PROCEEDINGS OF SYMPOSIA
IN APPLIED MATHEMATICS
Volume 38

Computational Complexity Theory

Juris Hartmanis, Editor

American Mathematical Society
Providence, Rhode Island

LECTURE NOTES PREPARED FOR THE
AMERICAN MATHEMATICAL SOCIETY SHORT COURSE

COMPUTATIONAL COMPLEXITY THEORY

HELD IN ATLANTA, GEORGIA
JANUARY 5–6, 1988

The AMS Short Course Series is sponsored by the Society's Committee on Employment and Educational Policy (CEEP). The series is under the direction of the Short Course Advisory Subcommittee of CEEP.

Library of Congress Cataloging-in-Publication Data

Computational complexity theory.
p. cm. – (Proceedings of symposia in applied mathematics, ISSN 0160-7634; v. 38)
Includes bibliographies.
1. Computational complexity. I. Hartmanis, Juris. II. American Mathematical Society. III. Series.
QA267.C592 1989 511.3 89-6857
ISBN 0-8218-0131-7 CIP

1980 *Mathematics Subject Classification.* (1985 *Revision*).
Primary 68Q05, 68Q10, 68Q20, 68Q75.

Printed in the United States of America.
The paper used in this book is acid-free and falls within the guidelines established to ensure permanence and durability. ♾
Portions of this publication were typeset using $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, the American Mathematical Society's TEX macro system.

Preface

During the last twenty-five years computational complexity theory has developed into one of the central and most active research areas of computer science. It has grown into a rich and exciting mathematical theory whose development is motivated and guided by computer science needs as well as technological advances. At the same time, it is clear that complexity theory, dealing with the quantitative laws of computation and formal reasoning, is concerned with issues and problems of direct interest to many other disciplines as well. In particular, complexity theory is of considerable interest to mathematics and some of the key open problems in complexity theory are basic questions about the nature of mathematics.

The six lectures, on which this volume is based, were given at the AMS Short Course on Computational Complexity Theory in conjunction with the ninety-fourth Annual Meeting of the Society on January 5–6, 1988.

It is our hope that these proceedings will be a good introduction to the study of computational complexity theory, a helpful guide to different aspects and applications of this work and, in selective areas, communicate recent results and interesting open problems.

J. Hartmanis, Editor

Table of Contents

Proceedings of Symposia in Applied Mathematics
Volume 38, 1989

Overview of Computational Complexity Theory

JURIS HARTMANIS

ABSTRACT. The purpose of this chapter is to introduce the reader to computational complexity theory by presenting the basic computational models, definitions, and key results. Emphasis is given to motivation and meaning of results and open problems at the expense of giving mostly outlines of proofs. Many results are mentioned without even hinting of proofs to indicate directions of further development of complexity theory.

1. Introduction. The systematic study of computational complexity theory has developed into one of the central and most active research areas of computer science. It has grown into a rich and exciting mathematical theory whose development is motivated and guided by computer science needs and technological advances. At the same time, it is clear that complexity theory, dealing with the quantitative laws of computation and reasoning, is concerned with issues and problems of direct interest to many other disciplines as well. It is quite likely that in the overall impact of computer science on our thinking, complexity theory will be recognized as one of the most influential intellectual contributions.

In particular, complexity theory is of considerable interest to mathematics and some of the key open problems in complexity theory are basic questions about the nature of mathematics. We believe that the future development of mathematics will be very strongly influenced by computer science. We expect that in the coming decades the strongest outside influence on the development of mathematics will come from the extended use of computing and from concepts and problems arising in computer science. This influence on the development of mathematics is likely to be as strong as the earlier influence of the natural sciences.

Looking at the development of computational complexity theory, one can observe very definite turning points and descry trends and styles in research. The early work in complexity theory defined the basic computational models, established the standard complexity measures, created the guiding research

1980 *Mathematics Subject Classification* (1985 *Revision*). Primary 68Q05, 68Q20, 68Q75.
This research was supported by NSF Research Grant DCR 85-20597.

paradigms, established complexity classes as the dominant structural concept and yielded the initial results about algorithms and complexity classes on which complexity theory is built. Since the early seventies, motivated by Cook's discovery of complete languages in NP and Karp's extension of these concepts to combinatorial problems and other complexity classes, complexity theory has been particularly concerned with the study of the feasible complexity classes below $PSPACE$. The new concepts of resource bounded reductions and complete languages for natural complexity classes were modifications of central concepts from recursive function theory, but in the new setting they initiated an unexpectedly rich and exciting area of research. This work also raised some of the most interesting, and apparently some of the hardest open problems in computer science. The best known of these problems is the $P = ?NP$ question.

It is quite surprising and impressive that in the whole awesome mathematical arsenal, there seem to be no results or techniques to directly attack these separation problems. The questions about the computational power of various computing models and relations between different resource bounded computations have raised new mathematical problems which do not appear to be amenable to any known techniques. This points out dramatically that we still have only a fragmentary understanding of the quantitative aspects of computing and that most likely new proof techniques will be required to gain a deeper outstanding. We can expect that major intellectual battles will be fought and that we will have quite a few surprises.

It should also be recalled that some of these separation problems are questions about the basic nature of mathematics and thus formal reasoning. In essence, the $P = ?NP$ problem is the problem about the quantitative computational difference between finding a proof for a theorem and checking a given proof for correctness.

More recently, one can perceive in the development of complexity theory a growing interest in the structural properties of the feasible computations. This research has a definite flavor and has emerged during the last decade as a cohesive subfield with a rich set of interlocking results and problems. It is partially characterized by interest in global properties of complexity classes, the relations between complexity classes, logical implications among certain unsolved problems about complexity classes, and the use of relativization to explore possible relationships. We shall refer to this area of work as structural complexity. The complimentary research stream in complexity theory is more concerned with specific algorithms and has a more pronounced combinatorial flavor.

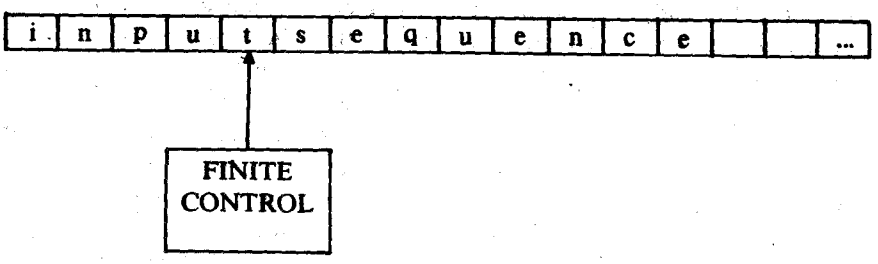
The first three chapters of this book are concerned with structural complexity problems.

2. Basic Models and Complexity Measures. The work on foundations of mathematics, and particularly Goedel's incompleteness results, motivated the

search for a precise formulation of what is and is not effectively computable. The intuitively most satisfying answer to this quest was given by A. M. Turing in 1936 in terms of an abstract computing device, the Turing machine. Turing's formulation has been shown to be equivalent to several other formulations of the concept of computability and we now accept the Church-Turing Thesis: *the intuitive concept of "effectively computable" is equivalent to computability by Turing machines.* All through these lectures, we will use the Turing machine model and its modification as our basic computer model.

We believe that effective computability is one of the fundamental concepts of mathematics and should be familiar to any mathematician. Still, we will shortly describe the basic one-tape Turing machine model; extensions to multi-tape models are obvious [HU].

A Turing machine M, as illustrated below, has a finite control, an input tape that is divided into cells, and a tape head that scans one tape cell at a time. The tape is infinite to the right. The n leftmost cells, for $n \geq 0$, hold the input, which is a sequence of symbols, chosen from a subset of the fixed, finite set of tape symbols, called input symbols. The remaining cells each hold the blank, which is not an input symbol.



A Turning Machine.

FIGURE 1

A move of the Turing machine is determined by the symbol scanned by the tape head and the state of the finite control. It consists of changing of the state of the finite control, (erasing and) printing a symbol on the tape cell scanned by the head, and moving the head one cell left or right.

Formally, a Turing machine (TM) is described by:

$$M = (Q, \Sigma, \Gamma, \delta, q_0), \text{ where}$$

Q is the finite set of *states*,
Γ is the finite set of *tape symbols*, including B, the *blank*,

Σ, $\Sigma \subseteq \Gamma$, not including B, is the set of *input symbols*,
δ is the *next move function* (δ may be a partial function)

$$\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\},$$

q_0, $q_0 \in Q$, is the *start state*,
$F \subseteq Q$ is the set of *accepting states*.

A finite sequence x over Σ, x in Σ^*, is accepted by M iff M started in state q_0 on the left most tape cell, reaches an accepting state q, q in F, in a finite number of moves. The set of sequences accepted by M is denoted by $L(M)$.

Turing made two key observations about these machines.

First, we can effectively list all possible one tape Turing machines (and this enumeration can be done by a Turing machine).

Second, there exist universal Turing machines which can simulate any other TM. More precisely, let $M_1, M_2, \ldots$, be a standard enumeration of one tape TM's. There exists a *universal* TM, M_u such that for all x and i (and an easily computable pairing function $\langle x, M_i \rangle$)

$$M_u(\langle x, M_i \rangle) \equiv M_i(x)$$

(*i.e.*, $\langle x, M_i \rangle \in L(M_u) \Leftrightarrow x \in L(M_i)$).

From these two facts (and a bit of programming), it follows that there are many recursively unsolvable problems (i.e., problems not solvable by TM's and therefore, by the Church-Turing Thesis, not effectively solvable). For example, the halting problem for TM's "Does M_i halt on input x?" is not recursively solvable. To see this, observe that the existence of a TM M_D which decides the halting problem would permit us to diagonalize over all Turing machines and exhibit a TM not on our list, producing a contradiction.

More precisely, let M_{new}, using M_D, the enumerator of $M_1, M_2, \ldots$ and the universal machine, M_u act as follows: for each i, M_{new} accepts $1^i \Leftrightarrow M_i$ does not accept 1^i. Clearly, if M_D exists, M_{new} is a TM, but it is not on our list of TM's, since its action differs from each machine M_i on input 1^i. Thus, M_{new} does not exist and (by Church-Turing Theses) we conclude that the halting problem is not solvable.

The above Turing machine model can be easily be extended to multi-tape Turing machines and to Turing machines that compute functions.

For computational complexity considerations, two very important extensions of the Turing machine model are the generalizations to nondeterministic Turing machines and to oracle Turing machines.

A *nondeterministic Turing machine* is a TM whose *next move function* δ maps $Q \times \Gamma$ into a subset of $Q \times \Gamma \times \{L, R\}$ (rather than a single element of $Q \times \Gamma \times \{L, R\}$). Thus, a nondeterministic TM may have choices in selecting the next move. Such a machine *accepts* input x iff there is a sequence of choices of the allowable moves which lead from the starting configuration to an accepting state.

It can be seen that if we put no time bounds on the computations, nondeterminism does not increase the computational power of Turing machines

since a deterministic machine can explore all possible choices of a nondeterministic TM and see if any of them leads to an accepting state. On the other hand, as we will see very soon, for time-bounded computations the question of nondeterminism versus determinism leads to the key open problems of complexity theory, which also turn out to be profound questions about the quantitative nature of mathematics.

Finally, we will introduce an *oracle Turing machine*, which is used extensively in complexity theory to investigate relativized computations.

An *oracle TM* is a multi-tape TM M with a distinguished work tape, the *query tape*, and three special states: *QUERY-STATE*, *YES-STATE*, and *NO-STATE*. During the computation, M may enter the *QUERY-STATE* and in one operation it is transferred to *YES-STATE* if the string currently on the query tape is in the oracle set A; otherwise, M is transferred to *NO-STATE*. In either case, the query tape is erased.

The set of strings accepted by M *with oracle set* A *is*:

$$L(M^A) = L(M, A)$$
$$= \{x | \textit{there is an accepting computation of } M \textit{ on } x \textit{ with oracle set } A\}.$$

The Turing machine model, first introduced to study what is and is not effectively computable, has also served well in the exploration of computational efficiency. The two most dominant complexity measures for computations are the time and space (tape or memory) used in the computation.

Let T and S be functions from natural numbers to natural numbers, $T: N \to N$, $S: N \to N$. Let Σ be a fixed finite alphabet. Σ^* denotes all finite sequences over Σ. For x in Σ^*, $|x|$ is the number of symbols in x.

A TM, M, runs in time $T(n)$ iff for all x in Σ^*, $M(x)$ halts in $T(|x|)$ or fewer steps.

A TM, M, runs in space $S(n)$ iff for all x in Σ^*, $M(x)$ uses no more than $S(|x|)$ tape cells.

A key concept in complexity theory is the complexity class consisting of all languages acceptable in a given resource bound. We will consider complexity classes defined by time or space bounded Turing machines.

$$TIME[T(n)] = \{A | A = L(M) \text{ and } M \text{ runs in time } T(n)\}$$
$$SPACE[S(n)] = \{A | A = L(M) \text{ and } M \text{ runs in } S(n) \text{ space}\}.$$

It can be shown that a slight increase in the time or space bounds yields more computational power. We state this result for space bounds.

To make this precise, we need tape bounds which are easy to compute. More precisely, we say that $S: N \to N$, $S(n) \geq n$, is *tape or space constructible* if there exists a TM, M, which runs in $S(n)$ tape and for all x marks off (uses) exactly $S(|x|)$ tape cells.

THEOREM. *If $S(n)$ is space constructible and*

$$\lim_{n \to \infty} \frac{R(n)}{S(n)} = 0,$$

then

$$SPACE[R(n)] \subsetneq SPACE[S(n)].$$

This result is proved by a simulation-diagonalization argument. An $S(n)$ space bounded machine M_D, using the universal machine, simulates machines using less than $S(n)$ space and accepts if the simulated machine rejects and rejects if the simulated machine accepts. Because of the limit conditions every $R(n)$ space bounded machine will be simulated on some sufficiently long input for which the simulation can run to completion, and therefore $L(M_D)$ is in $SPACE[S(n)] - SPACE[R(n)]$.

A related but slightly more complicated theorem holds for time bounded computations. The proofs of the corresponding results for nondeterministic time bounded computations are quite complicated [HU].

It should be noted that these results also hold for relativized computations. We say that the proof techniques in these results *relativize*. We will see later that there are many important problems in complexity theory that cannot be solved by proof methods that relativize. The classic $P =?NP$ problem is of this type. In general, a proof that a problem cannot be solved by proof techniques that relativize is a strong indication that the problem is out of reach of our current proof techniques [H]. Intuitively, we view the relativization of a problem in two contradictory ways (say $P^A = NP^A$ and $P^B \neq NP^B$) as an "independence" result for proof techniques that relativize. At present, we do not have a precise logical formulation of these concepts.

3. P, NP, and PSPACE. In this section we introduce the most important computational complexity classes: the deterministic and nondeterministic polynomial time computations, P and NP, and the polynomial space bounded computations, $PSPACE$. We also introduce the key concept of reductions between problems and the concept of complete problems. Let $D_1, D_2, \ldots$ be a standard enumeration of deterministic TM's with standard (easily detectable) clocks which stop their computation in polynomial time (i.e., each TM M_i appears on the list for each $k, k \geq 1$, with a clock stopping it at $n^k + k$, where $n = |x|$). Let $N_1, N_2, \ldots$ be the corresponding enumeration of clocked nondeterministic TM's.

The class of polynomial time computations forms a good mathematical model of the feasibly computable problems. We define it first in terms of language recognition:

$$P = \{L(D_i) | i \geq 1\}.$$

The class of nondeterministic polynomial time computations,

$$NP = \{L(N_i) | i \geq 1\},$$

contains many important practical problems, and consists of the computations which can be solved in polynomial time with a "lucky guess" of the solution followed by a verification that it is indeed a solution. It can be seen

that L is in NP iff there exists a polynomial time computable predicate, $R(\ ,\ )$ and a polynomial $p(\ )$ such that:

$$L = \{x | (\exists y)[|y| \leq p(|x|) \text{ and } R(x,y)]\}.$$

To indicate that the quantifiers $\exists$ and $\forall$ are polynomial bounded, we will write $\exists^P$ and $\forall^P$.

$$PSPACE = \{L(M_i) | M_i \text{ runs in space } n^k + k, \text{ some } k \geq 1\}.$$

$$NPSPACE = \{L(N_i) | N_i \text{ runs nondeterministically in space } n^k + k, \text{ some } k \geq 1\}.$$

Since we know that $NSPACE[S(n)] \subseteq SPACE[S(n)^2]$ [**HU, Sa**], we see that

$$PSPACE = NPSPACE.$$

It is easily seen that

$$P \subseteq NP \subseteq PSPACE.$$

It is widely believed that these containments are proper, but so far these separation problems have resisted all attempts to solve them. We view them as among the most important and intriguing open problems.

One of the most important properties of NP, and many other complexity classes, is the existence of complete problems to which all other problems in the class can be reduced to by simple computations.

A language A, $A \subseteq \Sigma^*$, is *many-one polynomial time reducible* to B, $B \subseteq \Gamma^*$, iff there exists a polynomial time computable function $f: \Sigma^* \to \Gamma^*$ such that

$$x \in A \Leftrightarrow f(x) \in B.$$

We write $A \leq_m^P B$.

A language A is NP *hard* iff:

$$(\forall B)[B \in NP \Rightarrow B \leq_m^P A].$$

A language A is NP *complete* iff A if in NP and A is NP hard.

In other words, A is NP *complete* if A is in NP and for any B in NP there is an f in PF (polynomial time computable functions) such that

$$x \in B \Leftrightarrow f(x) \in A.$$

The $\leq_m^P$ completeness for other complexity classes is defined similarly.

The existence of complete languages for NP and $PSPACE$ was not so surprising. The great surprise was that there were so many natural complete NP problems which appear in computer science, operations research, mathematics and engineering. After Cook's discovery that SAT, the set of satisfiable Boolean formulas in Conjunctive Normal Form (CNF), was NP complete a veritable goldrush fever was created in the search for different NP complete problems [**Co**]. The discovery of new NP complete problems has slowed quite a bit, but it is still continuous and we now know hundreds of different

NP complete problems [GJ, HU]. There seem to be fewer natural $PSPACE$ complete problems, but even this arsenal is quite impressive.

Today we accept NP completeness as a central computational complexity concept and as a strong indication that any problem shown to be NP complete or hard is not feasibly computable.

The existence of NP complete problems, as well as complete problems for many other complexity classes, can be verified by a simple construction of a "universal" language from the machines accepting languages in the corresponding complexity class. To construct the universal NP complete language, observe that there exists a nondeterministic machine N_u which can simulate any other N_i with only a polynomial increase in the computation time; let $p_i(\ )$ be this polynomial. Then

$$U_{NP} = \{N_i \# x \#^k | N_u \text{ simulating } N_i \text{ on } x \text{ accepts in } k \text{ steps}\}$$

is an NP complete language.

To see this, note that U_{NP} is accepted by a nondeterministic multitape TM in linear time, thus U_{NP} is in NP. Furthermore, for each N_i running in time q_i the polynomial time reduction:

$$x \to N_i \# x \#^{p_i \circ q_i(|x|)}$$

is a reduction of $L(N)$ to U_{NP}.

The first natural NP complete problem was the Boolean formula satisfiability problem:

$SAT = \{F | F \text{ is a satisfiable Boolean formula in conjunctive normal form.}\}$

It can be shown that we can restrict the CNF formulas to three terms per clause and still have an NP complete problem.

The formula given below is in SAT, but the general difficulty of finding satisfying assignments for such formulas can be partially surmised from this example:

$$F = (x_1 + x_2 + x_3)(\overline{x}_2 + \overline{x}_3 + \overline{x}_4)(x_1 + \overline{x}_4 + \overline{x}_5)(x_2 + \overline{x}_3 + x_5)(x_1 + \overline{x}_3 + x_4) \times (\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(x_3 + x_4 + x_5)(\overline{x}_1 + x_2 + \overline{x}_5).$$

THEOREM. *SAT is NP complete.*

OUTLINE OF PROOF. Clearly, SAT is in NP; a NP machine can guess a variable assignment, evaluate the formula and accept if F the assignment satisfies F.

To show that SAT is NP complete, we have to show that for all A in NP,

$$A \leq_m^P SAT.$$

That is, for any N running time $p(n)$ we must exhibit a polynomial time algorithm that maps x onto a Boolean formula in CNF, F_x, such that

$$x \in A \Leftrightarrow F_x \in SAT.$$

To indicate how this is done, let N be a one-tape nondeterministic TM which halts on all x, $|x| = n$, in $p(n)$ steps. Each step of a computation on x is described by the instantaneous description, ID, which gives the content of the first $p(n)$ tape cells and marks the scanned cell with the state of the machine (we view this state and tape symbol of this cell as a combined symbol). A computation of N on x is represented by the sequence of $p(n)+1$ $ID's$ each of length $p(n)$:

$$ID_0, ID_1, ID_2, \ldots, IP_{p(n)}.$$

The key idea of the proof is that we can write a set of clauses which can be satisfied iff N accepts x. The variables in these clauses will specify what symbol appears in each tape cell for each ID. They will insure that ID_0 is the starting ID on input x, that $ID_{p(n)}$ contains an accepting state, and that for all i, $0 \leq i \leq p(n) - 1$, ID_{i+1} follows from ID_i by a legal move of N.

The following variables are used to construct F_x. For each symbol s that can appear in the computation (i.e., a tape symbol, if the cell of the ID is not currently scanned and a combined tape symbol-state if the cell is currently scanned by N) and for each cell of each ID we introduce a variable $u_{s,i,t}$. The variable $u_{s,i,t}$ is 1 iff in ID_i cell t contains the symbol s.

Now by a tedious but straightforward method, we can construct F_x. First, we must insure that each cell contains exactly one symbol. This can be achieved by asserting that each cell contains a symbol, i.e., for all i and t, $0 \leq i, t \leq p(n)$,

$$\bigvee_{s \in S} u_{s,i,t}$$

must be 1. To insure that the cell contains a unique symbol we must have that for all i and t:

$$\neg \bigvee_{s \neq r} (u_{s,i,t} \wedge u_{r,i,t})$$

is 1. The product of these expressions is satisfiable only if that each cell contains a unique tape symbol.

Similarly, we can write out conditions that ID_0 contains x in the first n cells, with q_0 marked on the first tape cell, and blanks in the remaining cells, i.e., .

$$u_{s_0,0,0} \wedge u_{s_1,0,1} \wedge u_{s_2,0,2} \wedge \ldots \wedge u_{s_{p(n)},0,p(n)},$$

where s_0 denotes q_0 and the first symbol of x, s_1 the second symbol of x, etc.

To express the condition that each ID_i, $1 \leq i \leq p(n)$ follows from the previous one by a legal move of N, we observe that a move of N can only affect the previously scanned cell and the cells immediately to its right and left. Thus, the possible content of cell t in ID_{i+1} is determined by the content of cells $t-1$, t and $t+1$ of ID_i, $0 \leq i \leq p(n) - 1$. All other cells remain unchanged. Thus, either $u_{s,t,i} = u_{s,t,i+1}$ or $u_{s,t,i+1}$ is changed by a legal move

of N. Let $legal(s_1, s_2, s_3, s)$ designate all legal moves of N. Then the formula

$$\bigwedge_{0 \le i,t \le p(n)} \left(\bigvee_{legal(s_1,s_2,s_3,s)} (u_{s_1,i,t-1} \wedge u_{s_2,i,t} \wedge u_{s_3,i,t} \wedge u_{s,i+1,t}) \right)$$

expresses the condition that all ID_{i+1} follow properly from ID_i.

Finally, we can easily express the last requirement that $ID_{p(n)}$ contains an accepting state.

The product of these formulas is satisfiable iff there exists an accepting computation of N on x. Thus, x is accepted iff F_x is satisfiable. Furthermore, it can be seen that F_x has length polynomial in length of x and can be computed in polynomial time.

With more formula juggling, we can transform F_x into conjunctive normal form and then in a CNF with no more than three terms per clause.

This completes the outline of the proof. □

As mentioned earlier, Cook's proof that SAT is NP complete opened the floodgate to proofs that hundreds of different problems are NP complete. Almost all proofs of NP completeness show that a problem is in NP and that SAT (or some other well known NP complete problem) can be reduced to it. A proof that a problem is NP complete is now considered strong evidence that the problem is not feasibly solvable in its full generality. Thus, such proofs direct attention to study of special cases and to approximations to the problem. It is interesting to note that some NP complete problems have good P-time approximations and that others do not [GJ].

Today, it is clear that NP completeness is a fundamental concept and that the $P = ?NP$ question is one of the most important open problems in computer science and possibly in all of mathematics. Clearly, as it will be seen from these lectures, we seek not only a *yes* or *no* answer to the $P = ?NP$ question, but a general understanding of the structure of the feasible computational complexity classes, of which the $P = ?NP$ problem is the best known. We believe that the study of the structure of feasible computations is one of the most important and interesting areas of computer science, with relevance to a broad area of intellectual endeavors.

The importance of the $P = ?NP$ question is further enhanced when one realizes that this is a fundamental question about the computational complexity of doing mathematics. We outline this relationship informally below.

Let F be a reasonably rich, sound, axiomatizable formal system in which the validity of a given proof can be checked in polynomial time in the length of the theorem plus proof. For example, Pressburger Arithmetic, Peano Arithmetic, and Zermelo-Frankel Set Theory, properly encoded, are of this type (assuming their soundness). Then, the set of theorems provable in F with an indicated bound on the length of the proof, is an NP complete set.

More explicitly, the following set is NP complete:

$T_F = \{Theorem$: "Statement".

$Proof$: $\#^k\square|$ The "statement" has a proof in F of length $\leq k\}$.

It is interesting to note that all three formal systems mentioned above yield NP complete sets in this manner, though Pressburger Arithmetic is a decidable theory and the other two are not decidable. We will show, in the next section, that these three NP complete sets are in a strong technical sense very similar to SAT.

4. Isomorphism. A careful inspection of the many NP complete problems discovered in the early seventies revealed great similarities among them and lead to the conjecture that they all are p-isomorphic. This conjecture was supported by a powerful lemma about padding-functions which easily showed that the known NP complete problems were p-isomorphic, as outlined below.

Two sets A and B, $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$, are *p-isomorphic* if there exists a bijection $f\colon \Sigma^* \to \Gamma^*$ such that

$$x \in A \Leftrightarrow f(x) \in B$$

and f and f^{-1} are polynomial time computable. We write:

$$A \cong_p B.$$

We now state the polynomial time analogue of the Cantor-Bernstein-Myhill Theorem [**BH**].

A function $f\colon \Sigma^* \to \Gamma^*$ is *length increasing* iff

$$(\forall x)[|f(x)| > |x|].$$

THEOREM. *If $A \subseteq \Sigma^*$ is p-reducible to $B \subseteq \Gamma^*$ and B is p-reducible to A by length increasing, p-time invertible injections, then*

$$A \cong_p B.$$

From this theorem we can derive [**BH**].

PADDING LEMMA. *An NP complete set A is p-isomorphic to SAT iff there exist two polynomial time computable functions S and D such that:*

$$(\forall x, y)[D(x,y) \in A \Leftrightarrow x \in A]$$

and

$$(\forall x, y)[S \circ D(x,y) = y].$$

Since all natural NP complete problems have padding and unpadding functions, this lemma shows that they all are p-isomorphic.

Thus, we conclude that the sets T_F (of theorems provable in F with indicated bound on the length of the proof) are p-isomorphic for the earlier mentioned formal systems (decidable and undecidable) and that they are p-isomorphic to SAT. This observation again emphasizes the close relation between NP complete problems and the complexity theorem proving.

As a matter of fact, the p-isomorphism between T_F and SAT can be shown to even preserve the number of solutions. If F has k satisfying assignments, then the corresponding theorem will have exactly k different proofs of the prescribed length.

One way of disproving the Isomorphism Conjecture would be by showing that there exist NP complete sets of sufficiently different densities such that the densities cannot be preserved under polynomial time bijections. To pursue this approach, we consider sets which contain only polynomially many elements up to size n.

A set S, $S \subseteq \Sigma^*$, is *sparse* if for some k and all $n \geq 1$,

$$|S \cap \Sigma^n| \leq n^k + k.$$

Clearly, a sparse set cannot be p-isomorphic to SAT, since SAT is too dense to be mapped by a bijection onto a sparse set.

The possibility of finding sparse NP complete sets was resolved by Mahaney's result [Mah].

THEOREM. *There exist sparse $\leq_m^P$-complete sets in NP iff $P = NP$.*

Though Mahaney's result eliminated one possibility (if $P \neq NP$) of disproving the Isomorphism Conjective, more recent work has cast doubts on its validity [JY]. Intuitively speaking, the structure of NP looks today far more intricate than it did in 1977 and there may indeed be more than one isomorphism degree of $\leq_m^P$-complete problems in NP. Should there be non-isomorphic $\leq_m^P$-complete problems in NP then we know that there are infinitely many pairwise non-isomorphic complete problems and their isomorphism degrees form a rich structure [MY].

We do not have many candidates for natural NP complete problems not isomorphic to SAT. One candidate is the set, defined by generalized Kolmogorov complexity, of strings which can be easily computed from shorter strings.

$$K[n/2, n^2] = \{x | (\exists y)[|y| \leq |x|/2 \text{ and } M_u(y) = x \text{ in } n^2 \text{ steps}\}.$$

This set is easily seen to be in NP, but it is not known to be NP complete. On the other hand, there are relativized worlds (i.e., worlds where all machines have access to a given oracle) in which the above set is NP complete and others in which it is not. Therefore, this again seems like a very difficult problem to decide, because of its contradictory relativizations.

Besides the many-one complete sets, complexity theory also studies Turing complete sets. A set S is *polynomial time Turing complete* (denoted by $\leq_T^P$-complete) if S is in NP and

$$NP \subseteq P^S.$$

The search for sparse $\leq_T^P$-complete sets for NP has yielded a rich set of results, some of which will be mentioned in the next section and are further discussed in a later chapter.

It should also be mentioned that if $P \neq NP$ then there exist incomplete sets in $NP - P$ [La].

THEOREM. *If $P \neq NP$, then there exists sets in $NP - P$ such that SAT cannot be $\leq_m^P$, reduced to them.*

It is interesting to note that if $P \neq NP$ then we cannot recursively enumerate a list of NP machines $N_{i_1}, N_{i_2}, \ldots$ such that for all j, $L(N_{i_j})$ is an incomplete language and every incomplete language is given by some $L(N_{i_k})$. On the other hand, the set of complete languages of NP can be named by a recursive list of NP machines. Furthermore, we can show that if $P \neq NP$ then there are languages in $NP - P$ for which we cannot prove that they are not in P. More precisely, let F be any sound, axiomatizable formal system, then $P \neq NP$ implies that there exists an A in $NP - P$ such that for no N_i, $L(N_i) = A$, is it provable in F that

$$L(N_i) \text{ is not in } P.$$

Furthermore, if we can prove in F that $P \neq NP$, then we know that the above A has to be an incomplete language.

5. The Polynomial Hierarchy and Related Classes. The Polynomial Time Hierarchy, PH, was introduced in analogy to the classic Kleene Hierarchy from recursive function theory to classify computational problems with a more complex logical structure than NP problems. For example, the set of Boolean formulas with a unique satisfying assignment or the set of Boolean formulas whose lexicographically least satisfying assignment starts with a 1 do not seem to be in NP, and can be easily captured with additional quantifiers, or in terms of oracle computations.

The Polynomial Time Hierarchy can be defined inductively as follows, using relativized computations [GJ, St].

$$\Sigma_0^P = \Pi_0^P = \Delta_0^P = P,$$
$$\Sigma_1^P = NP, \Pi_1^P = coNP, \Delta_1^P = P^{\Sigma_1^P},$$
$$\Sigma_{k+1}^P = NP^{\Sigma_k^P}, \Pi_{k+1}^P = co\Sigma_{k+1}^P, \Delta_{k+1} = P^{\Sigma_k^P}, k \geq 1.$$

Here, we take for a class C, $coC = \{L | \overline{L} \text{ in } C\}$. We define the polynomial hierarchy

$$PH = \bigcup_k \Sigma_k^P.$$

Equivalently, PH can be defined in terms of sequences of alternating polynonimally bounded quantifiers, as follows [Wr]. We recall that any L in NP can be expressed as

$$L = \{x | (\exists^P y) R(x, y)\},$$

where $R(\ ,\ )$ is a polynomial time computable predicate and $\exists^P y R(x, y)$ stands for: there exists a y, $|y| \leq p(|x|)$, such that $R(x, y)$ holds, where $p(\ )$ is a fixed