the crucial role these numbers have in searching new large primes. Indeed today an open source software called GIMPS (Great Internet Mersenne Prime Search) is available on the web just to find Mersenne primes or to factorize Mersenne composites; everyone wishing to discover new primes, or to factorize big Mersenne numbers might experience it. In particular factoring $2^{67}-1$ is fast using GIMPS, but it was not in 1903; indeed, Cole told that finding his decomposition required "three years of sundays" (as he was looking for his result for three years, and devoted his week ends to his attempts).

So Cole's factorization needed three years of labour and long efforts. Anyway, it can be written in a single line, and you can check it in a few lines provided that you know a right factor; in fact, what you have to do at that point is just a division.

What we aim at emphasizing with this example is that factoring may be fast to check when one knows the right decomposition. In fact, given your composite input N, you have simply to ask a witness d (a non-trivial divisor $\neq 1, N$ of N) and then to check that d divides N exactly (*i.e.* with reminder 0). After that, you are confirmed that $N = d \cdot \frac{N}{d}$ and you may repeat your procedure with respect d and $\frac{N}{d}$: if you know that they are composite, you may ask new witnesses to confirm it and to proceed in your decomposition.

This is what you are expected to do, just a sequence of divisions, and it is known that the cost of a single division – its running time – is at least quadratic with respect to the length of divisor and dividend; furthermore, it is easily seen that the maximal number of divisions you should compute to accomplish the factoring of N is $O(\log N)$, because the number of prime factors of N cannot exceed $\log_2 N$. Also, notice that the length of a proper divisor d of N is, of course, less or equal than the one of N.

So factoring is fast to check, as said. However what we are looking for is a fast algorithm to *factor* a given input N, and not simply to *check its factorization* when done. But then we must realize that at the present time no fast factoring algorithm is known. Several deep and sophisticated procedures have been proposed in the latest years in this direction, involving ellyptic curves, sieve methods, continued fractions and so on, but none of them works quickly (*i.e.*, within a polynomial time with respect to the length of N).

By the way, this gap between the worst expected running times in decomposing a number N into its prime factors and in recovering N as the product of these factors, so between factoring and multiplying, is the foundation of the celebrated RSA public key cryptographic system. Also, we are forgetting here Peter Shor's *quantum*

algorithm [20], fastly handling, at least in principle, both primality and factoring. In fact, as said, our perspective in these notes is the classical one and we are neglecting *new* models of computation.

But let us come back to the difficulties we have observed about factoring. We are led in this way to introduce another computational complexity class, called *NP* and showing the same features as the elementary factoring procedure sketched a few lines ago. In detail, a problem S is in *NP* when it admits a corresponding problem S' in P such that, for every input N, N is in S if and only if one can find some witness d such that the length of d is polynomially bounded by that of N and (N, d) satisfies S'. Note that this is just what happens when S is the problem of factoring (in which case S' is divisibility).

Of course $P \subseteq NP$ (as, for S in P, we can take $S' = S$ and d empty).

But there are other computational complexity classes arising quite naturally in this setting and accompanying NP. For instance, come back to factoring and to the procedure described before. Assume now that your input N is prime. Then no single tentative divisor d can witness it in the way we have sketched. For, after realizing that d cannot divide N, you have no definitive evidence that N is not composite, but you have to check *all* the witnesses d before excluding N composite. Of course, we know since Agrawal-Kayal-Saxena that PRIMES is in P. But what we wish to emphasize here is that we cannot realize it fastly via the elementary algorithm, although the same procedure can manage quickly the case when N is composite and confirm fastly its decomposition into prime factors. In fact, in the composite case, invoking a unique right witness is sufficient.

This leads, at least in principle, to a new class, called *coNP* and including all the problems whose *complement* is in *NP*. Basically a problem S is in *coNP* when there are a problem S' in P and a polynomial p_S such that, for every input N, N is in S if and only if all the witnesses d having length polynomially bounded by that of N via p_S unanimously declare that (N, d) satisfies S'.

The precise relationship between P and *NP* (and *coNP*, too) is still an open problem, and is reputed quite difficult. Basically, what has to be understood here is whether a problem whose solutions can be fastly checked is also fast to be solved (when fast means polynomial).

$P = NP$ is a fundamental question in Applied Mathematics and in Theoretical Computer Science. In 2000 it was inserted in a list of seven *Millennium Problems* where the Clay Institute collected what it reputed the most difficult and basic questions in Mathematics today. $P = NP$ is a 1 million dollars problem, as this is the prize the Institute fixed for the solver of any question in the list. So you might well consider to spend some sundays, and even "three years of sundays," to look for its answer.

But what is quite relevant for our purposes, and in general, too, is that $P = NP$ is a genuine question of Logic. In fact this is the content of a theorem of Cook and Levin in the early seventies. Let us explain why in detail.

Consider Boolean propositional logic. You are given *formulas*, built for instance as conjunctions of disjunctions of elementary propositional variables p_0, p_1,... and negations. You want to know whether there is any truth assignment of these variables

p_0, p_1,..., taking values 0 or 1 (meaning *false* and *true*, respectively), such that the whole formula is consequently satisfied (*i.e.*, declared true). This *Satisfiability Problem* is a logical question. It is generally called SAT (for satisfiability, of course). Also, several very elementary and simple procedures can handle it. For instance, just list all the truth assignments for the variables involved in your formula and for each of them deduce the truth value of the whole formula. But this procedure, and all the procedures known so far, have the same deficiency as the elementary primality and factoring algorithm of the ancient Greeks: they may sometimes require up to exponentially many steps, so too many steps. For instance, if your formula contains n propositional variables, you should be aware that the possible truth assignments of these variables are 2^n and you might be obliged to check 2^n cases before having your answer.

Hence what we expect here is a fastly running algorithm.

By the way, note that, if your formula is satisfiable and you know a right truth assignment of the variables making it true, then you may confirm satisfiability very fastly: Just take this right assignment (a single case among the 2^n possible ones) as a witness and compute the corresponding truth value 1 for your formula. This shows that SAT is in NP, just as factoring.

Anyway, there is no evidence yet that SAT is in P. But the question is even subtler. In fact SAT is an *NP-complete* problem, meaning that (roughly speaking) $P = NP$ holds if and only if SAT is in P. Of course, it is trivial that, if SAT is out of P, then NP includes P properly. But what is surprising and amazing (and forms the subject of the Cook-Levin theorem) is that the converse is also true. For, any problem in NP can be translated by a suitable fast procedure to SAT, hence a quick algorithm solving SAT might apply in this way to any problem in NP.

Consequently a fast positive solution of the logical question SAT would imply $P = NP$ at all. Then answering SAT quickly is a millennium problem and a 1 million dollars question.

NP-complete problems include now a plenty of examples in Algebra, Graph Theory, Combinatorics, Chemistry, Biology, and so on (see [21, 22] or explore the website www.csc.liv.ac.uk/∼ped/teachadmin/COMP202/annotated.np-html). But SAT was the first, and is the father of all of them. In fact, showing that a given S in NP is NP-complete reduces to see that SAT can be deterministically and fastly translated to S; indeed this is the main road to single out new NP-complete examples S. By the way, one may wonder if factoring is one of them. This is still an open question. Of course, should P coincide with NP, the NP-completeness notion would trivialize and so our curiosity about factoring would make no sense. Also, it might be the case that $P \neq NP$ but FACTORING is in P. But there is a general feeling and some evidence that, assuming $P \neq NP$, FACTORING should lie in $NP - P$ but should not be NP-complete. By the way, these problems (the ones that belong to $NP - P$ but are not NP-complete) are called *NP-intermediate*.

5. Witnesses again

P, NP and the time criterion does not exhaust the search of new computational horizons about *feasibility*. Indeed, as said, P – the class of good computational

problems according to the Edmonds-Cook-Karp Thesis – is not so crowded and rich of examples as one would expect, although a lot of quite interesting problems, including in particular the *NP*-complete ones, are just on its threshold. So new proposals and new perspectives can be reasonable, and even welcome. For instance, *randomness* might be involved towards a right definition of feasibility. To illustrate what we mean here, let us refer again to PRIMES and introduce a primality procedure dating back to the late seventies. This is the *Miller-Rabin Probabilistic Algorithm* [23, 24].

Its ingredients are quite simple. Basically they reduce to two elementary facts:

1. Fermat's Little Theorem, already met before and saying that, if N is a prime, then, for every a coprime to N, the congruence $a^{N-1} \equiv 1 (\mathrm{mod}\, N)$ holds;
2. the easy proposition that, for a prime N, the only square roots of 1 modulo N are ± 1.

On these grounds, the Miller-Rabin Test works as follows.

Take your input $N \geq 2$. Can assume that N is odd (as 2 is the only even prime, indeed "the oddest prime" according to a celebrated joke of Zassenhaus). Accordingly $N-1$ is even and can be represented as $2^s \cdot t$ with $s > 0$ and t odd. At this point appeal to an integer witness a with $1 < a < N$.

- If a and N are not coprime, then you are done, as you have even got a divisor $\neq 1, N$ of N (the greatest common divisor of a and N); accordingly answer N COMPOSITE.
- Otherwise, if a and N are coprime, then check $a^{N-1} \equiv 1 (\mathrm{mod}\, N)$; if this congruence fails, declare N COMPOSITE again.
- Assume now a and N coprime and $a^{N-1} \equiv 1 (\mathrm{mod}\, N)$. This means:

$$(a^{2^{s-1} \cdot t})^2 \equiv a^{2^s \cdot t} \equiv a^{N-1} \equiv 1 (\mathrm{mod}\, N).$$

Now compute $a^{2^{s-1} \cdot t}$ modulo N.

(a) If what you get is $\not\equiv \pm 1 (\mathrm{mod}\, N)$, then use Fact 2 before and say N COMPOSITE.
(b) If $\equiv -1 (\mathrm{mod}\, N)$, then venture N PRIME.
(c) Similarly, if what you get is $1 (\mathrm{mod}\, N)$ and $s = 1$, then venture N PRIME.
(d) Finally, if you get $1 (\mathrm{mod}\, N)$ but $s > 1$, then look at $a^{2^{s-1} \cdot t}$ modulo N and repeat the previous procedure.

One sees that the total running time of these operations is $O(\log^5 N)$, which can make a real difference and a real improvement even with respect to AKS. Indeed the Miller-Rabin computations are very quick to be done even in practice.

However N COMPOSITE is a secure answer, but N PRIME is not. In fact, when the algorithm declares N PRIME, it is not because it has a proof that N is really prime, but because there is no evidence contradicting this conclusion, and nothing against it arose during the procedure.

But Rabin showed that the probability error of this algorithm after 1 attempt – so after appealing to a single witness a – is at most $\frac{1}{4}$, which implies that the same probability error after invoking k different witnesses a decreases to $\leq \frac{1}{4^k}$.

Now let us quote an authoritative opinion of E. Borel, according to which:

an event having probability 10^{-50} *will never happen and, even if it happens, it will never be observed.*

Note that $k = 100$ attempts suffice to reduce the Miller-Rabin algorithm probability error well below the Borel level 10^{-50}. Also, 100 different tests do not affect (both in theory and also in practice) the total running time $O(\log^5 N)$. In conclusion the Miller-Rabin procedure is:

- not completely secure,
- but highly reliable,
- and, what is also relevant, really fast!

Moreover, nowadays it may be combined with a new *Berrizbeitia-Bernstein* algorithm, 2005 [17, 18]:

- again inspired by AKS,
- having running time $\widetilde{O}(\log^4 N)$,
- absolutely reliable when it answers N PRIME, sometimes fallacious when it declares N COMPOSITE.

So the Berrizbeitia-Bernstein procedure is in some sense complementary to the one of Miller-Rabin. Hence let them work in parallel on a given N and wait for their quick answer. This combination is:

- almost infallible (indeed you would be very unlucky if the former algorithm should output that your N is COMPOSITE and the latter should say that it is not!),
- very highly reliable,
- much faster than AKS itself.

But let us come back to the purposes of this paper. What this example shows it that new perspectives can be experienced and opened when searching a reasonable notion of feasibility. For instance, why not to replace secure but slow answers by sometimes fallacious but highly reliable and fast computations? Accordingly consider the class of the problems which can be answered in at most polynomial time with a probability error $\leq 10^{-50}$. This class is usually called *BPP* (meaning Bounded Probabilistic Polynomial). *BPP* might be a strong candidate for the class of *feasible* problems. Trivially *BPP* includes *P* (where the answers are absolutely reliable), but it is not clear whether $BPP = P$ or not. Also, the relationship between *BPP* and *NP* is still to be clarified. Indeed there is a famous theorem of Sipser, Gacs and Lautemann [25, 26] saying that these classes (*BPP* and *NP*) are not so far from each other, anyway it is not currently known whether *BPP* includes *NP*, or conversely *NP* includes *BPP*.

Other computational classes based on this probabilistic approach were also introduced. *RP* (meaning Random Polynomial) collects the problems having a fast solution procedure which is sufficiently reliable, and indeed completely right in at least one possible answer but may be wrong in the other (as the Miller-Rabin algorithm does for PRIMES when checking *compositeness*: in fact, its answer N COMPOSITE is sure, the opposite conclusion N PRIME is not). *ZPP* (Zero Error Probabilistic Polynomial) is the class of problems having a fast solution procedure which is completely right in all the answers it gives, but may sometimes be silent. Of course, in this case, it is the silence probability that must be computed and minimized.

Even in this probabilistic framework Logic is often just round the corner. For instance, there is a theorem of Valiant and Vazirani [27] dealing with the relationship between the classes RP and NP and indeed stating their equality $RP = NP$ provided that a logical problem has a positive answer. This key question is called $USAT$ (meaning *Unique Satisfiability*) and specializes the SAT problem, aiming at singling out those propositional formulas having a *unique* truth assignment. The Valiant-Vazirani Theorem says that, if $USAT$ is in RP, then RP and NP coincide.

6. Interactions

Other criteria suggest further computational classes, and so additional candidatures for the right class of feasible problems. For instance, one might refer to the *memory* resources an algorithm requires to develop its computations, so basically to the space these computations need. In this perspective one might form the class of problems admitting a solution procedure excluding any error and requiring a polynomial space to be accomplished. This is called $PSPACE$. Clearly it includes P, as a Turing machine computation consisting of n steps cannot involve more than n memory bits.

By the way, the primality elementary procedure of the ancient Greeks suffices to show that PRIMES is in this class $PSPACE$. In fact, given an integer $N \geq 2$, take any tentative d and divide N by d; if this operation is successful, then stop and declare N COMPOSITE; otherwise erase the space you have used in this division and employ it again for another d'. The memory necessary to write any single d (and N) and to test their division is polynomially bounded by the length of N. More generally, this argument shows that $NP \subseteq PSPACE$.

But there is a much more crucial example we would like to introduce here: it comes from Mathematical Logic and it is called $QSAT$. It deals with *quantified propositional formulas*, involving not only variables p_0, p_1,... and connectives such as *and*, *or* and *not*, but also quantifiers $\forall$ and $\exists$. Indeed each variable occurring in a formula is subjected to some quantifiers. We get in this way statements like:

$$\forall p_0 \exists p_1 \text{ such that } p_0 \text{ or not } p_1,$$

or

$$\forall p_0 \exists p_1 \text{ such that } p_0 \text{ and } p_1,$$

and so on. $QSAT$ looks for an algorithm deciding whether any such formula is *true* or not. The capital letter Q refers to quantifiers and SAT to satisfiability. Truth is defined here in the obvious way: for instance, the former formula listed above is true (because, whatever is your opinion about p_0 you can find a truth assignment of p_1 for which the corresponding truth value of "p_0 or not p_1" is 1), while the latter is not (as, when p_0 is reputed false, then no truth assignment of p_1 can satisfy both p_0 and p_1). It is a fact, although not immediate, that $QSAT$ lies in $PSPACE$. Indeed, $QSAT$ is the core of $PSPACE$, in fact it is *PSPACE-complete*, meaning that any problem in $PSPACE$ can be deterministically reduced in at most polynomial time to $QSAT$: this is the content of a theorem of Stockmeyer and Meyer in 1973 [28].

Another relevant class in this framework, and another possible candidate as the class of *feasible* problem with respect to the memory criterion is $LOGSPACE$,

assembling the problems with a solution algorithm requiring logarithmic, so less than polynomial, space; of course, we refer here to the memory necessary to develop and accomplish the computation and we neglect the space used to write the input (as the latter is obviously linear, and so more than logarithmic, with respect to the length of the input). It is easily seen that $LOGSPACE$ is contained in P, and one conjectures that this inclusion is proper, so $LOGSPACE$ and P do not coincide.

Another source of new computational classes springs from *interactivity*. Indeed the models of computation we have described so far exclude any relationship between who puts a question and who tries to answer, *i.e.* between Man and the Machine. Man proposes an input and then waits for Machine's output (if any). But we might also admit that Man intervenes during a computation, reads its partial results and accordingly makes his question more precise.

To fix abstractly this setting, we might introduce two characters (as Babai and Moran did in their paper [29]):

- $\mathcal{M}$ = the *prover* = Merlin the Wizard (who knows everything),
- $\mathcal{A}$ = the *verifier* = the young Arthur (who knows nothing and has to learn everything).

Here is a rough description of the way our interactive algorithm proceeds. Suppose one has to compute something. Basically Merlin should convince Arthur of the solution, and Arthur should be convinced only beyond any reasonable doubt (say up to a 10^{-50} probability error). In fact Arthur does not know for certain that Merlin is not lying and so has to check very carefully what Merlin asserts. As said, at the beginning $\mathcal{A}$ has no information about the answer of the problem, so the only way he has to proceed is to press Merlin, to check and to contradict his statements *randomly*, just casting the dice. On this ground:

- $\mathcal{A}$ puts questions,
- $\mathcal{M}$ provides answers.

The number of messages between $\mathcal{A}$ and $\mathcal{M}$, as well as the length of their contents, must be polynomially bounded by the length on the input. At the end of this correspondence $\mathcal{A}$ gathers the information he got and checks deterministically the answer again in polynomial time. The final result clearly depends on the random contributions of $\mathcal{A}$. Accordingly it is required that $\mathcal{M}$ has high probability of convincing $\mathcal{A}$ of the right answer and low probability of cheating him.

Several classes are built in this way. NP itself corresponds to this scheme, in the particular case when the interaction between $\mathcal{M}$ and $\mathcal{A}$ reduces to a single message from $\mathcal{M}$, revealing the solution or a crucial information, so that a final fast computation of $\mathcal{A}$ is sufficient to conclude without any random step. Even BPP can be recovered in this framework, when it is $\mathcal{A}$ who casts the dice and, on this ground, accomplishes his computation without any help from Merlin. Let us also mention, for instance:

- MA, where only two interactions are allowed: firstly a message is sent from $\mathcal{M}$, after that $\mathcal{A}$ shares randomly and just casts the dice on the basis of what $\mathcal{M}$ said;

- or AM, again consisting of two interactions, but with inverted roles (so now $\mathcal{M}$ answers a random question of $\mathcal{A}$).

The reason why MA and AM are named in this way is transparent. Another class, more or less corresponding to the general description given above (so admitting polynomially bounded messages and lengths) is IP (meaning Interactive Polynomial). It was introduced by Goldwasser, Micali and Rackoff in 1985 [30]. It is easily seen that:

$$NP \subseteq MA \subseteq AM \subseteq IP.$$

But there is a very deep theorem of Shamir [31] connecting IP and the class $PSPACE$ introduced before and showing:

Theorem 3. (Shamir, 1992) $IP = PSPACE$.

Actually the inclusion $IP \subseteq PSPACE$ is not difficult to prove, as (even interactive) procedures running in a polynomially bounded time cannot exceed polynomially bounded resources of memory. So the critical point is to show the inverse inclusion $PSPACE \subseteq IP$. But here we may refer to the $PSPACE$-complete $QSAT$ as a common and legal representative of all the problems in $PSPACE$, and consequently limit ourselves to check that $QSAT$ lies in IP: again, a question of Mathematical Logic arises as the hearth of the matter and plays the key role in the proof. Indeed what Shamir's Theorem provides is just an interactive procedure testing $QSAT$ in at most polynomial time, as required by the definition of IP.

7. A tale of unrest

In conclusion, the candidacies for a right notion of feasibility (with respect to time, space, randomness, interactivity or anything else) are quite numerous. And indeed today a website www.complexityzoo.com, maintained by Scott Aaronson, takes care of a census of these computational classes. Up to June 2005, the complexity zoo collects 425 species, although the real number of its members might be quite different, perhaps lower, perhaps bigger. Indeed, should P and NP coincide, the level 425 would drastically decrease. On the other hand, some species in the zoo include and hide a lot of subspecies, so it might also happen that complexity classes are much more than 425, and even infinitely many. The site also describes the connections between all these classes.

But what remains behind this plethora of examples? What can we say about the basic question of singling out the right notion of feasibility?

1. With respect to the latter question, we have to admit that an ultimate and generally agreed answer seems to be far from being reached. Indeed, as said in our introduction, it is the notion itself of computability that is still and largely debated nowadays. It is possible that new perspectives (for instance, the quantum way to computation) may suggest new convincing candidates. Indeed they have already proposed new candidates, which are not considered in these pages (just because we chose to limit our analysis to the Turing scenery). However what we have said so far just prefigures a sort of *tale of unrest* (to quote Joseph Conrad), witnessing how fretful and far from a solution is our question.

2. Anyway this research is also disclosing new connections and reductions and relevant analogies between the classes obtained so far. We have mentioned in the previous section Shamir's Theorem $IP = PSPACE$. But we might quote here also the celebrated Baker-Gill-Solovay Theorem confirming how difficult the problem $P = NP$ is and how involving *oracles* may radically change its answer. Before stating the theorem, let us briefly recall what *oracle* means in this setting. Everyone knows that, in the world of ancient Greeks, citizens having to face too obscure mysteries might consider to consult oracles, and hence ask for gods' help and advice through some sybil or prophetess. Of course, modern Computer Science does not need sybils and prophets, but may still admit oracles, just meaning problems $\mathcal{O}$. So, when testing an instance of some problem S, one may consider to spend a step in the computation to ask $\mathcal{O}$, to propose a single input to it and to wait for its output. $\mathcal{O}$'s assistance may be invoked as many times as you like, and the cost of any single intervention of $\mathcal{O}$ is just one step of computation, as said. New computational classes arise in this way. For instance, for a given oracle $\mathcal{O}$, $P^{\mathcal{O}}$ collects those problems that can be deterministically answered within a polynomial time with the help of $\mathcal{O}$. $NP^{\mathcal{O}}$ is introduced in a similar way. That being said, let us recall the statement of the theorem of Baker, Gill and Solovay.

   Theorem 4. (Baker-Gill-Solovay, 1975, [32]) *There are two oracles $\mathcal{O}$ and $\mathcal{O}'$ such that:*

   (i) $P^{\mathcal{O}} = NP^{\mathcal{O}}$,
   (ii) $P^{\mathcal{O}'} \neq NP^{\mathcal{O}'}$.

   Mathematical Logic again supports the proof. In fact, showing (ii), although combinatorially intricate, basically requires a classical *Cantor diagonalization* procedure. But, what is even more relevant, (i) refers to the *PSPACE-complete* logical problem $QSAT$ as $\mathcal{O}$.

This also introduces and partially answers the final question we would like to treat. In fact, one might wonder whether this restless searching of new classes and frontiers in Theoretical Computer Science is Logic yet, or has anything to share with Logic. But we have seen all throughout this paper how often Logic appears round the corner, and even along the road, not only in the basic original question (about what does computable, or feasible, mean), but also in the theoretical developments it produced. Indeed $P = NP$ itself is a question of Logic, and logical problems, like SAT, $QSAT$ and further ones, often arise as the core of the theory and the hearth of the affair. For instance, the Baker-Gill-Solovay theorem, or that of Shamir, do witness this authoritatively.

Also, some relevant members of the complexity zoo are directly inspired by Logic. In some sense, this is also the case of NP and $coNP$. In fact, the difference itself between NP and $coNP$ depends on a question of very basic and elementary Logic, as in the former case, that of NP, we expect *some* witness, while in the latter, the one of $coNP$, we have to hear *every* possible witness. So the distinguishing line between NP and $coNP$ formally relies on the logical nature (existential, or universal) of the

defining conditions. But at this point new classes might come to mind, corresponding to more complicated formulas, such as those requiring that:

- *for every witness d, there is a witness d'...*,
- *for every witness d, there is a witness d' such that, for every witness d'',...*

and so on. Indeed one can build in this way infinitely many complexity classes and altogether form the so called *polynomial hierarchy*. P, NP and $coNP$ are the bottom levels of this hierarchy, while its top, *i.e.* the union of all the classes obtained in this way, is usually denoted PH – just meaning *Polynomial Hierarchy*. Of course, the hierarchy might collapse and so these new classes might coincide with each other. For instance nothing exclude that $P = PH$ (even if this would imply $P = NP$, because $P \subseteq NP \subseteq PH$). On the other hand, there is no evidence supporting this conjecture, and it may also happen that the classes in the hierarchy are pairwise distinct (in which case the complexity zoo would include infinitely many different members).

An alternative source of computational classes springs from propositional logic and some related algebraic facts. In fact computational problems are often of the following form: You are given an input – like a natural number N, or a formula α – and you have to decide whether this input satisfies or not a certain property – like primality, or satisfiability – and accordingly to output "Yes" or "No". The bits 1 and 0 usually represent these opposite answers; moreover the input itself can be encoded as a finite string of 0 and 1. In this sense a problem S is naturally accompanied by a countable sequence of functions f_n^S from $\{0,1\}^n$ to $\{0,1\}$ where n ranges over positive integers; for every n, f_n^S takes any string of length n in $\{0,1\}$ to the corresponding output in S, 1 or 0, "Yes" or "No". On the other hand there is a classical result dealing with functions f from $\{0,1\}^n$ to $\{0,1\}$ $(n > 0)$ and saying that each of them can be obtained, although not uniquely, as the composition of three functions NOT, AND, OR sending respectively:

- 0 to 1 and 1 to 0,
- $(1,1)$ to 1 and $(0,1)$, $(1,0)$, $(0,0)$ to 0,
- $(1,1)$, $(0,1)$, $(1,0)$ to 1 and $(0,0)$ to 0.

The proof use basic combinatorial arguments, and the theorem has a quite relevant logical interpretation, as it can be equivalently stated by saying that the three connectives *not*, *and*, *or* are sufficient to develop the whole propositional logic. But let us come back to our functions f. One might use the theorem above and choose to measure how "complicated" f is by looking at its decompositions via NOT, AND, OR and counting, for instance, the minimal number of NOT, AND, OR which are necessary to decompose f. One is led in this way to build quite naturally a new class, collecting the problems S such that in the corresponding sequence of functions f_n^S $(n > 0)$ the "complexity" of f_n^S is polynomially bounded with respect to n: So Logic does cooperate in generating this new member of the zoo. The class we get in this way is shown to include P, while it is not clear if it extends also NP. However it exhibits a remarkable anomaly, as it contains even some problems which are not algorithmically computable in the sense of Turing. In fact, take any set K of integers whose membership cannot be decided by a Turing machine, and consider the following problem S: Given a string of 0 and 1, output 1 if its length belongs to K and 0

otherwise. So, for every n, the function f_n^S corresponding to S is constant and can be decomposed by using NOT, AND, OR at most twice. Anyway S cannot be decided by a Turing machine, as K is not. To get over this trouble one may require that the process generating f_n^S from n for every n be carried out by a Turing machine, running of course in at most polynomial time. But the class one obtains under this additional condition is easily seen to coincide with P.

In conclusion, Logic does still have something relevant to do and to say in the computational framework. To support this claim further on, let us mention, as a final topic of these notes, *Descriptive Complexity*. This is a branch of both Mathematical Logic and Complexity Theory, and aims at characterizing complexity classes by the type of Logic needed to express their problems. It was Richard Fagin who opened this research area in the seventies, when he found a surprising and notable characterization of NP [33]; in fact Fagin showed that a problem is in NP if and only if it is described by a formula in existential second order logic (so allowing quantifiers over both individual and relation variables, but using only existential quantifiers). After that, a lot of similar results were obtained, mostly by Neil Immermann (see [34] or visit the website ww2.cs.umass.edu/∼immerman/descriptive_complexity.html). For instance, it turns out that:

- the class of problems which can be described by arbitrary formulas in second order logic is PH (the top of the polynomial hierarchy),
- $coNP$ corresponds – of course – to universal second order logic,
- $PSPACE$ to second order logic with a transitive closure

and so on. Even the class of problems which can be described by formulas in first order logic (the one forbidding quantifiers over relation variables) has its own characterization in terms of concurrent random access machines.

The sense of these results is clear. In fact, as we already pointed out, the broad expanse of complexity zoo may sound excessive, and many of the classes it includes may look quite artificial. But, once a class is given a nice logical description (as in the examples before), everyone has to agree how natural the class is. So, even in this direction, Logic can support and help Computability.

References

[1] Herken R (Ed.) 1994 *The Universal Turing Machine: A Half-Century Survey*, Springer
[2] Beeson M 1994 *Computerizing Mathematics: Logic and Computation*, in: [1], pp. 172–205
[3] Davis M 1994 *Mathematical Logic and the Origin of Modern Computing*, in: [1], pp. 135–158
[4] Brady A 1994 *The Busy Beaver Game and the Meaning of Life*, in: [1], pp. 237–254
[5] Birkhoff G and Von Neumann J 1936 *Ann. Math.* **37** 823
[6] Papadimitriou C H 1994 *Computational Complexity*, Addison-Wesley
[7] Sipser M 1996 *Introduction to the Theory of Computation*, Course Technology
[8] Hemandspaandra L A and Ogihara M 2002 *The Complexity Theory Companion*, Springer
[9] Fischer M and Rabin M 1974 *Complexity of Computation, SIAM-AMS Proc.* (Karp R, Ed.) **7** 27
[10] Edmonds J 1965 *Can. J. Math.* **17** 449
[11] Edmonds J 1967 *J. Res. Nat. Bur. Standards* **71B** 233
[12] Edmonds J 1967 *J. Res. Nat. Bur. Standards* **71B** 241
[13] Von Neumann J 1953 *Contributions to the Theory of Games II* (Kahn H W and Tucker A W, Eds.), Princeton University Press
[14] Rabin M 1963 *Israel J. Math.* **1** 203

[15] Cobham A 1964 *Proc. Int. Congress for Logic, Methodoly and Philosophhy of Science* (Bar-Hillel Y, Ed.), Jerusalem, Israel, pp. 24–30
[16] Agrawal M, Kayal N and Saxena N 2004 *Ann. Math.* **160** 781
[17] Granville A 2005 *Bull. Amer. Math. Soc.* **42** 3
[18] Bernstein D 2004 http://cr.yp.to/primetests/quartic-20040213.pdf (to appear)
[19] Lenstra H W jr and Pomerance C 2005 http://www.math.dartmouth.edu/∼carlp/PDF/complexity12.pdf (to appear)
[20] Shor P 1994 *Proc. 35th Ann. IEEE Symp. on Foundations of Comp. Sci.*, Santa Fe, NM, USA, pp. 20–22
[21] Garey M and Johnson D 1979 *Computers and Intractability: A Guide to the Theory of Completeness*, Freeman
[22] Johnson D 1990 *Handbook of Theoretical Computer Science – A: Algorithms and Complexity* (Van Leeuwen J, Ed.), Elsevier, pp. 67–161
[23] Miller G 1976 *J. Comput. System Sci.* **13** 300
[24] Rabin M 1980 *J. Number Theory* **12** 128
[25] Lautemann C 1983 *Information Processing Letters* **17** 215
[26] Sipser M 1983 *Proc. 15th ACM Symp. Theory of Computing*, Boston, MA, USA, pp. 330–335
[27] Valiant L and Vazirani V 1986 *Theor. Comp. Sci.* **47** 85
[28] Stockmeyer L 1973 *Proc. 5th Ann. ACM Symp. on Theory of Computing*, Austin, TX, USA, pp. 1–9
[29] Babai L and Moran S 1988 *J. Comput. System Sci.* **36** 254
[30] Goldwasser S, Micali S and Rackoff C 1985 *Proc. 17th ACM Symp. on Theory of Computing*, Providence, RI, USA, pp. 291–304
[31] Shamir A 1992 *J. Ass. Comput. Mach.* **39** 869
[32] Baker T, Gill J and Solovay R 1975 *SIAM J. Comput.* **4** 431
[33] Fagin R 1974 *Complexity of Computation, SIAM-AMS Proc.* (Karp R, Ed.) **7** 43
[34] Immerman N 1983 *Proc. 15th ACM Symp. on Theory of Computing*, Boston, MA, USA, pp. 347–354