The Journal of Symbolic Logic
Volume 59, Number 1, March 1994

ACCESSIBLE TELEPHONE DIRECTORIES

JOHN B. GOODE

Les Américains a découragé
Les Cadiens de parler français
Ils voulient on parle juste en anglais
Les Cadiens va les embêter.
Bruce Daigrepont

Abstract. We reduce to a standard circuit-size complexity problem a relativisation of the P=NP question that we believe to be connected with the same question in the model for computation over the reals defined by L. Blum, M. Shub, and S. Smale. On this occasion, we set the foundations of a general theory for computation over an arbitrary structure, extending what these three authors did in the case of rings.

Introduction. There is no domain where the inability of mathematicians to produce results of some practical value (except, of course, to collect funds and obtain positions) is so pathetic as it is in this new field of investigation which is known as Theoretical Computer Science. And, indeed, the only conclusion of that sort that one may draw from this paper is that telephone companies, academicians, and other compilers of dictionaries do the right thing when they list their entries in alphabetical order: to adopt another system of classification, for instance, randomness, will not increase ease of access for the users.

We first consider a variety of directories which have the shape of an infinite binary tree T with nodes labeled by 0 or 1; such a T can be decomposed as $T_g^\wedge\varepsilon^\wedge T_d$, where T_g is its left subtree, T_d its right subtree, and ε is the label of its root, taking value 0 or 1. By a *digital word* we mean a finite sequence of 0's and 1's; the trees T are in bijection with the sets E of digital words via the following correspondence: while reading the word $\underline{\varepsilon} = (\varepsilon_1, \dots, \varepsilon_n)$, we climb in the tree, going to the left when we meet a nul ε_1 and to the right when it is a 1; when we finish, if we find a label in the tree which is 1, we put $\underline{\varepsilon}$ in E, and if it is 0, we exclude the word. It is clear that, if we consider the tree T as a directory for the corresponding set E, then the consultation of this directory is done optimally, in linear time; one may doubt the utility of a directory which gives only a list of customers of the company, without their phone numbers, but we consider this as a point of detail. In spite of appearances, your familiar encyclopedia is not linear: it does have this tree-structure.

Received March 4, 1991; revised March 22, 1993.

We now consider more exotic dictionaries in the shape of infinite sequences S of 0 and 1; S can be written as $S = \varepsilon^\wedge S'$, where ε is its first value and S' is the sequence obtained by dropping this first value and pushing everybody one step to the left. When consulting S, we use the rule that to know the n^o digit of S you need n units of time. A classical way to build a sequence S that functions as a dictionary for a set E consists first in grouping the characters by pairs to increase their number, in using 00 as a separator (i.e., a punctuation mark), 01 for 0, 11 for 1, and in listing the words in E first by size, then in alphabetical order. The algorithm for consultation requires a time which is an exponential function of the length of the word, and the easy counting argument which follows shows that we cannot find a substantially better way of listing the entries in such a sequential dictionary.

Let us consider an algorithm $M(S_1, \ldots, S_k)$ for consultation of a k-tuple of sequential dictionaries $S_1, \ldots, S_k$; that is, an algorithm which decides membership in a set E of digital words with the help of this tuple of sequences and which works in a time $t(n)$ which is infinitely smaller than 2^n, as the length n of the word tested tends to infinity $(t(n) \cdot 2^{-n} \to 0)$. While testing a word of length n, the algorithm can use only the first $t(n)$ elements of each dictionary, which leaves $2^{k \cdot t(n)}$ possibilities for their usable part. But there are 2^{2^n} possibilities for the subset of E of words of length n, so that, for a big n, we can find a set E_n of such words that cannot be the product of the algorithm M, no matter what tuple of sequential dictionaries it uses. Since the number of conceivable algorithms is countable, we can diagonalize and produce a set E that cannot be decoded from any finite set of sequential dictionaries by any algorithm working in subexponential time. In the course of the text, we shall be in a slightly more complex situation, where our computational skills will include the ability of instant comparison of two dictionaries, but this will not alter deeply the argument above.

What we have shown is that if we put on the sets of words the natural topology (where the sets E containing a given word form a clopen set), then those words which are accessible consulting a tuple of sequential dictionaries in subexponential time form a meager set. Also, if we put on the sets of words the natural measure, where the facts that distinct words belongs to E are independent events each of probability $1/2$, it is easy to see that they form a set of measure zero.

All this is very plane truth, and we apologize to our reader for its lack of mathematical sophistication. Our aim is to put the most debated $\mathrm{P} = \mathrm{NP}$ question in a new light; we do not believe that this will help to answer the question itself, but we hope that this will be the pretext of technical developments more substantial than the ones we can offer today.

Now that we have done the essential mathematics contained in this paper, we can begin its ideological part: we believe that there is a good possibility that the $\mathrm{P} = \mathrm{NP}$ question raised by [BSS 1989] in their model for computation over the reals may be not a really different question from the $\mathrm{P} = \mathrm{NP}$ question in the ordinary sense. We shall see that this is the case for the similar problem associated with a structure consisting of all sequential dictionaries; we are conscious that this model is much more rudimentary than the reals, on which, to speak properly, we prove nothing; we only believe that a real number, when encoding a pattern of 0 and 1, will behave more as a sequence than a tree.

In the first section, we define what we mean by a computation conducted in an arbitrary structure, and in the second section we explain what becomes the P = NP problem in this context. We give very few details; first because the notions involved are plain extensions of the classical ones, which are described in many textbooks (for instance [S 1990]) and which we assume to be familiar to our reader; second because a special case has been treated with a luxury of details in [BSS 1989] and that to obtain the general case one only has to do the obvious adaptations. But a deeper reason is that the problem that everybody tries to solve is not P = NP—I mean P $\neq$ NP of course—properly, but a purely combinatorial question on boolean circuits that would imply the negation of P = NP, so that, in this context, it is not really important to know precisely what an algorithm is!

In the third section, we show that computations in a structure without functions can be reduced to standard ones; in the fourth section, we explore the notion of computation in the model of sequential dictionaries; in the conclusion, we compare this model with the field of real numbers, and we pose a general version of the P = NP question.

§**1. Computations over an arbitrary structure.** We believe that there will be general agreement on the fact that an algorithm is an iterative process which is controlled by a finite-state automaton. At a given step, the automaton is in a certain state and instructs the operator to perform a certain action or to make a certain test and then to go to another state, ready for the next step; when a test is done, the automaton offers two possibilities for this next state, according to the answer. When we need more precision, we assume that the operator manipulates a multitape Turing machine; this will influence our vocabulary but does not mean that computations need to be done on an electronic computer. Our theory adapts as well if your favorite tool consists of a reed pen with a clay tablet, pebbles, toothpicks on a chessboard, or ink and paper.

Among the actions, some are linked to the behaviour of the machine: begin, stop, move along the tapes, and so on. Others are specific and depend on the nature of the objects that we admit as contents of the cells of the tapes. Our idea is to consider an arbitrary structure A, in a first-order finite language, and to use its relations as tests and its functions as actions.

We assume that A contains two specific elements that we name 0 and 1 and that the relations $x = 0$ and $x = 1$ belong to the language of A. The other relations $r_1, \dots, r_p$ of A are of arbitrary arities; we often assume that equality is among them, but this is not a necessity. We shall somehow contradict our finitary petition by the introduction of a relation $x = a$ for every element a of A; in other words, our machines will be permitted to test if the content of a given cell is equal to a; of course, because of the finite nature of its automaton, a given algorithm will use only a finite number of these a's, which we call its *parameters*. The process of adding names for elements of the base of a structure in this way is called by model-theorists inessential expansion, or expansion by constants, of this structure. We insist on the fact that 0 and 1, being named in the language of A, are not to be considered as parameters.

Similarly, we shall assume that the two constant (i.e., nullary !) functions 0 and 1 are present in the language of A, the other functions $f_1, \dots, f_q$ being of arbitrary arities; also, an algorithm can use the constant functions corresponding to its parameters $a_1, \dots, a_m$ (in others words, the operator can be requested to write an a_i in some cell).

A configuration of our machine is given by a state in the automaton, a finite number of cells of the tapes, each of them being filled by one element of A (copies of the same element can figure in different cells), the others cells being empty, and a scanned cell on each tape. We begin in the initial state, and our entry is a finite sequence of elements of A, which is written on the entry tape. The transition function from one configuration to the next is defined by the automaton: we perform a certain local movement of the scanned cells, or we collect a tuple of values in the neighbourhood of the scanned cells, in a way that has to be specified, compute the value it gives at the function associated to the state, and store it where directed, or check if this tuple satisfies the relation associated with the state, and then we go to the next state.

What we call the *standard case* is the case where the structure A is reduced to the two elements 0 and 1, the only tests are the relations $x = 0$ and $x = 1$, and the only functions are the two constant functions. This case corresponds to an ordinary Turing machine, working in a binary alphabet; by our conventions, every standard computation can be conducted directly in any structure.

We assume that our reader understands what is meant by a (partial) function from A^* to A^*, where A^* is the set of finite sequences of elements of A, computed by a machine, or by a machine testing membership to a given subset E of A^*. In this context, this set E will often be referred to as a "problem"; we shall say that a problem is *digital* if it consists only of sequences of 0 and 1 (i.e., of digital words) and that it is a *numerical* problem if it consists only of sequences of 0, so that membership to E depends only on the length of the sequence. Our model for computation is the obvious extension of the one introduced in [BSS 1989], where A is the set of real numbers with equality and order as relations and sum, product, and division as functions. We do not share in this paper all the motivations of these authors, and we will not be concerned with the theory of recursivity at a general level (halting sets, enumerable sets, and so on) that they develop. We are interested only in the second part of their work, dealing with complexity theory, and we shall focus our attention on algorithms *that always terminate in a time which is bounded by a function $t(n)$ of the length n of the entry*.

We measure time by the number of runs in the automaton; that is, tests and evaluations of functions are accomplished in a single unit of time, independently of the value of their arguments. The only thing that counts to evaluate the complexity of an entry is its length, and, in contrast to [BSS 1989], we never depart from this convention, even in the case of the ring of the natural numbers. For us, in a classical arithmetical computation, numbers must be introduced as sequences of binary digits.

§2. Computations, formulae, and circuits.

To describe how an algorithm works given an entry $\underline{x}$ of length n, we introduce tuples $\underline{y}_i$ of variables to represent the contents of the cells at the i^0 step; the length of $\underline{y}_i$ is the number of cells that the machine can visit in i steps. Because of the local character of its elementary movements, it is a linear function of i. Let us allow the algorithm to run m steps. Since our actions and tests are defined by atomic formulae in A, the sequence of the $\underline{y}_i$ is the unique one (once $\underline{x}$ and m are fixed) to satisfy a certain quantifier-free formula $f_{n,m}(\underline{x}, \underline{y}_1, \ldots, \underline{y}_m, \underline{a})$, possibly involving the parameters $\underline{a}$ of the machine. As a consequence, when the algorithm is used to decide membership in a certain set E and, by our conventions, terminates in a time $t(n)$, then the n-tuples which are in E are exactly those which satisfy a certain quantifier-free formula $f_n(\underline{x}, \underline{a})$. We observe that the formulae $f_n(\underline{x}, \underline{z})$, being purely linguistic objects, can be represented by sequences of binary digits (in this sequence of formulae the length of $\underline{x}$ is n, but the length of $\underline{z}$ is fixed, since it represents the tuple of parameters of the algorithm) and that they are obtained in a very simple way from the automaton controlling the machine, just by unfolding the algorithm, so that, provided that the time $t(n)$ is bounded by a recursive function, they form a recursive list in the ordinary (i.e., standard) sense.

Conversely, when we write, in the obvious manner, formulae in binary alphabet (except for their eventual parameters, that we leave as they are) and accept them as entries for our algorithms, then checking the satisfaction of a quantifier-free formula by a tuple of elements of A becomes a linear-time algorithm, so that any list $f_n(\underline{x}, \underline{z})$ of quantifier-free formulae which is recursive in the standard sense defines a computable set in the sense of A, after we have fixed an arbitrary value $\underline{a}$ for the tuple of parameters $\underline{z}$. In the case of the standard structure $\{0, 1\}$ this gives nothing but a circular characterisation of computability, but, in the general case, this reduces the notion of computability in A to the standard notion, with the only restriction that the time alloted for computations be standard recursive. This is a good reason to skip the details in the definition for the general case!

Moreover, we observe that in many cases, thanks to the coding power of the parameters, we can make the economy of the word "recursive" in the sentence above; for instance, as we shall see, a real number can be used to code any sequence of formulae, so that in the real field any sequence, recursive or not, of quantifier-free formulae will define a computable set—the only restriction is that the total number of their parameters must be finite.

When working at a more precise level than general recursivity, we must observe that because of the possibility of branching at each step, according to the answer of the test, the size of the formula f_n is big: it doubles whenever a test is performed; it is an exponential function of $t(n)$. There is a more compact way to handle this situation, which is to represent the formula f_n by a circuit c_n, which we explain now. We recall that a *boolean circuit* is a finite oriented graph without cycles, whose vertexes we call gates, and edges arrows; the input gates, which are the ones that receive no arrow, are labelled by (propositional) variables; the other gates are labelled by $\neg$, $\wedge$ or $\vee$, a $\neg$-gate receiving one arrow only, and other gates exactly two; output gates are the ones which emit no arrow. When given a distribution of truth-values to the labels of the entry gates, we propagate them along the circuit according to the following rules: when crossing a $\neg$-gate, we take the dual truth-value, at a $\wedge$-gate we take the infimum of the two coming values, and at a $\vee$-gate we take their supremum; we eventually obtain truth-values at the output gates, in a time which is a linear function of the size (= number of gates and arrows) of the circuit.

REMARK 1. The $\vee$-gates and the $\wedge$-gates of our boolean circuits receive only two arrows; a more usual definition tolerates arbitrary fan-in for these gates, so considering conjunction and disjunction as function symbols of floating arity: we can obviously transform this kind of circuit into our more restricted one, squaring its size (the depth of the circuit—i.e., the maximal length of its paths—being affected by a logarithmic factor).

2. Another technicality is that we have affected variables to the entry gates, so that different gates may be affected by the same variable; this is not really useful since the fan-out of a gate is arbitrary, so that we could assume that distinct entry gates represent different variables: we do that only to be consistent with the usual definition of a term.

We generalize the notion of a circuit to an arbitrary structure $\mathcal{F}$ whose language contains only functions; we label the gates by these functions, taking the care that the number of arrows coming to a gate corresponds to the arity of its label; we also have to order them, since functions are no longer symmetric. To every circuit c is associated a function from $\mathcal{F}^n$ to $\mathcal{F}^k$, where n is the number of variables associated to the entry gates of c and k is the number of its output gates. The first thing to do is to transform the structure A into a purely functional structure, which we call $\mathcal{F}(A)$; we keep the functions of A as they are; we replace each relation $r(\underline{x})$ by its characteristic function $\tau(r(\underline{x}))$, taking value 1 if r is satisfied, 0 if not. Moreover, we need a "selection operator", which is a function on three arguments (x,y,z) whose value is y when x is 0, and z when x is 1 (and which is computable in the structure A!). When A is a ring, we have nothing to add since we can use the polynomial $(1-x)\cdot y + x\cdot z$. In the general case, the best approach is to take the function $s(x,y,z)$ whose value is y when x is 0, z when x is 1, and x otherwise. This function will help us to treat definitions by cases and also to manipulate truth-values, since the restrictions of $s(x,y,1)$, $s(x,0,z)$, and $s(x,1,0)$ to $\{0,1\}$ are the usual boolean functions. The author of these lines is ready to admit that it is awkward to introduce relations in a computational context, where only functions should be considered, so that we are eventually forced to replace them by functions! We do that to take advantage of a background in Model Theory, which is developed usually in a relational language, and also because the definition of the NP class that follows is mainly of relational character!

Now, we say a few words on how to represent a computation by a circuit: we add to the $\underline{y}_i$ a tuple of digital variables $\underline{z}_i$ to represent the current state and the addresses of the current scanned cells. Since the number of possible configurations for the state and the scanned cells at the i^0 step is a polynomial in i whose degree is the number of tapes of the machine, the transition from $\underline{y}_i{}^\wedge\underline{z}_i$ to $\underline{y}_{i+1}{}^\wedge\underline{z}_{i+1}$ is done by a circuit of polynomial size, so that the circuit $c_{n,m}$ which outputs $\underline{y}_m{}^\wedge\underline{z}_m$ from $\underline{x}$ has a size bounded by a polynomial in m, whose degree is the number of tapes plus one. When the algorithm is used to decide membership to E, the characteristic function of this set is given, for entries of length n, by a circuit c_n (with only one output gate) of size bounded by a polynomial in $t(n)$. All that we say here is well known in the standard case (see, e.g., [STERN 1990]), so that we give no more details.

The evaluation of a circuit c at a given $\underline{x}$ is still done in a time which is a linear function of its size. It is the transformation of c into a *term* of $\mathscr{F}(A)$, or a quantifier-free formula of A, that requires exponential time. To transform a circuit into an equivalent term, which we can view as a circuit in the form of a tree, when two arrows leave a gate, we have no choice but to duplicate what we have already done. When writing terms, or formulae, in the form of a circuit, we suggest that it is not necessary to compute twice the value of the same subterm (which can be the truth value of a relation !). To illustrate this remark, let us consider this very simple example where f is a binary symbol

$$x \rightrightarrows f \rightrightarrows f \rightrightarrows f \rightrightarrows f \rightrightarrows f \rightrightarrows f \rightrightarrows f \rightrightarrows f.$$

The term corresponding to this circuit would fill the page!

The following and immediate observation is at the core of the P = NP question: *existential quantifiers neutralize the opposition between formulae and circuits*. When we introduce variables to represent the values that reach the gates, we transform the circuit into an equivalent existential formula of comparable size. For an algorithm, this means that the quantifier-free formula $f_n(\underline{x})$ is equivalent to an existential formula $e_n(\underline{x})$, of the form $(\exists \underline{y})\phi(\underline{x}, \underline{y})$ where ϕ is quantifier free, whose length is polynomial in $t(n)$. We observe that this existential quantifier is of the most primitive kind, since the quantified variables, which represent the successive configurations of the machine, are meant to specify a unique object. Let us take a very simple example in the standard case. Let E be the set of (digital) sequences containing an even number of 0's; it is decided by a read-only algorithm, with two states, which instruct to change the state when reading a 0 and to remain in the same state when reading a 1. The quantifier-free formula $f_n(x_1, \dots, x_n)$ is defined by induction as $(f_{n-1} \wedge x_n = 1) \vee (\neg f_{n-1} \wedge x_n = 0)$. If you make an attempt to write f_{10}, you will realize that its size is exponential, doubling at each run. To get the circuit, in this very simple case, it is enough to introduce the double implication symbol: c_n is nothing but $c_{n-1} \Leftrightarrow x_n = 1$! Or if you prefer drawing:

$$\begin{tikzcd}
& c_{n-1} \arrow[r] \arrow[d] & \neg \arrow[r] & \wedge \arrow[d] & \neg \arrow[l] & x_n \arrow[l] \\
x_n \arrow[r] & \wedge \arrow[rr] & & \vee & &
\end{tikzcd}$$

To obtain the existential formula, we introduce a sequence $y_0, \dots, y_n$ to represent the successive states of the automaton: $(\exists y_0, \dots, y_n)\, y_0 = 0 \wedge \dots \wedge (x_1 = 1 \Leftrightarrow y_1 = y_{i+1}) \wedge \dots \wedge y_n = 0$.

Of course, in this very simple case, the algorithm can be represented by a polynomial sequence of formulae provided that we change the language, replacing the usual boolean functions by the double implication; this feature is common to all rational problems, where the algorithm is nothing but the iteration of the transition function of the automaton.

Now come some definitions: a problem is P if it can be solved by an algorithm working in time $t(n)$ bounded by a polynomial in n. It is NP if its solution, on an entry $\underline{x}$ of length n, supposes the guess of a sequence $\underline{y}$ of elements of A, of polynomial length, and the performance of a polynomial-time algorithm M on the entry $\underline{x}^\wedge\underline{y}$. We follow [BSS 1989] for this definition of the class NP, although, on linguistic basis, the term "nondeterministic algorithm" would be better adapted to the case where the sequence $\underline{y}$ is bound to take values 0 or 1. In this last case, we speak of NDP problems. Since there are only a limited number of possibilities for the choice of a digital sequence, an NDP problem is soluble in exponential time; but there is no reason why an NP problem should be algorithmically soluble at all. In the case where the structure A is formed by the natural numbers with their sum and their product, then thanks to Matiasevich's Theorem the NP sets are exactly the recursively enumerable ones!

The following problem is obviously NP: check if an existential sentence $(\exists\underline{y})f(\underline{a},\underline{y})$ is true in A, where $f(\underline{x},\underline{y})$ is quantifier free and where $\underline{a}$ is a tuple from parameters in A (in this problem, the length n of the entry is the length of the sentence written in binary alphabet, except for the parameters which are left as they are). This example is typical, as indicated by the following extension of the first, and still unsurpassed, result in complexity theory.

THEOREM 1. *In every structure A, any NP-problem can be reduced by a polynomial-time algorithm to the problem of satisfication of existential sentences, i.e., of satisfiability of quantifier-free formulae with parameters in A.*

PROOF. The reduction algorithm is as follows: to check if $\underline{x}$ of length n belongs to E, introduce $\underline{y}$ of length $q(n)$ given by the definition of an NP-problem, form the formula $f_{n+q(n)}(\underline{x},\underline{y})$ associated to the P-algorithm, and then check its satisfiability. □

This theorem provides a unified treatment of two previous results. The first is Cook's original reduction of the standard NP-problems to satisfiability in propositional calculus. In the reals case, it leads eventually to the problem of satisfiability of polynomial equations of total degree 4, as explained in [BSS 1989].

From the considerations above it should be clear that a problem is P in the sense of A if and only if there exist a sequence of (binary codes of) circuit $c_n(\underline{x},\underline{z})$, $\underline{x}$ being of length n but $\underline{z}$ being of fixed length, which is produced by a standard polynomial algorithm (I mean an algorithm working in a time bounded by a polynomial in n, *not* in $\log n$) such that, for a fixed choice of the parameters $\underline{a}$ and for each n, $c_n(\underline{x},\underline{a})$ defines the set of n-tuples which are solutions of the problem. One may object that this uniformisation is algorithmic nonsense: to write down the circuit and then apply it to the data, instead of using the original algorithm, is just a (polynomial) waste of time (as if, in an iterated computation, we keep the literal expression up to the end and replace the variables by their values only at the last step!). But the polynomial level is robust enough to tolerate these kinds of oddities, which have, for us, the practical effect reducing the definition of the P-algorithms in an arbitrary structure to the classical case: whatever will be the starting point of our intuition of what should be a computation, we shall necessarily obtain as the P-problems the class that we have described.

In conclusion, the question of whether $\mathrm{P}=\mathrm{NP}$ relative to a structure A is equivalent to the existence of a polynomial algorithm to solve the corresponding satisfiability problem. Of course, this question is meaningful only when A eliminates quantifiers (after naming some of its elements); if not, the answer is trivially no!

In other words, $P = NP$ in A means that there exists a tuple $\underline{a}$ of elements of A and a P-algorithm *in the sense of* A, which transforms every existential formula of the structure $(A, \underline{a})$ into an equivalent quantifier-free formula *given in the form of a circuit.*

§3. **Structures without functions.** By our conventions, any standard computation can be conducted in every model. In this section, we shall study structures whose computational power is no greater than for the standard one. If the structure A is finite, even if it has relations and functions, we can code everything by standard objects; we represent elements of A by binary codes, functions by their graph, and relations by their truth table, so that any computation can be simulated by a standard one, the necessary time being multiplied by a fixed constant (we do have quantifier elimination after naming every element in A!). But how to compare, with respect to the $P = NP$ question, an infinite structure with the standard one? We can have a look at the digital problems, a notion that makes sense for both structures: if A has the same P or NP digital problems as the standard model, this is an indication of a similitude to the standard case; this similitude will be more convincing if, moreover, satisfiability in A can be polynomially reduced (in the sense of A!) to a digital problem (i.e., if A has a digital NP-complete problem), as will be the case in the structures that we consider now, which have no functions in their signature (that is, the only functions used in course of computation are constant). In this situation, we shall say that $P = NP$ in A *reduces to the standard case*, since $P = NP$ in A is equivalent to the standard question. Observe also, that if A has the same digital P-sets as the standard model, then $P = NP$ in A would imply $P = NP$ standardly, since the standard satisfiability problem is also NP in A.

To avoid complications, we shall assume that equality is in the language of A.

THEOREM 2. *If the structure A is purely relational, then its digital P-sets are the same as the standard ones.*

PROOF. Since there are only constant functions, an algorithm, being given a digital entry, can print only 0 or 1, or one of its parameters $a_1, \dots, a_m$; so the answer to the tests involves only the restriction of A to this set of parameters, which is finite, and can be coded in binary digits. □

At the NP-level, we must face the problem of controlling the expressive power of the elements represented by the quantified variables. This problem vanishes when we have elimination of quantifiers in a finite relational language. Observe then that the structure A is omega-categorical.

THEOREM 3. *If A is purely relational, with a finite number of relations, and admits quantifier-elimination, then the satisfiability of quantifier-free formulae with parameters in A is polynomial-time reducible to the satisfiability of quantifier-free formulae without parameters.*

PROOF. Call the type of a tuple $\underline{a}$ in A the conjunction $\operatorname{tp}\underline{a}(\underline{x})$ of atomic formulae, and negation of atomic formulae, satisfied by $\underline{a}$. $\operatorname{tp}\underline{a}(\underline{x})$ is produced from $\underline{a}$ in polynomial time, the degree of the polynomial being sensitive to the arities of the relations involved. By quantifier-elimination, checking the satisfiability of $f(\underline{a}, \underline{y})$ is the same thing as checking the satisfiability of $\operatorname{tp}\underline{a}(\underline{x}) \wedge f(\underline{x}, \underline{y})$. □

So what we call a *complete pure type* $\mathrm{tp}(\underline{x})$ is the conjunction, for each atomic formula that we can form with $\underline{x}$, of this formula or of its negation. We shall say that the type is *equality free* if it declares that all the variables in $\underline{x}$ represent distinct elements. The following result helps to provide applications of Theorem 3.

THEOREM 4. *Under the hypothesis of Theorem* 3, *there is a reduction by a standard NP-algorithm of the satisfiability problem in A, for quantifier-free formulae without parameters, to the problem of checking that a complete pure equality-free type is satisfiable in A.*

PROOF. To check the satisfiability of $f(\underline{y})$, we first guess the equality type of $\underline{y}$. After renaming the variables, we obtain a formula $g(\underline{z})$ in which all the z's are assumed to be distinct. We then guess the type of $\underline{z}$. After checking that it is consistent with the theory of A, we check the propositional validity of $f(\underline{z}) \wedge \mathrm{tp}(\underline{z})$. □

Application of this theorem to the following structures shows that they have a standard NP satisfiability problem:

an infinite set in the pure equality language; nothing to check;

the generic binary, or n-ary relation (i.e., the model-completion of the theory of n-ary relations); nothing to check: all types are consistent;

dense linear order, without end points; one needs to check that a total, antisymmetric, binary relation has no cycle: any algorithm for sorting will do that.

In each of the structures above, the consistent types form a standard P-set. Since we can replace the guess of a tuple by the guess of its type, they satisfy NP= NDP; so, their digital NP sets are the standard ones.

§4. Arborescent and linear dictionaries.

In contrast with the preceding section, we shall now examine cases where individuals in A may have a great power for coding. We consider first the structrure Θ of arborescent dictionaries, whose basis is formed with the two constants 0 and 1, plus the set of all binary trees labeled by 0 and 1 (formally speaking, such a tree T is a function from $\{0,1\}^*$ to $\{0,1\}$), whose language has only equality as relation symbol, and the three functions which associate to a tree T its left subtree $g(T)$, its right subtree $d(T)$, and the label of its root $r(T)$. From the point of view of model theory, Θ is not too sophisticated. It is a stable, nonsuperstable, trivial theory with D.O.P., and it admits elimination of quantifiers. These facts are not absolutely evident, but we nevertheless leave their proofs as an exercise to our reader, giving the hint that Θ has some parenthood with the free pairing function described in [BP 1988] (this free pairing function modelizes the notion of files containing only subfiles, but no information at all!).

As was explained in the introduction, in this model every digital problem is solvable in linear time, and in a uniform way, with the help of the appropriate parameter. In the next paragraph we show, thanks again to a trivial counting argument, that Θ satisfies P $\neq$ NP.

Consider the following problem: being given a tuple of trees $(T_1, \dots, T_n)$, check if there is a 0 at the n^0 level of T_1. This problem is obviously NP, and even NDP: introduce a digital sequence to represent the path to follow in T_1. To show that it is not P, assume that it is solved by an algorithm working in subexponential time

$t(n)$. Using parameters $a_1, \ldots, a_m$; choose a large n so that $t(n) < 2^n$ and an entry $T_1, \ldots, T_n$ such that all their first n levels are filled by 1, but which have somewhere above a distribution of 0 and 1 that ensures that all the trees defined from the T_1's by terms in g and d of length less than $t(n)$ are different and different from the trees of the same kind that we can form with the parameters. So, when an equality of subtrees is tested in the course of the computation, the answer is systematically no. By hypothesis, this computation ends by a negative answer. In the course of it one of the 2^n values of level n of T_1 has not been checked. Let us change this value into a 0, leaving everything else fixed. The algorithm gives the same answer, which is a contradiction.

Let us now consider the structure Σ of sequential dictionaries formed by the two constants 0 and 1 plus all the infinite sequences S of 0's and 1's, in the language containing equality, the function $r(S)$ associating to S its first value, and the "successor-function" $s(S)$ which associates to S the sequence obtained by cancelling $r(S)$ from it. Σ has a very plain model theory. It is superstable, trivial, of rank one, and its quantifier elimination is easy to establish. We distinguish in Σ two kinds of 1-types over the empty set. In the first, there is an equality of subsequences which implies that the sequence of values is eventually periodic. Such a type is realized exactly once in each model of the theory of Σ. Other types are determined by their sequence of values which is arbitrary (it can be periodic: in this case the type is omitted in Σ!). Types of this second kind are realized as many times as you want, including none. They are orthogonal provided that they do not end by the same infinite sequence.

We may think that there is some artificiality in computing over Σ, where files can be subfiles of themselves, where we can check instantly that two files are equal (which does not amount to saying, if we keep in mind the saturated models of the theory of Σ, that they have the same content!) But the point is that these features are imposed on us when we interpret Σ in the field of the reals. We know from the introduction that the digital problems in Σ are soluble in exponential time but that the majority of them are not soluble in polynomial time. The numerical problems are, of course, soluble in polynomial (quadratic) time. We shall see that, in spite of its coding abilities, Σ is rather tame as far as complexity of algorithms is concerned. Once one understands what the saturated models of the theory of Σ are, the elimination of quantifiers is the direct consequence of a simple back-and-forth argument, so that the truth of $(\exists \underline{y})\phi(\underline{x}, \underline{y})$, where ϕ is quantifier free, is known when we know the quantifier-free type of $\underline{x}$, but we have to take better care of this elimination to show that this truth does not need an explosive quantity of the free type of $\underline{x}$.

Considering a tuple $\underline{x}$ of variables, each of them representing an element of $\Sigma - \{0, 1\}$, we define an n-type as consisting of the description of the first n values of the elements of $\underline{x}$ (we write if $r(s^k(x_i)) = 0$ or 1 for every $k \leq n$), plus the choice between equality and inequality for each pair of the first n successors of our sequences (we write if $s^k(x_i) = s^1(x_j)$ or not for every k, $1 \leq n$). An n-type over $\underline{a}$ is an n-type with variables $\underline{x}{}^\wedge\underline{y}$, whose $\underline{x}$-part is the n-type of $\underline{a}$.

LEMMA 5. *The satisfiability of an n-type over $\underline{a}$ with $k - 1$ new variables depends only on the kn-type of $\underline{a}$.*

PROOF. When we add just one y, the worst thing that we can do is to declare $y = s^n(a_i)$, forcing us to consider $s^{2n}(a_i)$. When we add $k-1$ variables, we are not forces to dig deeper than the kn^0 level. □

By definition, the satisfaction of a quantifier-free formula of length n depends only on the n-type: for an existential formula, thanks to Lemma 5, it depends only on the n^2-type.

LEMMA 6. *The satisfiability in Σ of an n-type can be decided in polynomial time.*

PROOF. Check that the equalities do not contradict the knowledge we have of the first digits of each sequence and that the inequalities are consistent with the uniqueness of periodic sequences. □

COROLLARY 7. *In Σ, the satisfiability of quantifier-free formulae with parameters is polynomial-time reducible to the satisfiability of quantifier-free formulae without parameters, which is itself a standard NP-problem.*

PROOF. To check the satisfiability in Σ of a formula $f(\underline{a}, \underline{y})$ of length n, we first produce from $\underline{a}$, in polynomial time, its n^2-type $\mathrm{tp}(\underline{x})$. By Lemma 5, we just have to check the satisfiability of $f(\underline{x}, \underline{y}) \wedge \mathrm{tp}(\underline{x})$ which is without parameters. To check the satisfiability of a quantifier-and-parameter-free formula $g(\underline{z})$ of length m, we guess the n-type of $\underline{z}$, which is a guess in a standard P-set by Lemma 6. □

So we see that, if $\mathrm{P} = \mathrm{NP}$ standardly, then the same is true in Σ.

We also observe that, in Σ, a P-algorithm without parameters which decides a digital problem is standard for a trivial reason: in the absence of parameters, the algorithm will print only 0 and 1 on its tapes. So that the reciprocal, that is, to infer from $\mathrm{P} = \mathrm{NP}$ in Σ that the same is true standardly, is just a question of neutralization of the role of the parameters. We shall use the more drastic trick to do that, which is to alter the $\mathrm{P} = \mathrm{NP}$ question itself! For this, we introduce a class which we would like to call "pseudopolynomial", or "circuit-size polynomial", but which is otherwise known as "nonuniform P" and is often denoted by P/poly. We hope that the reader will allow us to use the resources of the Greek Alphabet for its name: we shall say that a problem E is Π if there exist a polynomial $p(n)$ and for each n a circuit c_n, representing a quantifier-free formula with n variables (not counting the fixed tuple of parameters), whose size is bounded by $p(n)$ and which solves the problem E for entries of length n. We define the class $\mathrm{N}\Pi$ from the class Π, in the same way as NP was defined from P. One sees immediately that the analog of Cook's Theorem remains valid for the classes Π and $\mathrm{N}\Pi$: satisfiability is $\mathrm{N}\Pi$-complete.

It is a very remarkable fact that all the authors who have made a convincing attack of the $\mathrm{P} = \mathrm{NP}$ question, whether they relativize it by the addition of an oracle or consider only positive circuits, always prove a stronger statement than $\mathrm{P} \neq \mathrm{NP}$: they exhibit an NP problem which is not Π. It seems that the only manageable way to settle the $\mathrm{P} = \mathrm{NP}$ question is to replace it by a purely combinatorial statement and to eliminate its algorithmic flavor, so relegating the very notion of recursive function to the museum of accessories which escape any efficient mathematical treatment.

A problem can be Π without being P, because the existence for each n of such a short circuit does not mean that there is a short way to find it. Observe that any numerical problem, recursive or not, is Π standardly. But, in a structure like

Σ, we have $P = \Pi$; indeed, any sequence c_n of words of polynomial growth can be listed edge to edge (separated by punctuation marks!) in an element of Σ, so that the decoding of this dictionary is done in polynomial time (the degree of the polynomial is incremented by one: this would not be the case if we were using *combs*, that is, sequences of sequences, instead of sequences). The same is true in Θ, of course. What we have shown, in the introduction, is that the majority of the digital problems are not standardly Π!

Observe finally, that the digital problems which are P in Σ are exactly those which are Π in the standard sense, because, when n is fixed, the parameters interfere in the algorithm only by their $t(n)$-type; the elimination of this finite object of polynomial size does not increase substantially the size of the circuit. Since instead of guessing a tuple of elements we can guess its $t(n)$-type, NP = NDP in Σ, so that the digital NP-sets are exactly the standard NΠ-ones. In others words, P = NP in Σ is equivalent to the standard $\Pi = \mathrm{N}\Pi$ question.

Conclusion. Σ is coded in the obvious way in the field of the reals. For any x, $0 \leq x < 1$, we define $r(x) = 0$ and $s(x) = 2x$ if $2x < 1$, $r(x) = 1$ and $s(x) = 2x - 1$ if $2x \geq 1$. This interpretation misses the sequences which end with an infinite sequence of 1's, but we can ignore this detail. All computations in Σ can be simulated in the reals, and in **R** any digital problem is soluble in exponential time and any numerical problem in polynomial (quadratic) time. In **R**, polynomial-time complexity is equivalent to polynomial circuit-size complexity.

Besides these trivialities, we prove nothing on the real field, and, as was explained in the introduction, the analogy that we see between **R** and Σ rests on an act of faith. For instance, we believe that the majority of digital problems are unsoluble in **R** in polynomial time, although we are unable to provide a single example of such a problem. We do not even know that there is no uniform polynomial-time algorithm that will solve any digital problem with the help of appropriate real parameters!

We are also influenced by the fact that the only practical way to introduce real numbers in a computation is in the form of a sequence of digits. When doing so, the main difficulty is that, when sequences are equipped with their natural topology, the sum and the product are discontinuous functions. This difficulty is superbly ignored by computer scientists but will become an obsession for anybody trying to make a reverse coding of **R** into Σ. It may lead to cumbersome questions, possibly unnatural as far as computation is concerned, and remote from the motivations of [BSS 1989], but which, in our opinion, cannot be avoided when dealing with the P = NP question in **R** in the form they asked it. And if the connection in which we believe is real, then there is a possibility that this question may not be different from the standard circuit-size complexity question!

Finally, we observe that the relativizations of the **P** = NP question that we have considered here are of a very different nature from the ones that are obtained by the introduction of an oracle. Certainly, elements of our structure can have a great power for coding, but, since they are admitted as entries for the algorithms, it seems doubtful that a finite number of parameters will encompass the whole expressive power of the structure. In others words, we can ask if there exists any structure at all which satisfies $P = NP$.

Note. The referee of this paper has made many valuable suggestions; one of them is to mention two works that have been produced independently of ours, generalizing the Blum-Shub-Smale Model; they are Edward Bishop, *Machines and complexity over the p-adic numbers*, Ph.D. Thesis, Berkeley, California, 1991; and Nimrod Miggido, *A general NP-completeness result*, to appear in the Proceedings of the Smale Festival edited by Hirsh, Mardsen and Shub, Springer-Verlag, Berlin and New York, 1993.

REFERENCES

[BP 1988] Elisabeth Bouscaren and Bruno Poizat, ***Des belles paires aux beaux uples***, this Journal, vol. 53, pp. 434–442.

[BSS 1989] Lenore Blum, Mike Shub and Steve Smale, On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions, and universal machines, ***Bulletin of the American Mathematical Society,*** vol. 21, pp. 1–46.

[S 1990] Jacques Stern, ***Fondements mathématiques de l'Informatique***: ***logique et complexité***, McGraw-Hill, Paris.

UNIVERSITÉ CLAUDE BERNARD (LYON-1)
MATHÉMATIQUES, BÂTIMENT 101
69622 VILLEURBANNE-CÉDEX, FRANCE

E-mail: kazak@lan1.univ-lyon1.fr