ON SOME CENTRAL PROBLEMS IN COMPUTATIONAL COMPLEXITY[+]

Janos Simon

TR 75-224

January 1975

Department of Computer Science
Cornell University
Ithaca, New York 14853

[+]This research was supported in part by the National Science Foundation Grant GJ-33171X and Grant 70/755 from Fundacao de Amparo a Pesquisa do Estado de Sao Paulo, and by Universidade Estadual de Campinas.

Janos Simon was born on June 11, 1946 in Budapest, Hungary. In 1956 he emigrated with his parents to Brazil. He received the B. Sc. degree in Physics in 1968 and his engineering degree (Electrical Engineering) in 1969, both from Universidade de Sao Paulo. In March 1970 he joined the Department of Computer Science of Universidade Estadual de Campinas, from where he was on a leave of absence while studying at Cornell University (Fall 1970 - December 1974). He was a student member of the ACM and vice president of the Cornell Student Chapter of the ACM.

Table of Contents

	Page
Biographical Sketch	ii
Acknowledgments	iii
Abstract	vii
Chapter 1: INTRODUCTION	1
Chapter 2: DEFINITIONS AND BACKGROUND	5
2.1 Introduction	5
2.2 Machine Models	5
2.3 Measures of Complexity	9
2.4 The Importance of Polynomial Classes	11
2.5 The Big Problems	16
2.6 Complete Problems	19
2.7 Other Known Results	30
Chapter 3: RANDOM ACCESS ACCEPTORS	44
3.1 Introduction	44
3.2 PTAPE $\supseteq$ NPTIME - NRAM	45
3.3 PTIME - NRAM $\supseteq$ NPTAPE	54
3.4 Comments	67
Chapter 4: ON NONDETERMINISM	72
4.1 Outline	72
4.2 What is Nondeterminism	73
4.3 Complete Problems, Reductions and Conservation Properties	78
4.4 Threshold Machines	88
4.5 Polynomial Bounded Hierarchies	98
4.6 Approximate Solutions	118

	Page
Chapter 5: CONCLUSIONS AND REMAINING PROBLEMS	126
Bibliography	133

Abstract

In this thesis we examine some of the central problems in the theory of computational complexity, like the trade-offs between time and memory, the power of nondeterminism and parallelism, and the speed gained by adding new operations to random access machines. Our main result is the characterization of the power of multiplication in random access acceptors: we show, in Chapter 3, that for such models nondeterministic and deterministic computations are polynomially related and that there is a polynomial relationship between the amount of time required for acceptance by random access machines with multiplication, and the amount of tape required by Turing machines. Thus, the additional power gained by using multiplication is the same as that of memory over time (if any). We derive similar results for some other interesting instruction sets.

We also have some results for probabilistic and nondeterministic computations: we define threshold machines and show how probabilistic Turing machines may simulate them, and exhibit a set of complete problems for threshold machines. For nondeterministic computations, we present a hierarchy of the elementary recursive languages obtained by polynomially bounded quantification over objects of higher and higher type, which represent nondeterministic time bounded computations with larger and larger bounds. Finally, we discuss some deterministic approximations to nondeterministic computations, and conclude with a look at some open problems.

Chapter 1 INTRODUCTION

One of the main tasks of theoretical computer science is the building of a rigorous mathematical theory of the complexity of computation. Powerful results, that establish the basis for such a theory, have been obtained [54], [27], [4], [5], [24], but some fundamental questions must be answered in order to ontinue this ambitious program. There is a fairly general consensus about what these fundamental problems are: the relationships between the different resources such as time or memory needed for computations; the advantages, if any, of using probabilistic, parallel or nondeterministic methods; the power of programming features in computers, such as multiplication or shifts; the conciseness gained by describing languages using more and more powerful syntactic features are all such questions. Is it harder to find a proof than to verify that an alleged proof is correct? If so, by how much? How difficult is it to discover whether a given number is prime? How much more difficult is it to ascertain that a problem has a given number of solutions than to verify that it has at least one solution? The list of such apparently hard and interesting questions seems extremely long. Recent research in the area has, however, identified a small number of "super problems", in terms of which almost all these problems can be formulated. Moreover, since these questions may be posed as asking for algorithms with certain properties for solving some concrete important problems, efficient solutions would also have a practical value. We shall justify the claims above in Chapter 2; for the moment we

just present a list of "super problems":

1) The quantitative differences between deterministic and nondeterministic computations.

2) The time vs. memory trade-off in computations.

3) The computational speed gained by adding new operations to random access machines.

4) The descriptive power gained by adding new features to language definition schemes.

5) The advantages, if any, of using parallel or probabilistic models of computation.

It is generally expected that cracking these problems will require a long and concerted effort by many researchers [28], [41], and this thesis is intended to be a contribution to this effort.

In Chapter 2 we present an extensive review of recent work, as well as notation for the rest of the thesis. Chapter 3 contains our main results: we attack question (3) in the form posed by Cobham, namely, "Is it harder to multiply than to add? If so, why?" and reduce it to question (2). That is, we show that random access acceptors with a multiplication operation are more powerful than acceptors with addition if and only if tape-bounded computations are more powerful than time-bounded computations. More precisely, the class of languages recognized by polynomially time-bounded random access machines with multiplication equals the class of languages recognized by polynomially memory-bounded random access machines without multiplication. We also show that for machines with

multiplication, question (1) can be answered: nondeterminism adds very little power to the model. Our techniques yield a new proof of Savitch's result [60], that for tape-bounded computations nondeterminism decreases the amount of tape needed by at most a polynomial factor. We also examine some other instruction sets and their computational power.

In Chapter 4 we examine nondeterminism and, to some extent, probabilistic and parallel models of computation. The character of this chapter is quite different from the previous one: while in Chapter 3 we reduce one hard problem to another,. Chapter 4 is the result of our unsuccessful attempt to solve questions (1) and (5). Although these problems remain wide open, we did uncover some interesting facts and small results that should be useful in further studies and, we hope, may help to give some insight to other researchers. First, by looking at different interpretations of what we mean by nondeterminism, we get some results for probabilistic machines. We prove that an extensive class of combinatorial problems is complete for such machines (see Chapter 2 for definitions), and we exhibit some properties of these problems. Using techniques of [47] we get a technical result: a new hierarchy of the elementary recursive sets, obtained by bounded quantification over objects of higher type. Also in Chapter 4 we comment on some recent work, trying to obtain approximate solutions to hard (P-complete) problems which become easy if nondeterministic computations are allowed. We argue (by exhibiting examples in which the behavior of the proposed approximations is pathological) that such work cannot answer our questions about the

power of nondeterminism and suggest a new definition of approximation, which is relevant to our investigations. The technical trick which runs through the chapter, unifying these results, is the observation that most reductions among combinatorial problems studied in the area are quite strong, in the sense that not only will one have a solution if and only if the other will, but the structure of the solution to one problem is reflected in the solution to the reduced problem.

Our work, as well as the many powerful results of Meyer, Stockmeyer and Hunt, had the effect of reducing the number of problems on our list. This is important, since it shows that these questions (which have remained unanswered for many years) are difficult because they are just a disguised form of some really fundamental problems. On the other hand, this just makes us more curious about the answers. In Chapter 5 we speculate briefly about how to obtain them, and list some open problems suggested by our work.

Chapter 2 DEFINITIONS AND BACKGROUND

2.1 Introduction

This chapter should give the reader all the definitions needed to understand this thesis. However, some knowledge of theoretical computer science is assumed (e.g. [29]). After defining machine models and measures of complexity we justify the intensive study of polynomially bounded classes of computations and present some major open problems in terms of these classes. Finally we review the most important known facts about the problems.

2.2 Machine models

Throughout this thesis we will be considering computing devices as acceptors of sets of sequences, or languages. Alternatively we may view these as 0-1 valued functions, with 1 standing for "accept". The reason we make this restriction is because we will want to eliminate the possible advantages of one model over some other that are due only to the fact that a big output takes a long time to write down in some model.

We expect that the reader is familiar with Turing machines (abbreviated "Tm" from here on) as consisting of a finite control (possibly nondeterministic) that controls the movement of a read-only head on an input tape and the actions of read/write heads on worktapes. Tapes are divided into squares, one square being scanned at any given time by one head. Each square may contain a symbol from a fixed finite alphabet.

Heads may move one square at a time. Acceptance occurs by entering some designated final state of the finite control. If the machine is nondeterministic, it is said to accept if there is a sequence of moves leading to a final state (see [1 Chapter 7], [27]).

Random access machines (RAMs) were introduced as a more realistic model for computers. (See [1] Chapter 2, or [10] for motivation.) We define now a whole series of different RAM models.

A RAM acceptor or RAM with instruction set O is a set of registers $R_0, R_1, \ldots$ each capable of storing a non-negative integer in binary representation, together with a finite program of (possibly labeled) O-instructions. If no two labels are the same, we say that the program is deterministic, otherwise it is nondeterministic. We call a RAM model deterministic if we consider only deterministic programs from the instruction set.

Our first instruction set consists of the following:

O_1

$R_i \leftarrow R_j \quad (=k)$	(assignment)
$R_i \leftarrow \langle R_j \rangle$	(indirect addressing)
$R_i \leftarrow R_j + R_k$	(sum)
$R_i \leftarrow R_j \dot{-} R_k$	(proper subtraction)
$R_i \leftarrow R_j$ bool R_k	(boolean operations)
if R_i comp R_j label 1 else label 2	(conditional jump)
accept	
reject	

comp may be any of $<$, $\leq$, $=$, $\geq$, $>$, $\neq$. For boolean operations we consider the integers as bit strings and do the operations componentwise. Leading 0's are dropped at the end of operations: for example, 11 nand 10 = 1. bool may be any binary boolean operation (e.g. $\wedge$, $\vee$, eor, nand, $\supset$, etc.) accept and reject have obvious meanings. An operand of =k is a literal and the constant k itself should be used.

The computation of a RAM is started by putting the input in register R_0, setting all registers to 0 and executing the first instruction of the RAM's program. Instructions are executed in sequence until a conditional jump is encountered, after which one of the instructions with label "label 1" is executed if the condition is satisfied and one of the instructions with label "label 2" is executed otherwise. Execution stops when an accept or reject instruction is met. A string $x \in \{0,1\}^*$ is accepted by the RAM if there is a finite computation ending with the execution of an accept instruction. The sequence of instructions executed in such a way is called an accepting sequence.

O_0

is O_1 without the boolean operations. We shall call RAM's with these instruction sets RAM_1's (or simply RAMs) and RAM_0's, respectively.

O_2

is O_1 plus the instruction

$R_i \leftarrow R_j \circ R_k$ (concatenation)

which leaves in R_i the contents of R_j followed by the contents of R_k. Again, the operands may be literals. We shall call RAMs with instruction set O_2 CRAMs (C for concatenation).

O_3

is O_1 plus the instruction

$R_i \leftarrow R_j \cdot R_k$ (product)

which computes the product of the two operands (which may be literals) and stores it in R_i. RAMs with instruction set O_3 will be called MRAMs (M for multiplication).

O_4

is O_3 plus the instruction

$R_i \leftarrow R_j \div R_k$ (integer division)

which leaves in register R_i the result of dividing R_j by R_k and taking the integer part of the result. If R_k contains 0 the machine jams and rejects. These RAMs will be called PRAMs (P for powerful).

Finally, we describe VRAMs, defined in [53]. This model is quite different from the previous ones. There are two different kinds of registers: shift registers and general (vector) registers. The only interaction between the two is by means of the shift instructions

$V_i \uparrow I_k$ (shift right)

$V_i \downarrow I_k$ (shift left)

which shift the contents of general register V_i to the right or left by the amount contained in shift register I_k. For

shift registers we have the instructions of assignment, sum, proper subtraction and division by 2; for general registers we have only boolean operations. In addition, we have conditional jumps using the result of a comparison between two general registers or between two index registers to decide which label to jump to. Literals and indirect addressing may be used in all operations.

2.3 Measures of complexity

The two important resources that we want to measure are time and memory. For an accepting computation of a Tm we define time as the number of moves in an accepting computation and memory (space or tape) as the maximum number of squares used on any work tape. For nondeterministic machines we take the accepting computation with the minimum value of the quantities measured. The complexity measures defined for RAMs are:

(unit) time measure: the complexity of an accepting computation is the number of instructions executed in the accepting sequence. The complexity of the RAM on input x is the minimal complexity of accepting computations.

logarithmic, or length time measure: the complexity of an accepting computation is the sum of the lengths of the operands of the instructions executed in the accepting sequence. When there are two operands, we take the length of the longer; when an operand has length 0 we use 1 in the sum. The complexity of the RAM on input x is the minimal complexity among accepting computations.

memory measure: the maximum number of bits used at any time in the computation. (The number os bits used at a given time is the sum of the number of significant bits in all registers in use at that time.)

Unless otherwise stated, time measure will mean unit time measure.

We shall denote the class of languages accepted by deterministic Tms within time (tape) f(n) by TIME[f(n) (TAPE[f(n)]). We shall prefix notations by N if we want the measure to refer to nondeterministic computations. When we want to emphasize that the class is deterministic we may use the prefix D. Thus NTAPE[n] denotes the context-sensitive languages, DTAPE[n] or simply TAPE[n] the class of deterministic context-sensitive languages. In addition, we will often use PTIME or P for the class of languages accepted in deterministic polynomial time by Tms, and NPTIME or NP for the class of languages accepted in nondeterministic polynomial time by Tms and PTAPE for the corresponding tape class.

When we want our bounds to refer to RAMs, we shall use the name of the machine as a suffix. For example, DTIME[n^k]-MRAM denotes the class of languages accepted in deterministic time n^k by MRAMs, and

$$\text{NPTIME-CRAM} = \bigcup_{k=0}^{\infty} \text{NTIME}[n^k]\text{-CRAM}$$

The notation length-TIME[f(n)]-RAM will denote the languages accepted by RAMs within the bound f(n) in the logarithmic time measure.

Finally we will occasionally use the notations csℓ = NTAPE[n], dcsℓ = DTAPE[n], DLOG-TAPE = DTAPE[logn] and NLOG-TAPE = NTAPE[logn] which do not conform to the conventions above but are in general use.

We shall discuss relationships among these measures after discussing polynomial classes.

2.4 The importance of polynomial classes.

We have given special names to polynomially bounded classes. This is because they are relevant to the question, "What is practically computable?".

From the early research on effective computability a consensus emerged during the first half of this century about the precise meaning of the concept "effectively computable". This consensus is expressed in Church's thesis which in essence asserts that: a function is effectively computable (or simply computable) if and only if there exists a Turing machine which computes it.

Clearly, the class of effectively computable functions contains functions which are practically computable as well as functions which require arbitrarily large amounts of computing time (or any other resource by which we measure computational complexity) and thus these functions cannot be practically computed. So far there has not emerged any real consensus as to which functions are practically computable. It is also not clear whether the concept of practically computable functions is in any sense fundamental and whether it has a mathematical invariance comparable to the class of effectively computable

functions.

At the same time, there is already a general agreement that a process whose computation time (on a Turing machine) cannot be asymptotically bounded by a polynomial in the length of the input data is not a practical computation. For example, any function for which every Turing machine computing it requires at least a number of steps exponential in the length of the input is not practically computable, since no matter what the speed of the device which realizes the Tm, for sufficiently long inputs the time spent in computing will be prohibitive: if we reduce the time of every operation by a multiplicative factor we only get an additive increase in the number of feasible inputs.

Convention: A computation is feasible iff it runs in polynomial time (in the length of the input) on a deterministic Turing machine, i.e. if it is in P (PTIME).

It is easily seen that the class of feasible computations, PTIME, includes a wide variety of languages and solutions of problems and that it is quite invariant under changes of computing models. In particular it is well known that P = PTIME - RAM_0 = length-PTIME - RAM_0 (see [1 Chapter 1] for a proof). It is also true that P = PTIME - RAM. This may be proved easily for the unit time measure by noting that one can compute a boolean function of two bit arguments on a RAM_0 and a boolean function of a bit vector in time proportional to the length of the operands. Since the latter may increase at most by one per operation, the result follows.

By using the same idea, we have that

P = length - PTIME - RAM

= length - PTIME - CRAM

= length - PTIME - MRAM

= length - PTIME - VRAM.

Historical note: traditional RAM models do not have boolean operators. One may argue however that the reason they were left out of earlier RAM definitions (where + was an operator) was mainly because RAMs were mostly used to represent von Neumann computers working on numerical problems. All real computers have such capabilities, so if RAMs are to be a more or less realistic model of them, they should have boolean operations. As we saw, the two models are polynomially related. The idea of examining the power of boolean operators is due to [53].

On the other hand, there are very many other problems and languages about which we do not know whether they are feasible, that is, no polynomial time bounded algorithms have been discovered for them, nor has it been shown that such algorithms do not exist. Many of these problems are of considerable practical importance, and substantial effort has been expended to find deterministic polynomial time bounded algorithms for them.

A wide class of such important practical problems has the property that they can be computed in polynomial time if the computations can be nondeterministic. Remember that a nondeterministic Tm may have several possible transitions from a given state, and it accepts a string w if there is a sequence of moves, starting with the initial configuration with input w and

ending with a configuration in which the finite control is in a final state. The amount of resource used to accept w is the minimum over all such accepting sequences.

Thus a nondeterministic Turing machine can guess a solution and then verify whether it has guessed correctly. For example, consider the set

$L = \{w \mid w \in 1(0 \cup 1)^*$ and w does not denote a prime number$\}$.

It is not known whether L is in PTIME but it is easily seen that L can be accepted in polynomial time by a nondeterministic Tm which guesses an integer (a binary sequence in $1(0 \cup 1)^*$ not longer than w) and then tests deterministically whether the integer divides w.

To give another, more theoretical computer science oriented example, let R_i and R_j be regular expressions over the alphabet consisting of 0, 1, $\cdot$, $\cup$ and the delineators (,) and let $L(R_i)$ denote the set of sequences designated by R_i. Since we have not permitted the use of the Kleene star, *, in the regular expressions, we see that we can only describe finite sets and that the longest string in the set cannot exceed the length of the expression. Furthermore, the language

$$L_R = \{R_i, R_j \mid L(R_i) \neq L(R_j)\}$$

can easily be recognized in polynomial time by a nondetermin s-tic Tm which guesses a binary sequence w whose length does not exceed the length of the longest expression R_i or R_j, and then verifies that w is in $L(R_i)$ or $L(R_j)$ but not in both. So far no deterministic polynomial time algorithm has been discovered for this problem.

Thus, each of these problems is in NP, but it is not known whether it is in P. To emphasize the importance of NP, we list a series of important problems in NP. In all cases, we assume that the problem is coded by a string, and that the coding is straightforward and reasonably economical.

1. Given R_i, R_j regular expressions over $0,1,\cdot,\cup,(\ ,\ )$. Determine if the sets of sequences denoted by R_i and R_j are different, i.e. if $L(R_i) \neq L(R_j)$.

2. Given a formula of the porpositional calculus (or, more specifically, a Boolean expression in conjunctive normal form), determine if it is "true" for some assignment of the values "false" and "true" to its variables.

3. Given a (directed) graph G determine if the graph has a (directed) cycle which includes all nodes of G.

4. Given a graph G and integer k determine if G has k mutually adjacent nodes.

5. Given an integer matrix C and integer vector d determine if there exists a 0-1 vector x such that $Cx = d$.

6. Given a family of sets and a positive integer k. Determine if this family of sets contains k mutually disjoint sets.

7. Given a (n+1)tuple of integers $(a_1,a_2,\ldots,a_n,b)$. Does there exist a 0-1 vector x such that $\Sigma a_i x_i = b$?

It is easy to see that all these problems are in NP. Finally, we shall consider the polynomial tape bounded classes PTAPE and NPTAPE. There are many interesting problems in those classes too, for example

1') Given R_i, R_j regular expressions over $0,1,\cdot,\cup,*,(\ ,\ )$.

Determine if the sets of sequence denoted by R_i and R_j are different, i.e. if $L(R_i) \neq L(R_j)$.

At this point one might object that although many of these problems are important, it seems arbitrary to choose polynomial bounds instead of, say, n^5. A good case can be made for requiring closure under composition: in programming language terms, if a program is feasible, substituting single operations by feasible programs (subroutines) should result in a feasible program. Moreover, changes in the encoding of the program or in the computing device which seem intuitively harmless should not cause the class to be modified. In this sense changing a number from binary to decimal notation is a harmless change, but using unary notation is not. Polynomial classes seem to be the smallest classes satisfying these requirements and containing many interesting problems. On the other hand, if we are willing to consider these classes we do not have to make distinctions among models for which complexities differ only by a polynomial amount. We shall throughout this thesis use such "polynomial smearing" language: for example we shall say that Tms and RAMs have the same power (running time) as acceptors.

2.5 The big problems

In the previous section we have shown many examples of interesting problems, which can be solved within polynomial bound where the bound is on something other than deterministic time (viz. nondeterministic time in the case of problems 1-7, and tape in the case 1'). The list of such problems can be

extended (see [38], also Chapter 4 of this thesis) and the question of whether feasible algorithms exist for solving these problems is very important. That is,

is P = NP

and/or

P = PTAPE?

These questions are the formalized version of the first two of the fundamental questions listed in the introduction.

Intuitively, we feel that P $\neq$ NP, though all attempts to prove it have failed. On the other hand, the exciting thing is that if P = NP then its proof is very likely to reveal something fundamentally new about the nature of computing.

To emphasize this fact, recall that prime numbers have been studied for over two thousand years without discovering a fast (i.e. deterministic polynomial time) algorithm for their testing. Since the set of binary strings representing primes is in NP, this is just one more instance of the P = NP problem [52]. As an illustration we recall that in 1903 F. Cole showed that

$$2^{67} - 1 = 193707721 \times 76183825287$$

and claimed that it had taken him "three years of Sundays" to show that $2^{67} - 1$ was not a prime, as conjectured before. It is also striking how easily one can check whether the given factorization is correct, thus dramatically illustrating the difference between "finding a solution" and "verifying its correctness", which is the essence of the P = NP problem. For related problems see [22], [23].

Similarly, it is not known what the relationship is between time and memory. Clearly, a Turing machine operating in

polynomial time can visit only a polynomially bounded number of different tape squares and therefore we have

$P \subseteq PTAPE$ and $NP \subseteq NPTAPE$.

Furthermore, any nondeterministic Tm M_i operating in time n^k can make no more than n^k different choices. On polynomial tape a deterministic Tm can successively enumerate all possible q^{n^k} sequences of choices M_i can make and for each sequence of choices simulate deterministically on polynomial tape the corresponding M_i computation. Therefore we obtain

Proposition 2.1: $NP \subseteq PTAPE$.

We may present all the easy inclusions as

Corollary: $P \subseteq NP \subseteq PTAPE \subseteq NPTAPE$.

On the other hand, it is not known whether there exists a language in PTAPE which is not in NP. Intuitively, one feels that there must be such languages since in polynomial tape a Turing machine can perform an exponential number of operations before halting. At the same time, nobody has been able to prove that this exponential number of operations, restricted to polynomial tape, can be utilized effectively to accept something not in NP.

As a matter of fact, our ignorance about such trade-offs is even greater: while a counting argument similar to the above yields that $P \supseteq DLOG - TAPE$, it is not known whether the inclusion is proper, i.e. it is possible that $P = DLOG - TAPE$.

As we will show below, to prove that P = NP we do not have

to show that every problem in NP has an equivalent solution in P. All we have to show is that any one of the seven previously listed problems 1-7 has a deterministic polynomially time bounded algorithm. Similarly, if problem 1' is in P (NP) then P = PTAPE. The existence of such "complete" languages, which seems to simplify the problems, was a result of the research of S.A. Cook [9], and we dedicate the next section to these results.

Just as the first two questions of our list can be formalized in terms of polynomial classes,

"Is P (=PTIME - RAM) = PTIME - MRAM?"

formalizes our question 3. We shall show in Chapter 3 that it can be reduced to,

"Is P = PTAPE?"

i.e.,

"Is memory more powerful than time?"

Finally, questions about parallel and probabilistic machines will be tackled in Chapter 4, where we present all the relevant models and definitions.

2.6 Complete problems

A central idea in recursive function theory is the notion of reducibility. We say that a set (i.e. the membership problem of the set) A is reducible to a set B via reduction process r, and write $A \leq_r B$ if the membership problem of A may be solved by a process of type r and a knowledge of B. We may then define C to be complete with respect to r, in a class S if

i) $(\forall B)$: $B \in S$, $B \leq_r C$

ii) $C \in S$.

In recursive function theory, when these ideas were first used, S was the collection of r.e. sets, and the set

$K = \{\langle x,y\rangle \mid$ the x-th Tm, T_x, halts on input y$\}$

was proven to be complete with respect to many reducibilities. [55] In a sense, a complete set represents a whole class and is the "hardest" element of that class, "to within the complexity of r". Instead of formalizing these comments in a general setting, we shall first examine the case S = NP, $A \leq_r B$ iff there is $f \in P$ with $x \in A$ iff $f(x) \in B$. In complexity theory, the approach was first used by S. Cook [9] and the reducibility defined here is due to Karp [38]. Thus we define:

Definition: A language L is NP-complete (or complete in NP) iff L is in NP and for all L_i in NP there exists a function f_i, computable by a deterministic Tm in polynomial time, such that

w is in L_i iff $f_i(w)$ is in L.

Proposition 2.2: If L is a NP-complete language then L is in P iff P = NP.

Proof: To see this note that P = NP implies that L is in P. On the other hand, if L is in P then there exists a deterministic Tm, M, which in polynomial time accepts L. For any other L_i in NP there exists, by definition of NP-completeness, a deterministic Tm M_i' which computes a function f_i such that

$w \in L_i$ iff $f_i(w) \in L$.

Let $M_{D(i)}$ be the deterministic Tm which on input w applies M_i' to compute $f_i(w)$ and then applies M on $f_i(w)$ to test whether

$f_i(w)$ is in L. Clearly, the deterministic Tm $M_{D(i)}$ accepts L_i and operates in polynomial time since M_i' and M do. Thus L_i is in P, which completes the proof.

Next we show that NP-complete languages actually exist by constructing a "universal NP" language L_U. This language is somewhat artificial but it reveals very clearly why NP-complete problems exist and demonstrates a technique which has many other applications. After this proof we show that there are also "natural" NP-complete languages; as a matter of fact, all the previously listed problems 1-7 are NP-complete.

Theorem 2.1: There exist NP-complete languages.

Proof: We will show that L_U defined below is NP-complete. Let

$$L_U = \{\#M_i\#\text{CODE}(x_1x_2\ldots x_n)\#^{3|M_i|t} \mid x_1x_2\ldots x_n \text{ is accepted by the one-tape, nondeterministic Tm } M_i \text{ in time } t\}$$

where M_i is given in some simple quintuple form, $|M_i|$ designates the length of the representation of M_i, and $\text{CODE}(x_1x_2\ldots x_n)$ is a fixed, straightforward, symbol by symbol encoding of sequences over alphabets of arbitrary cardinality (the input and tape alphabet of M_i) into a fixed alphabet, say $\{0,1,\#\}$; with the provision that $|\text{CODE}(x_j)| \geq$ cardinality of the tape alphabet of M_i.

It is easily seen that a four-tape nondeterministic Tm M' can accept L_U in linear time. We indicate how M' uses its tapes: on the first sweep of the input M' checks the format of the input, copies M_i from the input on the first working tape and $\#^{3|M_i|t}$ on the second working tape. The third working

tape is used to record the present state of M_i. (in a tally notation) during the step-by-step simulation of M_i. It is seen that with the available information on its working tapes M' can simulate M_i on the input in time $2|M_i|t$ (for an appropriate, agreed upon representation of M_i). Thus M' operates in nondeterministic linear time and accepts L_U. Therefore, L_U is in NP and the assumption P = NP implies that L_U is accepted by a deterministic Tm M' operating in deterministic time n^p. Then for any nondeterministic Tm M_i working in time n^q we can recursively construct a Tm $M_{\sigma(i)}$ operating in polynomial time as follows:

1. for input $x_1x_2\ldots x_n$ $M_{\sigma(i)}$ writes down $\#M_i\#\text{CODE}(x_1x_2\ldots x_n)\#^{3|M_i|n^q}$
2. $M_{\sigma(i)}$ starts the deterministic machine M' on the sequence in (1) and accepts the input $x_1x_2\ldots x_n$ iff M' accepts its input.

Clearly, M_i and $M_{\sigma(i)}$ are equivalent; furthermore $M_{\sigma(i)}$ operates in time less than

$$2[3|M_i|n^q + |\#M_i\#\text{CODE}(x_1x_2\ldots x_n)|]^p \leq Cn^{pq}.$$

Thus $M_{\sigma(i)}$ operates in deterministic polynomial time, as was to be shown.

Next we assert that there exist very many natural NP-complete problems and that finding fast algorithms for some of them is of considerable practical importance.

We will prove that

$$L_R = \{(R_i,R_j) \mid R_i,R_j \text{ are regular expressions over } 0,1,\cdot,U,(\ ,\ ) \text{ and } L(R_i) \neq L(R_j)\}$$

is NP-complete. We have chosen to use this language since the proof utilizes a technique of describing Turing machine computations by means of regular expressions and this technique has interesting further applications. The main idea is due to L. Stockmeyer and A. Meyer [47], and the technique was used to prove many powerful theorems about the expressive power of language features [47],[48],[65],[30],[21],[31],[32],[34].

Theorem 2.2: L_R is NP-complete and so are all the languages associated with the problems 1-7.

Proof: We prove only that the first problem on our list is NP-complete; for the other proofs see [38].

The proof that L_R is NP-complete relies heavily on the fact (proven below) that regular expressions can be used to describe the "invalid computations" of nondeterministic Tms. More explicitly, for every nondeterministic Tm M_i operating in polynomial time there exists a deterministic Tm which for input $x_1x_2\ldots x_n$ in polynomial time writes out a regular expression describing the invalid computations of M_i on input $x_1x_2\ldots x_n$. Note that the input $x_1x_2\ldots x_n$ is accepted by M_i iff there exists a valid M_i computation on this input, which happens iff the set of invalid M_i computations on $x_1x_2\ldots x_n$ is not the set of all sequences (of a given length). Thus the test of whether $x_1x_2\ldots x_n$ is accepted by M_i can be reduced to a test of whether the regular expression, describing the invalid computations of M_i on $x_1x_2\ldots x_n$, does not describe all sequences (of a given length). This implies that if L_R is in P then P = NP and thus we see that L_R is NP-complete.

We now give the above outlined proof in more detail. Let M_i be a one-tape, nondeterministic Tm which operates in time n^k, and keeps repeating its accepting configuration. This version of Tm is slightly different from our definition, but it is more convenient and equally powerful. Let S be the set of states of M, q_0 the unique starting state, q_f the unique accepting final state, and let Σ be the tape alphabet of M_i. An instantaneous description of M_i is a sequence in $\Sigma^*(\Sigma \times S)\Sigma^*$ which indicates the tape content of M_i, the state M_i is in, and which symbol is being scanned. $VCOMP(x_1x_2\ldots x_n)$ denotes the set of valid computations of M_i on input $x_1x_2\ldots x_n$. A valid computation consists of a sequence of instanteous descriptions

$$\#ID_1\#ID_2\#ID_3\#\ldots\#ID_{n^k}\#$$

such that

a) $ID_1 = (x_1,q_0)x_2x_3\ldots x_n b^{n^k-n}$

b) $ID_{n^k} \in \Sigma^t(\Sigma \times \{q_f\})\Sigma^{n^k-t-1}$

c) ID_{j+1} follows from ID_j, $1 \leq j \leq k-1$ by one move of M_i.

Note that for all i, $|ID_i| = n^k$.

Define Γ as

$$\Gamma = \Sigma \cup \Sigma \times S \cup \{\#\}$$

and let the invalid set of computations be given by

$$NVCOMP(x_1x_2\ldots x_n) = \Gamma^{n^{2k}+n^k+1} - VCOMP(x_1x_2\ldots x_n).$$

We show next that there exists a deterministic Tm, M_D, which constructs for every input $x_1x_2\ldots x_n$ in polynomial time a regular expression using only $\cdot, \cup$, denoting the set of sequences in NVCOMP. To see this note that NVCOMP consists of:

R_1 = set of sequences which do not start correctly

R_2 = set of sequences which do not end correctly

R_3 = set of sequences which do not have a proper transition from ID_j to ID_{j+1}.

Thus

$$NVCOMP(x_1x_2\ldots x_n) = R_1 \cup R_2 \cup R_3.$$

Let $\bar{x}$ designate $\{y,\ y \neq x,\ y \in \Gamma\}$. Note that the regular expression for $\bar{x}$ has length of the order of the size of Γ, i.e. a constant. Then

$$R_1 = \bar{\#}\Gamma^{n^{2k}+n^k} \cup \Gamma(\overline{x_1,q_0})\Gamma^{n^{2k}+n^k-1} \cup \Gamma^2\bar{x}_2\Gamma^{n^{2k}+n^k-2}+\ldots \cup \Gamma^{n^{2k}+n^k}\bar{\#}$$

$$R_2 = \{\Gamma - (S \times \{q_f\})\}^{n^{2k}+n^k+1}$$

$$R_3 = \bigcup_{p=0}^{n^{2k}-3} [\bigcup_{\sigma_1\sigma_2\sigma_3 \in \Gamma} \Gamma^p\sigma_1\sigma_2\sigma_3\Gamma^{n^k-1}[\Gamma^3 - CORRECT(\sigma_1\sigma_2\sigma_3)]\Gamma^{n^{2k}-p-4}]$$

where CORRECT$(\sigma_1\sigma_2\sigma_3)$ is the set of correct M_i transitions in one move from $\sigma_1\sigma_2\sigma_3$, in the following instantaneous description.

These triples are sufficient to specify the transitions, since in a single move only the square being scanned may be modified and the only other possible change in the ID is the position of the read/write head. Since the head moves as most one square, the set CORRECT suffices to characterize valid transitions. For any given Tm CORRECT is a finite set depending only on the alphabet and on the transition rules of the machine, but not on the input.

A straightforward computation shows that $R_1 \cup R_2 \cup R_3$

can be written out for input $x_1x_2\ldots x_n$ by a deterministic Tm in polynomial time in n. Thus the desired M_D exists.

If L_R is in P then we have a deterministic Tm M_C which in polynomial time accepts (R_i, R_j) provided $L(R_i) \neq L(R_j)$. But then combining M_D with this M_C we get a deterministic polynomial time Tm which for input $x_1x_2\ldots x_n$ writes out (using M_D)

NVCOMP$(x_1x_2\ldots x_n)$

and then checks (using M_C) whether the expressions R_i and R_j are unequal. Clearly, the regular expressions are nequal iff there is a valid computation of M_i on $x_1x_2\ldots x_n$, but that happens iff M_i accepts this input. Since M_D and M_C operate in deterministic polynomial time, the combined machine accepting $L(M_i)$ also operates in deterministic polynomial time. Thus L_R in P implies P = NP. Clearly, P = NP implies that L_R is in P since it is easily seen that L_R is in NP. This completes the proof.

The properties of complete languages proven for NP-complete languages hold for many other interesting classes. For example, the analogue in the case of polynomial tape is

Definition: We say that a language L in PTAPE is tape complete iff for every L_i in PTAPE there exists a deterministic polynomial time computable function f_i such that

$w \in L_i$ iff $f_i(w) \in L$.

From this we immediately get

Proposition 2.3: NP = PTAPE (P = PTAPE) iff there exists a tape complete problem in NP (P).

We now show that tape complete problems exist. In order

to emphasize the similarities with the NP case, we consider the following two languages:

$$L_{UT} = \{\#M_i\#CODE(x_1x_2\ldots x_n)\#^t \mid M_i \text{ accepts } x_1x_2\ldots x_n \text{ using no more than } 2+|M_i|+CODE(x_1x_2\ldots x_n)+t \text{ tape squares}\}$$

$$L_R^* = \{R_i \mid R_i \text{ is a regular expression over } 0,1,\cdot,\cup,*,(\ ,\ ) \text{ and } L(R_i) \neq (0 \cup 1)^*\}$$

Theorem 2.3: L_{UT} and L_R^* are tape complete languages. Thus NP = PTAPE (P = PTAPE) iff $L_{UT} \in$ NP (P) iff $L_R^* \in$ NP (P).

Proof: It is easily seen that L_{UT} is accepted on the amount of tape needed to write down the input, if we permit nondeterministic operations. Thus L_{UT} is in PTAPE. Furthermore, if L_i is in PTAPE then there exists a deterministic Tm M_i which accepts L_i in n^k tape, for some k. But then there exists a Tm, $M_{\sigma(i)}$, which for input $x_1x_2\ldots x_n$ writes

$$\#M_i\#CODE(x_1x_2\ldots x_n)\#^{n^k-|CODE(x_1x_2\ldots x_n)|-2-|M_i|}$$

on its tape in deterministic polynomial time. Designate the function computed by $M_{\sigma(i)}$ by f_i. Then w is in L_i iff $f_i(w)$ is in L_{UT}, and we see that L_{UT} is tape complete.

To prove that L_R^* is tape complete observe that a nondeterministic Tm can guess a sequence and then on linear amount of tape (using standard techniques [44]) check that the sequence is not in R_i, since the length of the shortest word not in L_R^* must be bounded by the size of R. Thus L_R^* is in PTAPE; as a matter of fact L_R^* is a context-sensitive language, as is L_{UT}. Now we again will exploit the power of regular expressions to describe invalid Tm computations efficiently.

For a Tm M_i which operates on n^k tape, we define

$$VCOMP(x_1x_2\ldots x_n) = \#ID_1\#ID_2\#\ldots\#ID_{HALT}\#$$

as we did earlier , with $|ID_j| = n^k$. Since now we have the Kleene star available in our regular expressions, we define

$$NVCOMP = \Gamma^* - VCOMP.$$

Thus

$$NVCOMP(x_1x_2\ldots x_n) = R_1 \cup R_2 \cup R_3 ,$$

where R_1, R_2 and R_3 represent the sets of strings which do not start right, which do not end right, and where there is an incorrect transition from ID_i to ID_{i+1}, respectively. The details are quite similar to the previous proof and we write down the expressions for R_1, R_2 and R_3 to indicate the use of the *, which was not available in the other proof. It should be pointed out that M_i could perform an exponential number of steps before halting and therefore we cannot use the techniques of the previous proof. This proof makes an essential use of the Kleene star. Again $\bar{x}$ denotes any y, $y \neq x$ and y in $\Gamma = \Sigma \cup \{\#\} \cup \Sigma \times S$, q_0 the initial state and q_f the final state.

$$R_1 = \{\bar{\#} \cup \#[(x_1,q_0) \cup (x_1,q_0)[\bar{x}_2 \cup x_2[\bar{x}_3 \cup x_3[\ldots \cup \bar{\#}]\ldots]\Gamma^*$$

$$R_2 = (\Gamma - \Sigma q_f)^*$$

$$R_3 = \Gamma^*\sigma_1\sigma_2\sigma_3\Gamma^{n^k-1}[\Gamma^3 - CORRECT(\sigma_1\sigma_2\sigma_3)]\Gamma^*$$

where $CORRECT(\sigma_1\sigma_2\sigma_3)$ is a correct sequence of the next ID if in the previous ID in the corresponding place appears $\sigma_1\sigma_2\sigma_3$.

It is easily seen that a deterministic Tm exists which for input $x_1x_2\ldots x_n$ writes out the regular expression $R_1 \cup R_2 \cup R_3$ on its tape in deterministic polynomial time. If we denote this

function by f_i we see that

w is accepted by M_i iff $f_i(w) = R_1 \cup R_2 \cup R_3 \neq \Gamma^*$

since w is accepted by M_i iff there exists a valid computation of M_i on w, but this happens iff NVCOMP(w) $\neq \Gamma^*$. This concludes the proof that L_{UT} and L_R^* are tape complete languages and we see that NP = PTAPE (P = PTAPE) iff L_R^* or L_{UT} is in NP (P).

Let us now return to the abstract formulation. Suppose that a reducibility $\leq_r$ was defined by

$A \leq_r B \leftrightarrow x \in A$ if $f(x) \in B$

where f is in some resource-bound class C. Suppose moreover that we have two families of languages L_1 and L_2 such that

a) $L_2 \supseteq L_1$

b) L_1 is closed under functions in C

c) there is a universal language $\mathcal{L}$ for L_2, $\mathcal{L} \in L_2$, i.e. for all $\mathcal{L}_a \in L_2$, there is an $f \in C$ such that $x \in \mathcal{L}_a$ iff $f(x) \in \mathcal{L}$.

If these conditions are all met, then

$L_1 = L_2$ iff $\mathcal{L} \in L_1$.

The reader should find it easy to verify that these conditions were met in our previous theorems about complete languages, and that these were actually the only conditions used in the proofs.

The proof is identical to the one we presented for NP. There are many classes which satisfy the conditions above: we saw this to be true of NP and PTAPE, but there are complete languages for exponential time, for P (under logarithmic tape

bound reducibility[37]), for context sensitive languages [21], etc. Such results are clearly very powerful, since they allow us to decide equivalence of two classes by settling the membership problem of a specific language in a class. The existence of such universal languages is a very nice algebraic property that may be very important in the theory of computational complexity. They are also very useful in AFL theory, complexity of parsing, etc. We shall present more about complete languages in the next section.

It is also possible to define $A \leq_r B$ by oracle machines (this was the original approach of Cook). It is not known whether the two reducibilities agree on NP. For a discussion of reducibilities and their comparative power, see [42].

2.7 Other known results

First of all, we present a result which adds considerable strength to the fact that complete languages exist. We will first examine the case of NP vs. P, but again the argument is quite general.

Consider the language L_U of the previous section. We saw that if L_U is in P then we can recursively obtain for every M_i running in time n^q an equivalent deterministic Tm running in time $O[n^{pq}]$. Unfortunately, for a given Tm we cannot recursively determine the running time and thus we do not know whether M_i runs in polynomial time or not. Even if we know that M_i runs in polynomial time we still cannot recursively determine the degree of the polynomial. Nevertheless, we have

Theorem 2.4: P = NP iff there exists a recursive translation σ and a positive integer k, such that for every nondeterministic Tm M_i, which uses time $T_i(n) \geq n$, $M_{\sigma(i)}$ is an equivalent deterministic Tm working in time $O[T_i(n)^k]$.

Proof: The "if" part of the proof is obvious. To prove the "only if" part assume that P = NP. We will outline a proof that we can recursively construct for any M_i, running time $T_i(n) \geq n$, an equivalent deterministic Tm $M_{\sigma(i)}$ operating in time $O[T_i(n)^k]$ for a fixed k.

In our construction we use two auxiliary languages:

$B_i' = \{\#w\#^t \mid M_i \text{ accepts } w \text{ in less than } t \text{ time}\}$

$B_i'' = \{\#w\#^t \mid M_i \text{ on input } w \text{ takes more than } t \text{ time}\}$.

Clearly, the languages B_i' and $\Sigma^* - B_i''$ can be accepted in nondeterministic linear time. Therefore, by our previous result, we can recursively construct two deterministic machines M_i' and M_i''' which accept B_i' and $\Sigma^* - B_i''$, respectively, and operate in time $O[n^p]$. Now we obtain M_i'' from M_i''': M_i'' recognizes B_i''. M_i'' is a deterministic polynomial time bounded Tm since if $\Sigma^* - B_i''$ is in NP, it is in P (by hypothesis), and P is closed under complements.

From M_i' and M_i'' we can recursively construct the deterministic Tm $M_{\sigma(i)}$, which operates as follows:

1. For input w, $M_{\sigma(i)}$ finds the smallest t_0 such that $\#w\#^{t_0}$ is not in B_i''. This is done by checking with M_i'' successively $\#w\#$, $\#w\#^2$, $\#w\#^3$, ...

2. $M_{\sigma(i)}$ starts M_i' on input $\#w\#^{t_0}$ and accepts w iff M_i' accepts $\#w\#^{t_0}$.

Clearly, $M_{\sigma(i)}$ is equivalent to M_i and $M_{\sigma(i)}$ operates in time

$$O[\sum_{\ell=1}^{T_i(n)} \ell^p] = O[T_i(n)^{p+1}]$$

By setting $k = p+1$, we have completed the proof.

The theorem is due to Hartmanis and Hunt [21]. The "translation arguments" used in these proofs, although known for some time [57] can be used to derive many interesting (and easy) results of the form, "If certain equivalences hold -- say P = NP -- then some other equivalences, for bigger complexity bounds, must also hold." The trick is to suffix a "padding" to the input of the language recognized within the higher bound so that it becomes computable within the smaller one. As an illustration of the technique we present a minor theorem.

Theorem 2.5: Suppose NP = DLOG-TAPE. Then

a) NLOG-TAPE = DLOG-TAPE

b) for all tape constructible bounds $L(n) \geq \log n$,

DTAPE[L(n)] = NTAPE[L(n)]

c) P = NP

Proof: NLOG-TAPE $\subseteq$ P by the results in [11]. Thus, DLOG-TAPE $\supseteq$ NP (by hypothesis) $\supseteq$ P (trivially) $\supseteq$ NLOG-TAPE which proves a). To prove b), suppose $A \in$ NTAPE[L(n)], $L(n) > \log n$. Consider the language

$$A' = \{x\#^t \mid x \in A,\ t = 2^{L(|x|)}\}$$

a new symbol

Clearly, $A' \in$ NLOG-TAPE, and therefore, by hypothesis,

A' $\in$ DLOG-TAPE. Consider the deterministic Tm M' that accepts A' within that bound: its worktapes will be L(n) bound, but its read head may advance into the #'s. To recognize A we build a deterministic L(n) bound Tm that mimics the actions of M' as long as M' does not attempt to read past x. If this happens we simulate the input # and, on a special counter, keep track of the position of the read head. The length of this track need not be more than L(n), so that the whole procedure works within bound L(n). This proves b).

Finally, P $\supseteq$ DLOG-TAPE (by counting argument)

DLOG-TAPE $\supseteq$ NP (by hypothesis)

so

P = NP.

Many such results can be proven. For example in [6], Book proves that NP $\neq$ csl (although all other possibilities remain open). For related results, see [7].

This sort of algebraic technique might give us some insight into the hard problems. For example, it is not known whether the language

L_p = {w| w $\in$ 1(0 $\cup$ 1)* and w designates a composite number}

is an NP-complete problem. It is easily seen that L_p is in NP, as stated before. A somewhat harder proof shows that, surprisingly,

$\tilde{L}_p$ = {w| w $\in$ 1(0 $\cup$ 1)* and w designates a prime}

is also in NP [52]. Thus the "guess and verify" method can be used to design (nondeterministic) polynomial time algorithms

to test whether an integer is or is not a prime. At the same time, translation arguments show that if L is an NP-complete language then $\Sigma^* - L$, the complement of L, is a complete language for the class

$$\text{co-NP} = \{L \mid \Sigma^* - L \text{ is in NP}\}.$$

It is easy to prove, analogously to Proposition 2.2, that NP = co-NP iff a co-NP-complete language is in NP. Therefore, if L_p were a complete language (for NP), NP would be closed under complements. This last fact would be quite unexpected. At a very intuitive level: if we consider the set of unequal regular expressions over $0,1,\cdot,\cup,(\ ,\ )$:

$$L_R = \{(R_i,R_j) \mid L(R_i) \neq L(R_j)\},$$

which is in NP (in fact is NP-complete), then it seems impossible (with our current state of knowledge about computing) that this computation could be carried out in deterministic polynomial time. Similarly, it seems impossible that the set of pairs of equal regular expressions,

$$L_{R\neg} = \{(R_i,R_j) \mid L(R_i) = L(R_j)\},$$

could be in NP, since in this case we would have to give a proof in nondeterministic polynomial time that there does not exist any sequence on which R_i and R_j differ. This appears to be a completely different situation than the proof that L_R is in NP and we do not know any methods which can exploit the power of nondeterminism to yield such a proof.

Of course, P is closed under complement, so that a proof that NP is not could automatically yield that $P \neq NP$. A proof of this sort could possibly show that $P \neq NP$ without giving any

insight into the actual deterministic time complexity of the class NP. On the other hand, it is conceivable that P $\neq$ NP, but NP is closed under complementation.

The power of nondeterminism in tape-bounded computations is much better understood. Savitch [60] proved the only non-trivial relationship about nondeterministic and deterministic classes:

Theorem 2.6: For all tape-constructable functions $L(n) \geq \log n$, $NTAPE[L(n)] \subseteq DTAPE[L(n))^2]$.

Proof (sketch): The instantaneous descriptions of an L(n) tape bounded Tm M may be written in tape cL(n). (For $L(n) < n$ represent the position of the input-reading head by a counter of length logn.) We may assume, wlog, that there is a unique final ID and that the final ID keeps repeating itself once it is reached, which must occur (at least for one choice of moves) within time $2^{dL(n)}$. Then recognize deterministically the same language as M, within tape $L(n)^2$ with the following recursive method:

generate the initial ID of M on x

generate the final accepting ID

to test whether it is possible for M to have a sequence of moves of length $\ell = 2^{dL(n)}$ leading from the first ID to the second,

if there is a sequence of length 1,

then accept (return success)

else generate in order all possible configurations c of length L(n) (they will be candidates for midpoints

of such a sequence). For each configuration c, test (using the same procedure recursively) if it is possible to reach c from the first configuration, and if so, whether it is possible to have a sequence of moves that takes M from c to the second ID.

For sequences of length 1 the test is trivial (just check M's possible moves). Each activation of the recursive procedure uses O(L(n)) tape (since only one configuration c is stored at a time) and at most L(n) levels of recursion are simultaneously active.

Corollary: PTAPE = NPTAPE.

In the next chapter we provide an independent proof of this result.

Finally, we look briefly at the expressive power of language features, our question 4. We saw that

$$L_R = \{(R_i, R_j) \mid R_i, R_j \text{ regular expressions over } 0,1,\cup,\cdot,(\ ,\ ) \text{ and } L(R_i) \neq L(R_j)\}$$

is an NP-complete problem and if we added the expressive power of the Kleene star, *, the language

$$L_R^* = \{(R_i, R_j) \mid R_i, R_j \text{ regular expressions over } 0,1,\cup,\cdot,*,(\ ,\ ) \text{ and } L(R_i) \neq L(R_j)\}$$

became a tape complete language. Though we cannot prove that NP ≠ PTAPE, we conjecture that they are different, and therefore the Kleene star made the decision problem harder by the difference, if any, between NP and PTAPE.

It should be noted that without the Kleene star we cannot describe an infinite regular set and with $\cdot, \cup, *$ all regular sets

can be described. From this alone we would suspect that the decision problem (recognition of) L_R^* should be harder than for L_R. Whether it really is harder, and by how much, remains a fascinating and annoying open problem.

Next we take a look at how the computational complexity of the decision problem for regular expressions changes as we permit further operations. We know that all regular sets can be described by regular expressions using the operators $\cup, \cdot, *$. At the same time, regular sets are closed under set intersection and set complementation. Therefore we can augment our set of operators used in regular expressions by $\cap$ and $\neg$, and we know from experience that these two operators can significantly siplify the writing of regular expressions. The surprising thing is that it is possible to prove that the addition of these operators makes the decision problems about regular expressions much harder [65]. In particular, the addition of the complementation operator makes the decision problem for equivalence of regular expressions practically unsolvable [65], [48], [32].

Theorem 2.7: The language

$$\tilde{L}_R = \{(R_i, R_j) \mid R_i, R_j \text{ regular expressions over } 0,1,\cdot,\cup,*, \neg, (\ ,\ ) \text{ and } L(R_i) \neq L(R_j)\}$$

cannot be recognized for any k in $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$ tape. In other words, $\tilde{L}_R$ cannot be recognized on tape bounded by an elementary function.

The basic idea of the proof is very simple: if we can show that using extended regular expressions (i.e. with $\cdot, \cup, *, \neg$) we can describe the valid computations of Tms using very

large amounts of tape by very short regular expressions that are easily obtained from the Tm and its input, then the recognition of $\tilde{L}_R$ must require very large amounts of tape. To see this note that to test whether w is in $L(M_i)$ we can either run M_i on w, using whatever tape M_i requires, or else write down the regular expression, R = VCOMP(w), describing the valid computations of M_i on w, and then test whether $L(R) = \emptyset$. Since w is accepted by M_i iff there exists a valid M_i computation in w, if the expression R is very short and the recognition of $\tilde{L}_R$ does not require much tape then this last procedure would save us a lot of tape. This is impossible, since there exist languages whose recognition requires arbitrarily large amounts of tape and these requirements cannot be (essentially) decreased [64,29]. Thus either method of testing whether w is in $L(M_i)$ must require a large amount of tape; which implies that the recognition of $\tilde{L}_R$ must require a large amount of tape.

For the sake of completeness we will mention a result [31] about regular expressions without $\neg$, but with $\cup,\cdot,^*$ and $\cap$.

Theorem 2.8: Let

$$L_R^{\wedge} = \{(R_i,R_j) \mid (R_i,R_j \text{ regular expressions over } 0,1,\cup,\cdot,^*, \cap,(\ ,\ ) \text{ and } L(R_i) \neq L(R_j)\}$$

Then the recognition of $L_R^{\wedge}$ requires tape $L(n) \geq 2^{\sqrt{n}}$.

From the two previous results we see that the additional operator in regular expressions added considerably to the descriptive power of these expressions, in that the added operators permitted us to shorten regular expressions over $\cup,\cdot,^*$ and that this shortening of the regular expressions is reflected in the

resulting difficulty of recognizing $\tilde{L}_R$ and $L_R^{\wedge}$, respectively. Thus, in a sense, these results can also be viewed as quantitative results about the descriptive powers of regular expressions with different operators.

The approach, and most of the proofs, are due to Meyer and Stockmeyer (See Stockmeyer's thesis [65] for a complete treatment) and Hunt. In the proofs the complexity of the recognition of unequal pairs of regular expressions is established by the following method: one describes valid or invalid Tm computations by regular expressions and then relates the efficiency of describing long Tm computations by short regular expressions to the complexity of the decision problem; the more powerful the descriptive power of our expressions (or languages) the harder the corresponding decision problem. We can actually state this somewhat more precisely as:

Heuristic Principle: If in some formalism one can use expressions of length n or less to describe Tm computations using tape up to length L(n), then the decision procedure for equality of these expressions must be of at least tape complexity L(n).

For example, if a formalism enables us to state that a Tm accepts an input of length n using tape at most 2^n, and the length of such an expression is n^2, then any procedure that decides equality of two such expressions will have tape complexity at least $2^{\sqrt{n}}$. In [31] it is shown that regular expressions over $0,1,\cup,\cdot,\cap,^*$ are such a formalism -- from which Theorem 2.8 follows.

Clearly, this principle also implies that if the formalism

is so powerful that no computable function L(n) can bound the length of tape used in Tm computations which can be described by expressions of length n, then the equivalence problem in this formalism is recursively undecidable. Thus this principle gives a nice view of how the expressive power of languages escalates the complexity of decision procedures until the equivalence problem becomes undecidable because the length of the Tm computations is no longer recursively bound to the length of the expressions describing them. Thus a formalism in which we can say ,"the i-th Tm halts", so that the length of this formula grows recursively in i, must have an undecidable decision problem. Very loosely speaking, as long as we can make assertions about Tm computations in our formalism, without describing the computations explicitly, we will have undecidable decision problems. As long as we must describe the Tm computations explicitly, the equivalence problem will be soluble and its computational difficulty depends on the descriptive power of the formalism.

This principle has yielded many superexponential lower bounds for logical theories [45,46,13] and exponential or worse lower bounds for formal language problems [33,34,65] and is one of the most important tools in this area of research. Although we will not use it explicitly, no background of the results in the field would have been complete without mentioning it.

There is very little known about probabilistic or parallel machines in our context, except for a paper by Gill [9]. We shall present the relevant facts and definitions in the text.

We conclude this section with another observation. Suppose we have a logical theory Th, axiomatized by a set Ax of axioms. Suppose moreover that the theory is "hard" and there is a theorem T which is provable, but the length ℓ of its proof is horrendously long. Consider the theory obtained by adding $\neg T$ to the axioms: this is an inconsistent. But, if Th is sufficiently powerful, the shortest proof that it is inconsistent will be horrendously long (otherwise we could have proven T by assuming $\neg T$ and obtaining a contradiction). So, in a sense, impossibly hard theorems are uninteresting.

Note that part of the importance of NP is that it is the class of families of short proofs: i.e. if we assume something that we could not prove in deterministic polynomial time, the proof that our assumption was false could be very short. As an application, we prove that although, as we saw, equivalence of regular expressions is a hard problem, membership is always easy. We use the notation $\gamma(x)$, x a regular expression, that may contain γ's, to denote the set

$$\{y \mid |z| = |y| \text{ , } z \text{ denoted by } x\} .$$

Theorem 2.9: Let R be a regular expression on $(\Sigma, \cup, \cdot, \cap, \neg, *, p)$ and $x \in \Sigma^*$. The language

$$\mathcal{L} = \{(R,x) \mid x \in \mathcal{L}(r)\}$$

is in PTIME.

Proof: Let $S = \{y \mid y \text{ is a substring of } x\}$.

$|S| = 1 + n \cdot 1 + (n-1) \cdot 2 + \ldots \leq n^3$, $n = |x|$.

Let $R = \{r \mid r \text{ a regular expression, subexpression of } R\}$.

$|R| \leq \ell^3$, $\ell = |R|$.

We build the |R| by |S| matrix A. The (i,j) component is 1 iff the j-th substring of x is in the language of the i-th subexpression in R. Clearly, once we have the matrix, we shall in particular know whether $x \in \mathcal{L}(R)$. We proceed by induction of the complexity of the subexpression.

1) (Base) r is atomic -- for simplicity let $\Sigma = \{0,1\}$. Clearly, it is trivial to decide for $y \in$ Substr(x) whether $y = 0, 1$, or ε, the only possible cases. So we may fill out the rows of A that correspond to atomic r. (time required $\leq \ell^3 n^3$)

2) Suppose α, β are in R and their rows in A are known. We produce the row of A corresponding to δ.

a) $\delta = \alpha \cup \beta$.

Put a 1 in every column that α or β has a 1 in.

(time required $\leq 3n^3$)

b) $\delta = \alpha \cap \beta$

Put a 1 in every column that both α and β are 1 in.

(time required $\leq 3n^3$

c) $\delta = \neg\alpha$.

Put a 1 in every column that α has a 0 in.

(time required $\leq 2n^3$)

d) $\delta = \alpha \cdot \beta$.

For every string y_1 in $\mathcal{L}(\alpha)$ and every string y_2 in (β) put $y_1 \cdot y_2$ in δ.

(time required $\leq cn^3 \cdot n^3 = cn^6$)

e) $\delta = \gamma(\alpha)$.

For every string y in $\mathcal{L}(\alpha)$, add all z: $|z| = |y|$ to $\mathcal{L}(\delta)$. (time required $\leq n^3 \cdot n^3 = n^6$)

Each such operation increases the length of the regular expression by at least 1. It may need a lookup to see whether the expression was already computed, which takes time $\leq \ell^3$. Therefore, time $\leq c \cdot n^6 \cdot \ell^3 \cdot \ell = c\ell^4 n^6$.

To deal with *, expand it by

$$\alpha^* = (\emptyset + \alpha \cdot (\emptyset + \alpha \cdot (\emptyset + \alpha \cdot (\ldots)))))$$

up to level n. This may expand the length of R to ℓn. Note that the bounds are unnecessarily sloppy. For example, the lookup can be avoided by using the structure of R to store the matrix in a convenient form. We just want to show that $\in$ p. If we produce only the objects we need to build up to R, we shall not use more than $\ell' = \ell \cdot n$ expressions (because of *), but not ℓ'^3. (Base) will take $\sim \ell' n^3$ time and the lookup takes only time $\sim \ell'$, so that the upper bound is no more than

$$c \cdot \ell' \cdot n^6 \cdot \ell' = c\ell^2 n^8$$

but this may be improved, probably. Since $\ell, n < |y|$,

$$\{y \mid y = (R,x) \text{ and } x \in \alpha(R)\} \in \text{DTIME}[n^{10}].$$

Thus, in the case of the non-empty regular expressions for which this is hard to prove (non-elementary), the length of the first string in the set must be very long.

Chapter 3 RANDOM ACCESS ACCEPTORS

"The subject ... is perhaps most directly indicated by simply asking two questions: first is it harder to multiply than to add? and second, why?"

A. Cobham

3.1 Introduction

The quote above is from a 1964 paper [8], which shows that interest in the power of different RAM instruction sets is a long-standing one. We show that for a wide range of instruction sets

1) nondeterministic and deterministic polynomial time bounded classes are identical.

2) polynomial time bounded languages accepted by such RAMs are exactly the polynomial tape bounded languages.

Such models include MRAMs, CRAMs, PRAMs and VRAMs (see Chapter 2 for definitions). Note that for ordinary RAMs time is polynomially related to Tm time, both for deterministic and nondeterministic models (i.e. PTIME - RAM = P and NPTIME - RAM = NP).

The contents of this chapter are the following: in the next section, we prove

Theorem 3.1: PTAPE $\supseteq$ NPTIME - MRAM.

while in section 3.3 we prove that

Theorem 3.2: PTIME - CRAM $\supseteq$ NPTAPE.

It is easy to prove, and we do it in section 3.3, that

PTIME - MRAM $\supseteq$ PTIME - CRAM.

These inclusions show

PTIME - MRAM $\supseteq$ PTIME - CRAM $\supseteq$ NPTAPE $\supseteq$ PTAPE $\supseteq$ NPTIME - MRAM $\supseteq$ PTIME - MRAM,

from which we get

Corollary: a) PTIME - MRAM = NPTIME - MRAM, which is our result 1) and should be compared with the situation for RAMs without multiplication, where

PTIME - RAM = NPTIME - RAM iff P = NP.

b) PTIME - MRAM = PTAPE, which is our result 2); for machines without multiplication,

PTIME - RAM = P.

c) PTAPE = NPTAPE, which was known [60], but our proof is independent of Savitch's.

Fact b) may be interpreted as a characterization of the power of multiplication, i.e. the power added by the availability of multiplication is equivalent to the computaticnal power which polynomially tape-bounded computations have over polynomially time-bounded computations (if any).

In section 3.3 we also show that all the "powerful" RAMs--MRAMs, CRAMs, PRAMs, VRAMs -- have polynomially related running times. In section 3.4 we comment about the results and give a small example of how restrictions on the power of PRAMs give rise to "traditional" complexity classes. (cf. [25], [26], [63])

3.2 PTAPE $\supseteq$ NPTIME - MRAM

In this section we prove our main theorem, the simulation

of MRAMs working in polynomial time by Turing machines using polynomial tape. We shall not attempt a very efficient simulation, but we try to make the construction as clear as possible.

Suppose the MRAM M operates in time n^k, where n is the length of the input. Our Tm simulator T will write out in one of its tapes a guess for the sequence of operations executed by M in its accepting computation and check that the sequence is correct. The sequence may be written down deterministically, by enumerating all such sequences of length n^k in alphabetical order. Since the number of instructions of M's program is a constant, the sequence will be of length cn^k for some constant c. To verify that such a sequence is indeed an accepting computation of M we need to check that one step follows from the next when M's program is executed -- which is only a problem in the case of conditional instructions, when we must find out the contents of a register. We shall define a function FIND(r,b,t) which will return the value of the b-th bit of register r at time t. Our theorem will be proved if this function is computable in polynomial tape -- the subject of the remainder of this section. Note that since we are testing for an accepting sequence, it does not matter whether we are simulating deterministic or nondeterministic machines.

First, let us prove that the arguments of FIND may be written down in polynomial tape. Note that in t operations the biggest possible number that may be generated is a^{2^t}, produced by successive multiplications: a, a^2, $a^2 \cdot a^2 = a^4$, $a^4 \cdot a^4 = a^8$,... a^{2^t}, where a is the maximum of x and the biggest literal in M's

program. To address a bit of it, we need to count up to its lnegth, that is, up to $\log_2(a^{2^t}) = 2^t \log_2 a$, which may be done in space $\log_2(2 \log_2 a)$. In particular, for $t = n^k$, space n^{k+1} will suffice, so that $\underline{b}$ may be written down in polynomial tape.

Clearly, $\underline{t}$ may also be written down in polynomial tape.

There is a small difficulty with $\underline{r}$: since we allow indirect addressing, although in time n^k at most n^k registers are accessed, the address of a register may be as high as 2^{2^t}, which has length 2^t and cannot, therefore, be written in polynomial tape. However, at the cost of at most a square factor in time, we may restrict an MRAM operating in time $\underline{t}$ to use only its first t registers:

Let M' be an arbitrary MRAM. M" will mimic M' but only use its fifst 2t registers. M" uses its registers in pairs: the first component of the register pair holds an address, the address of a register of M'; the second component has the actual contents of that register. When M" has to simulate a move of M' which accesses register $\underline{s}$, M" first determines whether a first component holding $\underline{s}$ exists among the first t register pairs of M". If so, M" accesses the second component of that pair. Otherwise, M" creates a new pair in the first two available locations by storing $\underline{s}$ in the first component (register) and using the second for the $\underline{s}$-th register of M'. Clearly the simulation of a move of M' takes at most ct steps for some constant c, so that M" qperates in time ct^2. It uses only its first 2t registers.

We shall suppose that M uses only its first n^k registers.

We have shown that in that case all arguments of FIND may be written down in polynomial space.

Now let us describe FIND and prove that it operates in polynomial tape.

Informally, FIND works as follows: FIND(r,b,0) is easily computed given the input. We shall argue inductively. FIND(r,b,t) will be computed from the previous values of FIND--clearly the only interesting case is when $\underline{r}$ was altered in the previous move. For example, if the move at t-1 was r ← pVs, then FIND(r,b,t) = FIND(p,b,t-1) V FIND(s,b,t-1). This recursion in time does not cause any problems, because we may first compute FIND(p,b,t-1) and then reuse the tape for a call of FIND(s,b,t-1), so that if ℓ_{t-1} is the amount of tape needed to compute FINDs for times up to t-1, we have the recurrence

$$\ell_t = \ell_{t-1} + c \quad (\ell_0 = cn^{k+1})$$

which has the solution

$$\ell_t = c'n^{k+1}.$$

In the case of shift machines, studied in [53](VRAMs), this is the only recursion necessary to carry out such a simulation. However, with our machines, in the case of multiplication of two ℓ-digit numbers, we may have to compute up to ℓ factors and get the carry from the previous column in order to obtain the desired bit. Since ℓ may be 2^{n^k}, we must be able to take advantage of the regularity of operations in order to be able to compute within polynomial tape. Also, the carry from the previous column may be quite big: in the worst case, when we multiply $(1)^{\ell}$ by $(1)^{\ell}$ the carry may be ℓ. This is still manageable, since in time n^k, $\ell \leq 2^{n^k}$; an accumulator of length n^k

will suffice. We also need to generate up to ℓ pairs of bits, multiply them in pairs and add them up. This may be done as follows: we store the addresses of the two bits being computed, compute each of the two bits of the product separately, multiply the two results and update the addresses to get the addresses of the two bits of the next product. The product is added to an accumulator and the process is repeated until all product terms have been computed. Then we need the carry from the previous column.

We cannot compute this carry by a recursive call of FIND, because since the length of the register may be exponential, keeping track of the recursion would take exponential tape. Instead, we compute the carries explicitly from the bottom up -- i.e., we first compute the carry at the rightmost column (finding the bits by recursive calls of FIND on pairs and multiplying them), and then, with that carry and FIND, we compute the carry from the second rightmost column, and so on. The space needed is only for keeping track of which column we are at, one recursive call of FIND, one accumulator and one previous carry holder. Each of these may be written down in space n^{k+1}, so that we have the recursion

$$\ell_t = \ell_{t-1} + cn^{k+1}$$

which implies

$$\ell_t \leq cn^{2k+1}$$

and the simulation of $\cdot$ may be carried out in polynomial space.

The argument for + is similar, but much easier since only 2 bits and a carry of at most 1 are involved.

A bastard PL/I (PL/B ?) of FIND follows:

```
FIND:  PROCEDURE(r,b,t), returns (digit)
       /* We omit the trivial code for t = 0 */
       /* We suppose FIND has access to global variables
          that specify M's action at all times */
       if instruction at time t-1 not of the form q ← p
          op s then return (FIND(r,b,t-1)) fi
       if r ≠ q then return (FIND(r,b,t-1))
          /* register was not modified at time t-1 */
       else
          if op = boolean operation
             then /* compute relevant bits from operands */
                  BIT 1 = FIND(p,b,t-1)
                  BIT 2 = FIND(s,b,t-1)
                  return (BIT 1 op BIT2)
             else
             if op = ·
                then /* loop through columns until current
                        one is reached */
                     COLUMN = 0
                     CARRY = 0
                     while COLUMN ≤ b do
                       FIRSTPTR = 0 /* addresses of bits to
                       SECONDPTR = COLUMN    be multiplied */
                       ACUM = 0
                       while SECONDPTR ≥ 0 do /* add up
                                products in ACUM */
```

```
                BIT 1 = FIND(p,FIRSTPTR,t-1)
                BIT 2 = FIND(s,SECONDPTR,t-1)
                ACUM = ACUM + BIT1   BIT2
                FIRSTPTR = FIRSTPTR + 1
                SECONDPTR = SECONDPTR - 1
                        end
            ACUM = ACUM + CARRY /* get total
                                   sum in column */
            CARRY = if ACUM > 0 then (ACUM - 1)/2
                    else 0 /* shift right by 1 */
                       end
          return (ACUM mod 2)
else /* op = + */
     /* compute carries from right to left,
        as for · */
     COLUMN = 0
     CARRY = 0
     while COLUMN ≤ b do
         ACUM = 0
         BIT 1 = FIND(p,COLUMN,t-1)
         BIT 2 = FIND(s,COLUMN,t-1)
         ACUM = BIT 1 + BIT 2 + CARRY
         CARRY = if ACUM > 0 then (ACUM - 1)/2
                 else 0
                    end
     return (ACUM mod 2)
fi
                                  end: FIND.
```

Let us analyze the tape requirements of FIND.

All the inputs are representable in tape n^{k+1}. Moreover, no loop control variable exceeds an input variable; the same is true of FIRSTPTR and SECONDPTR. Also, we saw that the greatest possible carry is of order 2^{n^k}, and therefore representable in tape n^k. Therefore all variables are representable in space cn^{k+1} in any activation of FIND. The only possible problem arises with the recursion: however note that we have only one active call at a time in every activation of FIND and it has its t parameter smaller by one than the t of its calling routine. Thus, at most n^k activations of FIND may be present at any given time, and since each of them occupies at most cn^{k+1} squares of tape, the whole procedure works in space cn^{2k+1}.

This ends the proof of our theorem.

The features of FIND that carry the proof through are:
1) the possibility of computing the relevant digits of results of previous computations one at a time; even though there is an exponential number of them, the rule for their formation is easy.
2) the fact that the carry may be computed explicitly, in an orderly fashion from right to left. In this way, the only information needed from one column to the next is the carry from the previous column which, luckily, is just small enough to be representable in polynomial space.

For the benefit of the reader who got lost in the details: we have proven:

Theorem 3.1: Polynomial time bounded nondeterministic MRAM-recognizable languages are recognizable in polynomial tape by Turing machines.

In the next section we show the converse.

In the remainder of this section we extend the simulation to PRAMs. First note that a straightforward extension of the technique used to prove Theorem 3.1 fails: to compute the carries in a bit-by-bit simulation of the division algorithm we may need exponential space if all operands are exponentially long -- as would be the case if we tried to divide a number of length 2^n by one of length $2^{n/2}$. When either the quotient or the divisor are only polynomially long the usual division algorithm can be programmed to run within polynomial space, a fact which the reader is encouraged to verify. Thus, in particular there is no loss of generality in assuming that MRAMs have the instruction of division by 2. With this instruction set, we may simulate arbitrary division, using multiplication and shifts -- i.e. division by 2. We make use of an algorithm similar to Newton's root-finding technique which, given an n-bit integer p, computes $2^{2n-1}/p$ in O(logn) operations. This number is basically the reciprocal of p: to find [a/p] we find "1/p", multiply by a and shift by an appropriate amount. A shift corresponds to a division by 2, which is part of our instruction set. The computation of the reciprocal is done by a recursive technique: it is easy to get the first (most significant) digit of b = 1/p. At stage i, we have an approximation of b_i to b satisfying

$$b = b_i + (1/p)\ (1 - b_i p).$$

Using b_i as an approximation for $1/p$, we obtain the recursive formula

$$b_{i+1} = b_i + b_i(1 - b_i p).$$

Note that the method converges quadratically and may be programmed to yield $1/p$ to $2k$ bits from an approximation to k bits in a constant number of operations (Algorithm 8.1 in [1]).

To obtain the result $[a/p]$, we need to compute $1/p$ to an accuracy of $\log_2 a + 1$ digits. Since, as we saw, in t operations a is of order 2^{2^t}, we need to get $2^t + 1$ bits of $1/p$, which may be done in $O(t)$ operations. Thus an MRAM acceptor may simulate a PRAM acceptor with a loss of efficiency of at most a square factor.

3.3 PTIME - MRAM $\supseteq$ NPTAPE

This section starts with a collection of programming tricks. We give only an outline of the techniques used, hoping that the interested reader will be capable of filling in the details. Complete proofs (at least almost complete) for the VRAM case may be found in [53]. We shall use CRAMs in our constructions.

The idea of the proof is the following: given a Tm T operating in polynomial tape on input x, we first generate all possible configurations of this computation (a configuration of T on an input of length x consists of the state of the finite control, the contents of the worktape and the positions of T's heads.) We then obtain the matrix of the relation "follow in one move" -- i.e. if A is the matrix of the relation then

$a_{ij} = 1$ iff T passes from the i-th to the j-th configuration in one move. Clearly, x is accepted by T iff $a^*_{be} = 1$ where A^* is the transitive closure of A and b and e are initial and accepting final configurations respectively. To make matters simple we shall suppose without loss of generality that T has only one accepting configuration e. We shall see that parallel bit operations, together with operations that expand rapidly the length of a register, enable us to do each of these steps in very little time.

First let us see now to compute efficiently the transitive closure of a matrix A. We suppose that initially the whole matrix is in a single register. Remember that

$$A^* = I \vee A \vee A^2 \vee A^3 \vee \ldots \vee A^n \vee \ldots$$

where A is n by n and A^i is the i-th power of A in the "and-or" multiplication (i.e. if $C = A\,B$, $C_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$). Moreover, we may compute only the products $(I \vee A)$, $(I \vee A^2)$, $(I \vee A)^2 \cdot (I \vee A)^2 = (I \vee A)^4, \ldots$ where the exponent of $(I \vee A)$ is a power of 2. Since there are only logn of these $((I \vee A)^{n+k} = (I \vee A)^n)$ transitive closure of n by n matrices can be done in time logn times the time for multiplication. Throughout this section, "multiplication" will mean "$\wedge$" and "multiplication of matrices", "and-or" multiplication. Also, for simplicity, we assume n to be a power of 2.

To multiply two matrices efficiently, we observe that if we have several copies of the matrix stored in the same register in a convenient way, we can obtain all products in a single "$\wedge$" operation: all we need is that for all i, j, and k

a_{ik} be in the same bit position as b_{kj}. For example, if we have

$$\overbrace{(a_{0,0}a_{0,1}\cdots a_{0,n-1})(a_{0,0}\cdots a_{0,n-1})\cdots(a_{0,0}\cdots a_{0,n-1})}^{n \text{ times}}$$
$$(a_{1,0}\cdots a_{1,n-1})^n \cdots (a_{n-1,0}\cdots a_{n-1,n-1})^n$$

in one register (each row is repeated n times) and

$$[(b_{0,0}b_{1,0}\cdots b_{n-1,0})(b_{0,1}b_{1,1}\cdots b_{n-1,1})\cdots(b_{0,n-1}\cdots b_{n-1,n-1})]^n$$

(matrix stored by columns, repeated n times)

in the other, the "$\wedge$" of the two registers yields all terms $a_{ik} \wedge b_{kj}$. Supposing we are able to produce these forms of the matrices easily, all we have to do is collect terms and add ($\vee$) them up. Again, if we are able to take advantage of the parallel operations at their fullest, we should not have to do more than logn operations, since each c_{ij} is the sum of n products. Note that in our example, $c_{0,0}$ is the sum of the first n bits, $c_{0,1}$ of the next n, and in general c_{ij} is the sum of bits $1\cdot n + (j-1)n$ to $i\cdot n + jn - 1$.

We show first an algorithm to add up a row of bits -- the idea will be used in many of the constructions. For clarity, let us suppose that we have 8 elements, stored in register A, to add up.

$$A = a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7$$

We shall add the first half to the second half in parallel, in the following way: we use the mask

$$M = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$$

to get

$A \wedge M = 0 \quad 0 \quad 0 \quad 0 \quad a_4 \quad a_5 \quad a_6 \quad a_7$

Now if we slide A and A ∧ M relative to each other in such a way that a_4 and a_0 are superimposed (say by prefixing 0000 to A) we may add the two in parallel.

$(000).A = 0 \quad 0 \quad 0 \quad 0 \quad a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7$

$(A \wedge M) = 0 \quad 0 \quad 0 \quad 0 \quad a_4 \quad a_5 \quad a_6 \quad a_7$

$B = (0000.A) \vee (A \wedge M) =$

$0 \quad 0 \quad 0 \quad 0 \quad a_0 \vee a_4 \quad a_1 \vee a_5 \quad a_2 \vee a_6 \quad a_3 \vee a_7 \quad a_4 \quad a_5 \quad a_6 \quad a_7$

We may get rid of the final characters by forming $A = B \wedge M$. Now we have

$A = 0 \quad 0 \quad 0 \quad 0 \quad a_0 \vee a_4 \quad a_1 \vee a_5 \quad a_2 \vee a_6 \quad a_3 \vee a_7$

which, except for the leading 0s, is the same problem as before. Moreover, we may update the mask simply by $M = ((00).M) \wedge M$:

$M = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1$

which again elects the second half of A. Thus $B = A \wedge M$:

$B = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad a_2 \vee a_6 \quad a_3 \vee a_7$

and $A = ((00).A) \vee B$ produces

$A = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad a_0 \vee a_2 \vee a_4 \vee a_6 \quad a_1 \vee a \vee a_5 \vee a_7 \quad a_2 \vee a_6 \quad a_7$

the last two terms of which are eliminated to produce

$A = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad a_0 \vee a_2 \vee a_4 \vee a_6 \quad a_1 \vee a_3 \vee a_5 \vee a_7$.

The reader may easily verify that in the next iteration

$M = (0.m) \wedge M = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1$

$B = A \wedge M = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad a_1 \vee a_3 \vee a_5 \vee a_7$

$A = (0.A) \vee B$

$= 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ a_0 \vee a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5 \vee a_6 \vee a_7\ \ a_1 \vee a_3 \vee a_5 \vee a_7$

$A = A \wedge M = 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ a_0 \vee a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5 \vee a_6 \vee a_7$

which contains the desired result.

What is even nicer, if we use $(M)^{n^2}$ (n^2 copies of M) to begin with and use a register contnining all the products $a_{ik} \wedge b_{kj}$ in the format of our example, we shall get all elements of the product matrix (i.e. all sums) in time logn. The program is:

```
ADDUP: PROC
       M = (0^(n/2) . 1^(n/2))^(n^2)
       K = N/2
       while K ≥ 1 do
           B = A ∧ M
           A = ((0^k.A) ∨ B) ∧ M
           M = (0^k.M) ∧ M
           K = K/2
               end
                        end: ADDUP
```

We still have to show that M, and 0^k for k a power of 2, may be obtained efficiently; we do not give details but the concatenation of a string with itself p times results in a string of length 2^p.

Another problem is to expand a matrix from some standard input form (say stored by rows) into the forms used in forming the product. The idea is again the same: use masks and concatenations to get lots of elements in the places where we need

them in parallel. For example, to get

$$(a_{0,0}\cdots a_{0,n-1})^n(a_{1,0}\cdots a_{1,n-1})^n\cdots(a_{n-1,0}\cdots a_{n-1,n-1})^n$$

from

$$a_{0,0}\cdots a_{0,n-1}\ \ a_{1,0}\cdots a_{1,n-1}\ \cdots\ a_{n-1,n-1}$$

we first take the second half away, so that we may put the n/2-th row in its final place, using the mask $M' = 0^{n^2/2}1^{n^2/2}$:

$$B = M' \wedge A = 0\ldots0\ a_{n/2,0}\ \ a_{n/2,1}\ \cdots\ a_{n-1/n-1}\ .$$

We have to slide B under A in such a way that $a_{n/2,0}$ occupies the $n^3/2$-th bit position. For clarity of presentation, let us assume at this point that we have an "initial substring" operator, SUBSTR (A,B), that produces A minus its initial substring of length length B. (e.g. SUBSTR (10011,11) = 011.) Later we shall show how to do without this operator.

To put B in its place, all we need is:

```
Z = 0^(n^2/2) †
SHIFT = 0^(n^2/2)
K = N
while K ≥ 1 do
      SHIFT = SHIFT.SHIFT
      K = K/2
          end /* SHIFT is now 0^(n^3/2) */
SHIFT = SUBSTR(SHIFT,Z) /* now SHIFT is 0^(n^3/2 - n^2/2) */
          /* -- exactly the amount we need to move B by */
```

---

† Strictly speaking, this is illegal. $0^k = 0$ in our RAMs. However we may use 1^k and, in concatenations, do $(1^k.A)$ eor $1^k = 0^k.A$, so that we will use 0^k as an abbreviation.

We get the desired result by setting

$$A = (A \wedge -M') \vee B = a_{0,0}\ a_{0,1}\cdots a_{n/2-1,n-1}\ 0\ 0\cdots a_{n/2,0}\cdots a_{n/3,1}\cdots a_{n-1,n-1}$$

(position $\uparrow$ $n^3/2$)

This is the first step in our method, but it is reasonably clear how to proceed: the next mask should be

$$(0^{n^2/4}\ 1^{n^2/4}\ 0^{n^2/4}\ 1^{n^2/4})^n ,$$

which we may obtain by:

$N = (0^{n^2/4}.M')$ eor M'

$M' = \text{SUBSTR}(\ ,0^{n^2/4})$

and the second halves set $B = A \wedge M'$ will have to be shifted by $n^2/4\ (n-1)$ (remember, the previous iteration shifted B by $n_2/2\ (n-1)$. This will put rows n/2 and 3n/2 in their places. The reader should have no difficulty writing down an efficient program which produces:

$$A = a_{0,0}a_{0,1}\cdots a_{0,n-1}\underbrace{0\cdots0}_{n^2-n}\ a_{1,0}a_{1,1}\cdots a_{1,n-1}\underbrace{0\cdots0}_{n^2-n}\cdots a_{n-1,0}\cdots a_{n-1,n-1}$$

(the program outlined will run in time $O((\log n)^2)$, but may be modified to run in time $O(\log n)$). Finally,

```
SHIFT = 0^n
K = N
while K ≥ 1 do
      A = A ∨ (SHIFT.A)
      SHIFT = SHIFT.SHIFT
      K = K/2
      end
```

produces (in O(logn) moves) the matrix in the desired form.

Basically the same trick works to obtain the column form from the stored-by-row form: first we produce, as before, the form

(row 0) 0^{n^2-n} (row 1) 0^{n^2-n} ... (row n-1)

from which we get (using the same technique)

$$a_{0,0}0^{n-1}\ a_{0,1}0^{n-1}\ a_{0,2}0^{n-1}\ldots a_{0,n-1}0^{n-1}\ a_{1,0}0^{n-1}\ldots$$

$$a_{n-1,n-1}$$

(i.e. position of $a_{ij} = n^2i + nj$) from which, by using the mask $0^{n^3/2}1^{n^3/2}$ and again the same tricks, one obtains A in column order. Concatenation of this with itself logn times gives us the form needed for matrix multiplication in O(logn) operations.

We would like to emphasize the the routines presented above are not the most efficient or most economical in terms of storage. We just wanted to give a hint of the basic techniques and hope that the interested reader will be able to derive the complete programs himself, by using the tricks shown. In any case, we consider that we have proven (by sufficiently complicated example) that transitive closure may be computed in $L((\log n)^2)$ moves.

We still have to convince the reader that given a polynomial tape bounded Tm with input x, we can obtain the matrix of the "follow in one move" relation easily. We shall do this in an even sketchier way than our exposition of the method for computing transitive closure.

If a Tm operates on an input of length n in tape n^k, there

are at most $O(2^{cn^k})$ different configurations, c a constant. Let us take a convenient encoding of these in the alphabet {0,1} and interpret the encodings as integers. By convenient encoding we mean one that is linear in the length of the tape used by the machine, where the positions of the heads and the state may be easily found, and which may be easily updated. Then, if we generate all the integers in the range 0 - $(2^{cn^k}-1)$ (where c depends only on the encoding) we shall have produced encodings of all configurations, together with numbers that are not encodings of any configuration. The reader might amuse himself by writing a CRAM program that produces all integers between 0 and $m = 2^p-1$ in time p. (Hint: for a straightforward program get $(m = 2^p)$ $1^{m/2}0^{m/2}$, then $(10^{p-1})^{m/2}$, then $0\ 1^{m/4}\ 0^{m/4}\ 1^{m/4}\ 0^{m/4}$ and $0\ (10^{p-1})^{m/4}\ 0^{m/4}\ (10^{p-1})^{m/4}$, V them together, etc.)

We shall, for the rest of this section, use single-tape Tms. This is technically convenient and can be done without loss of generality. It is well known that in the operation of such a Tm the character under the read-write head, the two symbols in the squares immediately to the right and left of it, the state of the finite control and the position of the input head uniquely determine the next configuration. Then we test whether configuration c_j follows from c_i as follows: suppose c_i and c_j are stored in registers R and S. We first build a pattern which picks up head positions (i.e., once we build the pattern, we obtain from R and S, in a constant number of moves, bit vectors which have a 1 at the position scanned by the head

and 0s everywhere else -- moreover the sequence of moves is independent from the contents of R and S. For example, suppose that the head position is indicated by the pattern 11011 appearing beginning at some position $p \equiv 0 \pmod{15}$ in the encoding of the configuration, which we suppose of length $\ell' = 5$. Then $M_h = (00100)^{\ell}$ is a mask with the property that $T = M_h$ eor R will have 11111 starting at a position $p \equiv 0 \pmod 5$ iff R had 11011 there. Using a procedure similar to ADDUP, we get a vector which is 1 only at such positions and 0 everywhere else). Now, again in a manner that does not depend on where the head is in c_i, we may, in another constant number of moves, obtain the three squares of R that matter for the determination of the next configuration, as well as the state of the finite control. We save this information and zero the corresponding bits in the encodings, both in R and in S. All of this can be done in a constant number of moves, which are independent of the contents of R.

Now to verify that the transition was a permissible move of the Tm we have to check that the non-blank portions of R and S are identical and that the blanked-out bits satisfy a move rule. The latter is verified by table look-up, where the size of the table depends only on the Tm but not on the input, while the former is checked by first taking R eor S (R and S now have 0s where a move might change R) and using a version of ADDUP to verify that the result consists only of 0s. This will take only $O(\log \ell)$ moves, where ℓ is the length of the ID.

Thus, we know how to detect the fact that c_j follows from

c_i in O(logℓ) CRAM moves, where ℓ is the length of the configuration and the moves do not depend on the contents of c_i or c_j. This is important, because it shows that if we have $c_{i0}\ c_{i1}\cdots c_{ik}$ in R, $c_{j0}\cdots c_{jk}$ in S, we may, in O(logℓ) moves, test simultaneously whether c_{jk} follows from c_{ik}. Now, the way to generate the transition matrix in time O(nk), where n is the length of the input, is easy enough to guess: first we generate all integers in the range 0 - $(2^{n^k} - 1)$; call these configurations c_i. Then, as in the matrix product routine, we form

$$(c_0)^m\ (c_1)^m \cdots\ (c_{m-1})^m$$

where $m = 2^{n^k}$ and $(c_0)^m$ means m-fold concatenation, and

$$(c_0\ c_1 \cdots\ c_{m-1})^m$$

in O(logm) = $O(n^k)$ operations, and in $O(n^k)$ operations determine simultaneously for all i and j whether c_j follows from c_i (i.e. obtain a vector of bits which is 1 iff c_j follows from c_i). This is done in two stages: first we obtain, for each pair c_i, c_j a string of length ℓ which contains only 0s iff c_j may follow from c_i in a single move. All such strings are obtained concatenated to each other in a single register. Now, by using ADDUP we obtain a 0 at the beginning of each string if it contained only 0s, and a 1 otherwise, and compactify the string, eliminating the "garbage" between any two such numbers. The compactification routine uses the technique of the matrix rearrangement programs sketched before. After compactification the matrix is in the format required for the transitive closure algorithm.

This completes the description of our simulation algorithm:

putting everything together we still have a procedure which runs in polynomial time, since the matrix may be computed in $O(\log 2^{cn^k})$ moves and its transitive closure in $O((\log 2^{cn^k})^2 = O(n^{2k})$ moves.

Finally, some comments about the instruction sets necessary to do this simulation. In our programs we used, besides parallel bit operations, the following: concatenation (.), SUBSTR, and loop control operations (comparisons and divisions by 2). We first show, as we have promised, how to eliminate SUBSTR. The basic idea is simple, and we used it implicitly in ADDUP: the SUBSTR operation is used to drop off an initial substring of a string to obtain alighment -- but the same effect can be obtained by concatenating a string of 0s of the same length to the other string. This has the disadvantage that now we have a certain amount of useless garbage preceding certain variables, but that can be taken care of by the following:

first, it is easy to see that we may always assume that the initial segment is a string of 0s, since for any prefix P, $(P.A)$ eor $P = 0^{\text{length}(P)}.A$;

second, we maintain, for each variable, an associated "garbage indicator" -- another register, which contains a string of 1s of length equal to the useless initial segment. Whenever a variable with a nonempty garbage indicator is used in conjunction with others, if the operation is a boolean one we prefix the other operand with the garbage indicator transformed into 0s. If we want to form $C = A.B$ but we have only $A' = 0^{n_1}.A$, $B' = 0^{n_2}.B$, $G_{A'} = 1^{n_1}$, $G_{B'} = 1^{n_2}$, we form

$C = A'.B'$ $(= 0^{n_1}.A.0^{n_2}.B)$

$C = C$ eor A $(= 0^{n_1+ n_2+ \text{length}(A)}.B)$

$C = C$ V $((G_{B'}.A)$ eor $G_{B'})$ $(= 0^{n_1+ n_2}.A.A)$

$G_C = G_A.G_B$.

In both cases we have only a constant amount of overhead per operation. Thus SUBSTR is not necessary.

As for the loop control, it is again easy to see that division by 2 is not necessary, since it is SUBSTR(,1). Thus the main theorem of this section may be written:

CRAMs without arithmetic instructions may simulate PTAPE in polynomial time.

We now sketch proofs of how our more powerful RAM models may simulate CRAMs.

1) VRAMs

Clearly A.B is the same as A V (B ↑ length(A)). All we have to show is that the necessary lengths are attainable in polynomial time for polynomial time bounded CRAM computations.

Initially we store the lengths of all constants used in the CRAM program in the VRAM's program. The length of the input may be obtained in linear time. The longest string obtainable in t moves from a set S of strings, by a CRAM is given by the program:

```
DUPL: I = 0
      while I < t do
            S0 = S0 · S0
            I = I + 1 ; end
```

where S_0 is the longest string in S. The length of this string will be $2^t \text{length}(S_0)$. It is easy to devise a binary search type VRAM-algorithm that will find the length of a string in time $O((\log \text{length}(x))^2)$: one builds a string of 1s, doubles them and tests when this procedure produces a string longer than x. When this hpapens, after i duplications, we know that

$$2^{i-1} \leq \text{length}(x) < 2^i .$$

We take $\text{SUBSTR}(x,1^{2^{i-1}})$ and call the procedure recursively. Clearly, at most log length(x) iterations are needed, each of which takes at most O(log length(x)) time -- hence the time bound. Thus, if a CRAM operates in time n^k, we may simulate each of its steps in at most $O(n^{2k})$ VRAM steps and, therefore, the whole computation in time $O(n^{3k})$. Again, the simulation technique is not optimal, but, we hope, transparent.

2) VRAMs

First we note that "shift left" instructions are unnecessary for VRAM acceptors: the proof is identical to the argument given to show that SUBSTR is not necessary for CRAMs -- roughly, that one shifts everybody else right instead of shifting one register left, and takes into account that initial segments of some registers should be considered garbage. With this in mind, all we have to simulate is the instruction

$$V_j \leftarrow V_i \uparrow J_k \quad \text{(shift right)} .$$

Clearly, this is equivalent to the MRAM instruction

$$V_j \leftarrow V_i \cdot 2^{I_k}$$

and all we have to show is that it is possible to have 2^{I_k} in an MRAM register when I_k is used in a VRAM shift instruction.

We shall argue, as in the CRAM case, that the contents of I_k cannot be too big: since the only operation that increases the contents of an index register is addition, the program that creates the biggest possible number in an index register in t steps of a VRAM's computation consists of adding a register repeatedly to itself. If the initial contents of the register was k, we produce $2^t k$ after t operations. Therefore, in general, we must have at time t all index registers containing numbers of length t to c at most, c a constant depending only on the machine. But we may generate all numbers of the form 2^m, length(m) $\leq$ t+c in time polynomial in t: we get the powers of 2 by multiplying 2 by itself and include those factors in the final product for which m has a 1 bit.

This proves the inclusion PTIME - MRAM $\supseteq$ PTIME - VRAM and concludes the chain of implications proving PTAPE $\subseteq$ PTIME - MRAM, the objective of this section.

3.4 Comments

After the programming details of the previous two sections, it might be useful to restate the results of this chapter. We considered MRAMs, random access machines with multiplication as a primitive operation, and proved two important facts about their power as recognizers:

1) deterministic and nondeterministic time complexity classes are polynomially related (or PTIME - MRAM = NPTIME - MRAM)

2) time-bounded computations are polynomially related to Tm tape (PTIME - MRAM = PTAPE).

Since it can be proven that RAM time and Tm time are polynomially related, we also proved

3) RAM running times with and without multiplication are polynomially related if and only if Tm time and tape measures are polynomially related.

This last observation is interesting, since it seems to imply that the elusive difference between time and memory measures for Tms might perhaps be attacked by "algebraic" techniques developed in "low level" complexity theory. We obtained no results in this direction: the sort of problem for which lower bounds on the number of multiplications are known compute functions, and for transducers we do already know that tape is more powerful than time.

We also note that RAMs may simulate MRAMs in polynomial time, as long as MRAMs operate in polynomial space and time. Therefore MRAMs are more powerful than RAMs if and only if the unit and logarithmic time measures are not polynomially related -- i.e. if (in our "polynomial smearing" language) the two are distinct measures.

Many "if and only if" type categories follow, in the same vein, from 1), 2) and 3). For example:

"The set of regular expressions whose complements are nonempty is accepted in polynomial time by a deterministic Tm iff every language recognized by an MRAM in polynomial time is recognized by a deterministic RAM in polynomial time."

The reader may write down many of these: some of them sould quite surprising at first.

As we saw, if MRAMs are different from RAMs, they must use

an exponential amount of storage. This suggests asking whether it is sufficient to have a RAM_1 and exponential tape to get an MRAM's power or, equivalently, to look at operations that make RAM - PTIME classes equivalent to PTAPE. The answer, as we saw, is that almost anything that expands the length of registers fast enough will do, as long as we have parallel bit operations: multiplication, concatenation or shifting all have this property. In particular, one of our CRAM models has nothing but concatenation, tests and parallel bit operations (no indirect addressing). On the other hand, we saw that adding more and more powerful operations (indirect addressing, shifts by shift registers, division by 2, SUBSTR, multiplication, integer division) do not make the model more powerful, once we have a fast memory-augmenting device. The stability of this class of RAMs makes them a nice characterization of memory-bound complexity classes. We also think they might be useful for studying parallelism.

Minsky suggested [49] that one of the objectives of theoretical computer science should be the study of trade-offs (e.g. between memory and time, nondeterminism and time, etc.). Our constructions trade exponential storage for polynomial time (simulation of Tms by MRAMs) and polynomial tape for exponential time in the other simulation. Whether this trade-off is real or the result of bad programming is not known, isnce P = PTAPE? is an open problem. If P $\neq$ PTAPE, then PTAPE would provide us with a class of languages which have a trade-off property: they may be recognized either in polynomial time or in

polynomial storage, but not simultaneously.

Corollary : PTIME ≠ PTAPE iff there exists a language L which can be recognized by MRAMs in polynomial time and polynomial memory, but not simultaneously.

Note that if such an L exists, any tape complete problem be chosen to be it, for example L_R^*. Moreover, if P ≠ PTAPE, in the model in which they are recognized in polynomial time, the checking of the fact that one configuration came legally from the previous one would of course have to take exponential tape. P ≠ PTAPE would also imply that the tape · time measure for Tms is too coarse, since it would put in the same class PTAPE and exponential tape.

As an illustration of what restrictions to the power of MRAMs may do, we present the

Lemma: The class of languages recognized by the restricted RAMs below are:

(instructions in brackets may be removed)

instruction set	restriction	language
comp, (÷),(·-), bool, (i.a.†),+,*	(N)PTIME	PTAPE
comp, (÷),(·-), (bool), (i.a),+	NPTIME	NP
comp, (÷),(·-), (bool), (i.a),+	PTIME	P
comp, (÷),·- , bool	none	csℓ
comp, (÷),·-, bool	deterministic	dcsℓ
comp, (÷),(·-), bool, (i.a.),+,*	time ≤ $(\log n)^i$	LG††

† indirect addressing

†† LG = {languages recognizable in tape $(\log n)^i$ for some i}

instruction set	restriction	language
comp, $\div$	division only by 2	NLOG-TAPE
comp, $\div$	division only by 2, deterministic	DLOG-TAPE
comp, $\div$	2 registers only	Regular sets

Lines 4 and 5 were observed by [68]; line 6 is obtained by noting that Theorems 1 and 2 hold when strengthened to time and tape constructible bounds greater than the logarithm; 7 and 8 follow from the characterization of logarithmic space-recognizable languages as the ones accepted by k-head finite automata for some k.

Finally the idea of using boolean operations may be crucial to our results. It was first used in [53] where VRAMs were defined. Stockmeyer (personal communication) has an argument to show that boolean operations are expensive on MRAMs (but the argument fails for PRAMs). The question of whether they are necessary is an interesting open problem.

Chapter 4 ON NONDETERMINISM

"If you want to understand the invisible, look very carefully at the visible"

The Talmud

4.1 Outline

In this chapter we first discuss several notions of nondeterminism, trying to get an insight into the nature of nondeterministic computations by examining these alternative approaches. In the rest of the chapter we extend the definitions of nondeterministic polynomially time bounded languages in several directions, obtaining several new results. In section 4.3 we recapitulate the notion of complete languages, examine some of the combinatorial NP-complete problems and show that there is an interesting class of combinatorial problems that are easily reducible to each other and seem to lie between NP and PTAPE.

In section 4.4 we define a model of parallel Turing machines, the threshold machines, and show that the computational power of polynomially time bounded threshold machines is between NP and PTAPE, and is precisely equal to the power of polynomially time bounded probabilistic Turing machines of Gill [19]. Moreover, the combinatorial problems introduced in section 4.3 are complete languages for these parallel machines. We also exhibit a sequence of models of self-reproducing tesselation automata that are naturally suggested by this approach, which model deterministic, nondeterministic, threshold and tape-bounded machines. The increasing power of such ma-

machines results clearly from increasing use of parallelism and increasing ease of communication among the automata.

Section 4.5 is the extension of one of the formal descriptions of nondeterminism (the polynomial hierarchy of Meyer and Stockmeyer) and gives a new subrecursive hierarchy within the elementary functions.

Section 4.6 examines approximate solutions to NP-complete problems. Existing approaches are criticized and a new definition of approximation is given, as well as some arguments that seem to indicate that no really good approximate solutions exist for NP-complete problems. Throughout the chapter the idea of cardinality-preserving transformations occurs over and over again. We conclude section 4.6 by pointing out the importance of such reductions for suggesting problems and conjectures.

4.2 What is nondeterminism?

"Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood ...
...
I took the one less traveled by,
And that has made all the difference."

Robert Frost

We say in the introduction that the problem of the cost of nondeterminism is a very important one and in particular that the question P =? NP must be answered if a number of interesting problems in computational complexity are to be

solved. It is also clear that P =? NP is a very difficult problem, and it is likely that a new insight will be necessary to settle it. We study, therefore, alternative descriptions of nondeterminism in this section. We do not claim originality or profoundness to these models, but although none of them provided us with an insight that was powerful enough to produce a proof for the "big" problem, the next four sections contain many interesting results and questions that were inspired by the examination of such formalizations.

We shall indicate now models for nondeterministic computations. We restrict ourselves to language recognition (or equivalently 0-1 valued functions); the general case of nondeterministic transducers is more complicated, in that several possible "natural" definitions of nondeterministic computation are possible and they are not equivalent. We hope to do some work in this area in the future.

a) Machines with nondeterministic moves

The first formalization is that of a machine with a many-valued transition function. This was the formalism we presented in Chapter 2. Thus, in some situations the machine has several possible choices for the next move. For the purpose of counting steps, we take the choice which results in an accepting configuration and, among these, has the minimal length. Note that it is quite possible that for some choice of moves the machine rejects its input or that it jams, but we still accept the string in some other sequence of moves. It is also conceivable that the machine diverges if a certain

sequence of choices is made, but we may, at a small extra cost, run a "clock" in parallel with the machine. Since we are mainly interested in polynomial time bounded computations, we may assume wlog that the clock forces every computation to halt after some fixed, polynomial number of steps. This eliminates the problem above.

It is sometimes easier to think about such computations as proofs in a formal system (or equivalently as derivations in a type 0 grammar). It is natural then to assume that several possible moves exist in a given situation -- that is exactly what happens in a proof system or in a derivation. The measure used becomes natural too: the length of a proof is the length of a shortest proof; "white forces mate in three moves" means that if white chooses the best possible move, he will win the game in three moves (even if other moves might result in a loss or draw), cf. [14].

In these systems we are defining a computation (or proof) as the shortest among a set of computations. We might as well keep the "bad" choices and look at nondeterministic computations as

b) A tree of computations

The root of the tree is the initial configuration and every node has as successors all the configurations that can be obtained in one step from it. Because of our convention about the "clock", the tree is finite. The length of an accepting computation is the shortest among paths from the root to a node which represents an accepting configuration. Alternatively,

for deterministically constructable time bounds, we may simply shut off all computations by the clock and assume that accepting configurations keep repeating, until they are shut off by the clock. For the study of polynomial classes there is no loss of generality in the second approach. Looking now at the tree, we see that the string is accepted iff there is a leaf which is an accepting configuration.

This approach has the advantage that the tree contains all the computations and thus seems more tractable mathematically. In fact, it suggests the question, "What is computed if acceptance is defined by some other predicate on the tree?" We shall consider this problem in section 4.4

It is clear that nondeterministic computations may be considered

c) Restricted parallel machines

i.e. the parallel machines have no interaction among themselves, except for a final OR-ing of their outputs. In section 4.4 we shall discuss this alternative model.

Finally, an alternative approach is to consider nondeterministic predicates as being obtained by projection from deterministic predicates [61].

d) Projection (polynomially bounded existential quantification)

Given a predicate on two variables $R(x,y)$, the predicate $S(x) = \exists y\ R(x,y)$ is said to be obtained from R by projection. Clearly any nondeterministic computation can be written as a projection of a deterministic predicate, with y being (an encoding of) the choices that the machine must make in order to

accept x. Moreover, such a sequence must be of polynomial length at most, if the machine is polynomially time bounded, since at most a polynomial number of choices was made.

Conversely, every predicate of the form

$$\exists y\ R(x,y)$$

where R is computable in deterministic polynomial time and the length of y is bounded by a polynomial in the length of x can be computed by a machine in NP that guesses y one character at a time (that takes polynomial time) and then computs, again in polynomial time, R(x,y).

The advantage of this method is that as R ranges over the recursive predicates,

$$\{x \mid R(x)\}$$

ranges over the recursive sets, whereas for unbounded quantifiers,

$$\{x \mid \exists y\ R(x,y)\}$$

ranges over the r.e. sets, which we know to be a different class of languages. We study such definitions in section 4.4, where we obtain a new nondeterministic hierarchy for the elementary languages.

Finally, from a language-theoretic point of view, because of the existence of complete problems, we can think of any language in NP as represented by a subset of a complete language, i.e.

e) Inverse homomorphic image of a subset of the satisfiable Boolean formulas

We shall discuss such characterizations and their implications in sections 4.3 and 4.5.

4.3 Complete problems, reductions and conservation properties.

In this section we re-examine some of the known P-complete problems [9],[38],[58],[62],[18].We have discussed the existence of such languages in Chapter 2. Let us now prove a (somewhat strengthened) version of Cook's fundamental result that the set of satisfiable boolean expressions is an NP-complete language. This was (2) in our list of NP-complete languages in Chapter 2.

Theorem 4.1: Let M be a fixed nondeterministic polynomial time bounded Tm, and x an input to it. Then there is a boolean expression $b_{M,x}$ such that

1) the length of $b_{M,x}$ is bounded by a polynomial in x .
2) $b_{M,x}$ can be constructed in deterministic time bounded by a polynomial in x..
3) there are k different assignments that make $b_{M,x}$ true iff there are k different sequences of moves that make M accept x within its time bound.

Proof: Given M, a nondeterministic n^k time bounded Tm, and an input x, we build $b_{M,x}$ to be satisfiable iff M accepts x. The encoding consists of OR-ing together expressions that state that

1) the encoding is the encoding of ID's.
2) the initial state is "right".
3) the last state is accepting.
4) the transitions are correct.

To do this we get the following variables:

$Sq^i_{s,t}$ true iff at time t the i-th tape square contains the symbol s.

$St_{q,t}$ true iff at time t the Tm is in state q.

H^i_t true iff at time t the Tm's head is scanning the i-th tape square.

To state 1) we write down that for all times t, $1 \leq t \leq T = n^k$, one and only one square is scanned, that only one state is occupied, and that every square has only symbol. "Only one square is scanned at time t" is expressed by

$$Onescan_t = (H^1_t \vee H^2_t \vee \ldots \vee H^T_t) \text{ and } [\bigwedge_{1\leq i,j\leq T} (\neg H^i_t \vee \neg H^j_t)]$$

and

$$Osc = Onescan_1 \text{ and } Onescan_2 \text{ and } \ldots \text{ and } Onescan_T$$

ensures that at all times exactly one position is scanned.

Similarly, "one symbol per square" is ensured by first stating

$$Onesymb_{i,t} = (Sq^i_{symb1,t} \vee Sq^i_{symb2,t} \vee \ldots \vee Sq^i_{symbs,t}) \wedge [\bigwedge_{1\leq symb<symb'\leq s} \neg Sq^i_{symb,t} \vee \neg Sq^i_{symb',t}]$$

(where we assume that the alphabet has s symbols)

$Onesymb_{i,t}$ says that square i at time t contained exactly one symbol. Now

$$Onesymb = \bigwedge_{1\leq i\leq T} \bigwedge_{1\leq t\leq T} Onesymb_{i,t}$$

is the desired formula. Similarly, we define Ostate to mean that at all times exactly one state is assumed by the finite control.

The correctness of the initial configuration is expressed by

$$\text{INIT} = [St_{q_0,0} \wedge H_0^1 \wedge Sq^1_{x_1,0} \wedge Sq^2_{x_2,0} \wedge \ldots \wedge Sq^n_{x_n,0} \wedge Sq^{n+1}_{\not b,0} \wedge \ldots \wedge Sq^T_{\not b,0}]$$

where q_0 is M's initial state, $x = x_1x_2\ldots x_n$ and $\not b$ is the symbol for blank.

If q_f is M's accepting state, acceptance is guaranteed by

$$\text{Accept} = St_{q_f,t}.$$

To ensure proper transitions we require that at all times and all positions a valid transition rule be followed. For example if

$$\delta(a,q) = \{(b,S,p)(c,R,p')\}$$

(S = stay, R = move right)

then we require

$$\text{Corr}_{i,t} = St_{q,t} \wedge Sq^i_{a,t} \wedge H^i_t \supset [(H^{i+1}_{t+1} \wedge St_{p',t+1} \wedge Sq^i_{c,t+1}) \vee (H^i_{t+1} \wedge St_{p,t+1} \wedge Sq^i_{b,t+1})]$$

at all times t and positions i (actually the boundary positions should be dealt with separately but this is easy).

Lemma: Let b be a boolean expression. Then there is an expression b', using only the connectives $\vee$, $\wedge$, and $\neg$ (in fact in conjunctive normal form) such that

1) b' is satisfied by k truth assignments iff b is satisfied by k truth assignments
2) length(b') is bounded by a polynomial in the length of b
3) b' may be obtained from b in polynomial time (in the length of b).

The new expression b' may have additional variables, that did not appear in b.

Proof: Let c be an innermost subformula of b which is a truth-functional combination of two variables, say x_1 and x_2, and in which the truth functional connective is not V. Replace the subformula by a new variable y_1 and conjoin to b four clauses which guarantee that any truth assignment which satisfies b must assign the same truth value to y_1 as to c.

For example, if c is $x_1 \wedge \bar{x}_2$, then we replace c by y_1 and add the clauses

$$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{y}_1) \wedge (\bar{x}_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \bar{y}_1) \wedge$$

$$(x_1 \vee \bar{x}_2 \vee \bar{y}_1)$$

These may be read as: "If x_1 is true and x_2 is true then y_1 must be false (or else the first clause is false); if x_1 is true and x_2 is false then y_1 must be true (or else the second new clause is false), etc.

Similarly,

$$(x_3 \supset x_4) \equiv y_2$$

is implied by

$$(\bar{x}_3 \vee x_4 \vee \bar{y}_2) \wedge (x_3 \vee x_4 \vee y_2) \wedge (x_3 \vee \bar{x}_4 \vee y_2) \wedge$$

$$(\bar{x}_3 \vee \bar{x}_4 \vee y_2) .$$

Now $(x_1 \wedge \bar{x}_2) \quad (x_3 \wedge x_4)$ may be written as

$$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{y}_1) \wedge (\bar{x}_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \bar{y}_1) \wedge$$

$$(x_1 \vee \bar{x}_2 \vee \bar{y}_1) \wedge (\bar{x}_3 \vee x_4 \vee \bar{y}_2) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee y_2) \wedge$$

$$(x_3 \vee \bar{x}_4 \vee \bar{y}_2) \wedge (x_3 \vee x_4 \vee \bar{y}_1) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_3) \wedge$$

$$(y_1 \vee y_2 \vee y_3) \wedge (y_1 \vee \bar{y}_2 \vee y_3) \wedge (\bar{y}_1 \vee \bar{y}_2 \vee y_3) \wedge y_3.$$

The example above should clarify the algorithm. It is easily seen that if an assignment $x_1,\ldots,x_n$ satisfies the

original formula, there is a unique set of values of y's that make the translated formula true, and conversely if an assignment makes the translated formula true, its restriction to the x-variables satisfies the original formula. Since each subformula is uniquely characterized by its connective, and the elimination of each connective introduces only four new formulas containing three terms each, the claims of the lemma are proven. Versions of this lemma appear in [3],[66].

Note that the same trick can be used to reduce the number of variables per clause in a formula to three. ([8])

Now, let $Corr'_{i,t}$ be the version of $Corr_{i,t}$ obtained by th algorithm of the previous lemma. Then if

$$Corr = \bigwedge_{1\leq t\leq T} \bigwedge_{1\leq i\leq T} Corr'_{i,t}$$

then

$$F = Osc \wedge Osymb \wedge Ostate \wedge Init \wedge Accept \wedge Corr$$

has the properties claimed for $b_{M,x}$.

Many different definitions of polynomial reductions were studied in [42], in analogy to the reductions of recursive function theory. The most useful of these seems to be the many-one reducibility of Karp, i.e.

$A \leq B$ (A reducible to B) iff there exists a function $f \in p$ such that $x \in A$ iff $f(x) \in B$.

We just showed that there is a reducibility that does more: there are k different ways in which x may be accepted by a certain machine that accepts A iff this is true of f(x) and B. This suggests a generalization of the satisfiability problem:

Definition: The k-satisfiability problem is, "Given a boolean expression b (in conjunctive normal form) and an integer k, are there at least k different assignments of the values "true" and "false" to b's variables that satisfy b?"

Our theorem proves not only that satisfiability is an NP-complete problem, but also shows that k-satisfiability has an analogous completeness property with respect to nondeterministic polynomial time bounded Tms that may accept their input k different ways. We know from the work of Karp [38] and others that many combinatorial problems are reducible to each other and to the satisfiability problem. We shall see that similar reductions hold between the k-satisfiability problem and many combinatorial problems.

Definition: given two families of combinatorial problems A and B, a mapping $f(\ ): A \to B$ is parsimonious iff

1) given a problem $a \in A$, $f(a) \in B$ has exactly k solutions iff a has exactly k solutions.

2) f(a) may be computed in time polynomial in the size of a.

Convention: given a (reasonably well-known) combinatorial problem A (i.e. the recognition problem for the language

$\{a \mid a \in A$ and a has a solution $\}$

we shall denote by k - A the language

$\{(k,a) \mid a \in A, k \in N$ and a has at least k distinct solutions$\}$.

Let B be another combinatorial problem, and suppose that there exist parsimonious mappings $f(\ ): A \to B$ and $g(\ ): B \to A$. Consider the languages k - A and k - B. Then the following is true:

(m,a): (m,a) $\in$ k - A iff (m,f(a)) $\in$ k - B

(m,b): (m,b) $\in$ k - B iff (m,g(b)) $\in$ k - B .

Thus, the existence of a parsimonious mapping between A and B, and between B and A ensures that the recognition problem of the languages k - A and k - B are equally difficult (i.e., within a polynomial of each other). Whenever this happens, we say that k - A and k - B are related by parsimonious reductions.

For the notation used in the next theorem, see the references.

Theorem 4.2: The following combinatorial problems are related by parsimonious reductions:

1) k-satisfiability of arbitrary boolean expressions.
2) k-satisfiability of CNF expressions with at most 3 literals per clause.
3) k-CLIQUE [38]
4) k-NODE COVER [38]
5) k-SET PACKING [38]
6) k-SET COVER [38]
7) k-(0-1) INTEGER PROGRAMMING [38]
8) k-EXACT COVER [38],[15]
9) k-HITTING SET [38]
10) k-KNAPSACK [38]
11) k-STEINER TREE [38]
12) k-JOB SEQUENCING [38]
13) k-PARTITION [38]
14) k-INEQUIVALENT REGULAR EXPRESSIONS OVER $(\cup,\cdot)$ [30]
15) k-MAX CUT [38]

16) k-MAXIMUM SATISFIABILITY WITH 2 LITERALS [18]

17) k-SIMPLE MAX CUT [18]

18) k-MINIMUM NODE DELETION BIPARTITE SUBGRAPH [18]

19) k-NETWORK FLOWS WITH MULTIPLIERS [58]

20) k-acceptance of a string by a polynomial time bounded nondeterministic Tm.

Proof: We have proven 1) $\leq$ 20) and 20) $\leq$ 1). In fact, each of the problems 1) - 19) has an easy reduction to 20), namely consider the Tm that guesses a solution to the problem and then verifies deterministically that the guess is correct (two different choices of actions during the guessing process must lead to different guesses). The proofs of the other reductions are mostly a routine check that the original reductions given to show that the 1-PROBLEM is NP-complete are in fact parsimonious, as the reader is encouraged to verify. Exceptions to this are:

1) k-EXACT COVER. In Karp's paper this is obtained by reducing it to the chromatic number problem, which is reduced to SATISFIABILITY WITH 3 VARIABLES. Unfortunately, this latter reduction is not parsimonious. However, the direct encoding of accpeting Tm computations described by Galil [15] is parsimonious.*

2) k-MAX CUT was proven through k-SIMPLE MAX CUT.

3) k-MAXIMUM SATISFIABILITY. In [18] the reduction is not parsimonious. However, the following encoding will work (notation is from [18]):

* The definition of exact cover in that paper is more restrictive then in Karp, but a straightforward modification of the construction will result in Karp's definition.

If the original formula was $S = \bigwedge_{i=1}^{m} (a_i \vee b_i \vee c_i)$, get

$$S' = \bigwedge_{i=1}^{m} [a_i \wedge b_i \wedge c_i \wedge d_i \wedge (\bar{a}_i \vee \bar{b}_i) \wedge (\bar{a}_i \vee \bar{c}_i) \wedge (\bar{b}_i \vee \bar{c}_i)$$
$$\wedge (a_i \vee \bar{d}_i) \wedge (b_i \vee \bar{d}_i) \wedge (c_i \vee \bar{d}_i) \wedge (a_i \vee \bar{d}_i)$$
$$\wedge (b_i \vee \bar{d}_i) \wedge (c_i \vee \bar{d}_i)]$$

where the d_i's are new variables.

Then S is satisfiable iff 10m of S' 's clauses may be simultaneously satisfied, and this reduction is parsimonious.

Comments:

1) The list above is certainly not exhaustive. Its only purpose is to try to impress upon the reader the idea that the class of these combinatorial problems is quite large.

2) Since the reduction is polynomial time bounded, if any of these languages are in P, all of them are. (Also, most of the reductions are log-space bounded). We shall study these properties with more detail in the next section.

3) The reader familiar with NP-complete problems might be wondering about the absence of some famous NP-complete problems from our list. In particular, HAMILTONIAN CIRCUIT [38], REGISTER ALLOCATION [62] and their variants do not appear. The reason is that for the second a program using m registers may have several valid arrangements of its assignments such that all use at most m registers. For the HAMILTONIAN CIRCUIT problem, the difficulty is in the reduction used: the usual technique is: given a graph G, build G' with k "new nodes" and 4 nodes per each node of G, such that G' has (k-1)! directed Hamiltonian circuits for each node cover of size k in G (see

[1] Ch. 10). Again, these circuits differ only in nonessential ways, namely in the order of the "new nodes". If we are willing to relax our definitions a little, by considering these solutions to be equivalent, we could add them to our list of complete problems. The intuitive idea about these reductions is that we can "read off" the solution of one problem from the solution of the problem it reduces to. Formally,

Definition: A reduction f, $f:A \to B$, is transparent if

1) there exists a relation $R(b,S_1,S_2) \in P$ among the solutions S_1,S_2 of $b \in B$ such that given a problem $a \in A$, f(a) has exactly k distinct solutions modulo R iff a has exactly k distinct solutions.

2) $f \in P$.

Theorem 4.3: The following problems can be added to 1) - 20) if "parsimonious" is replaced by "transparent":

21) k-DIRECTED HAMILTONIAN CIRCUIT [38]

22) k-UNDIRECTED HAMILTONIAN CIRCUIT [38]

23) k-UNDIRECTED HAMILTONIAN CIRCUIT with vertex degree at most 3 [18]

24) k-[m-register compilable programs] [62]

Proof: remarks above.

The idea behind the definition is the following: if a reduction f is transparent, given a problem $a \in A$, and a solution s to $f(a) \in B$, we can extract from s -- in deterministic polynomial time -- a solution to a. In an arbitrary reduction, we would conceivably have to have the answers to a polynomial number of problems in B, and the problems would depend on the

answers to the previous questions, although, or course, the existence of a solution can be obtained by asking a single question. In recursion-theoretic terms, a solution is computable by a single oracle call by a fixed program, while in the general case the full power of Turing reducibility would be needed. In other words, problems related by such reductions are really different syntactic versions of the same problem. It appears to us that these reductions are more natural than the usual definition and that they deserve further study.

In the next section we discuss a machine model for accepting our k-NP languages.

4.4 Threshold machines

Let M be a nondeterministic polynomial time bounded Tm. wlog assume that M clocks itself -- i.e. given an input x, M first computes the time bound in which it is supposed to operate, writes it out in tally notation on a separate "clock" tape, and then between any two transitions decrements the value of the clock by one. When the clock reaches 0, the machine halts. This restriction does not increase computing times by more than a polynomial amount, and it is very convenient for our purposes. We will assume in this section that all Tms have been modified to have such behavior.

Now consider the set of all computations of M on a given input x of length n. As we mentioned before, we may arrange the sequences of instantaneous descriptions (ID's) as a tree: at the root we shall put the initial description, and as descendants of a node labeled y we will put all the ID's that

result from a legal move of M on y. We call the structure obtained the computation tree of M on input x. The structure is indeed a tree (by construction) although it is conceivable that several of the ID's appear as descendants of different nodes. We may also assume, wlog, that it is a binary tree, since the outdegree of a node is the number of possible transitions of M from a given configuration, and that can always be bounded by 2.

The nice thing about computation trees is that they contain all the information about what M could possibly have done on input x in the allotted time. Since we are assuming our machines to be polynomially bounded and clocked, the tree has as most polynomial depth, and, because of our assumption about M, the number of nodes is at most proportional to 2^T, where T is M's running time. It is also clear that T accepts x iff the computation tree has a leaf in which the finite control is in an accepting state. The path from the root to such an accepting leaf will be a description of the corresponding accepting computation.

Ideally we would like to show that such computation trees must in fact be generated in order to solve P-complete problems, thus proving $P \neq NP$. (For example, if we consider Tms with trees as inputs, which are read in one symbol at a time, we measure the size of the input actually read in, and our problem is to find whether or not some leaf has a given property; then it is easy to show that every deterministic machine has to traverse the input tree in some fixed order and therefore take exponential time. That is, the best deterministic

strategy is to generate the computation tree.) Failing that, it is still interesting to study properties of such trees. So let us look at trees which come from this kind of rules, and ask what they mean computationally. We saw that given a tree (i.e. one node follows from the next by M's rules) the trees that satisfy

$\exists$ an accepting leaf

represent languages in NP. The next question to ask is, what do trees S that are characterized by

(*) S's leaves satisfy predicate R( )

or even

S satisfies predicate R'

represent?

The simplest predicate after $\exists$ is: "there are k". We may extend the question to: there are at least k, no more than k, at least half, etc. The nice fact is that such machines recognize exactly the k-NP languages defined earlier.

Lemma: If the predicates above are polynomial time computable, the languages defined by

{x| $\exists$ Tm M and the computation tree of M on x has property (*)}

are recognizable within polynomial tape by a Tm.

Proof: Using polynomial space we can generate the tree, in a depth-first manner, visit each leaf and test for the property.

Definition: A (polynomially bounded) threshold Tm recognizable language is the set

{x| Tm M such that the computation tree of M on x has at least k accepting leaves}

where k may be a fixed constant or fraction. M is called a k-threshold Tm.

Our previous remarks imply

Lemma: a) threshold languages $\subseteq$ PTAPE

b) NP $\subseteq$ threshold languages

c) co-NP $\subseteq$ threshold languages

d) k-NP problems $\subseteq$ threshold languages.

co-NP denotes the languages whose complement is in NP, and they are recognizable by the predicate "no leaves accepting" (or equivalently, "all leaves nonaccepting"). Fact d) follows from Theorem 4.2, the interreducibility of k-NP problems and the observation that each accepting leaf corresponds to an accepting computation.

Gill [19] defined probabilistic time bounded Tms as Tms that in some configurations could probabilistically choose between two equiprobably moves. The machine accepts a string if there is a probability $p > 1/2$ that there will be an accepting computation within the desired time bound.

Theorem 4.4: Probabilistic polynomial time bounded languages and polynomial time bounded threshold Tm recognizable languages are the same.

Proof: a) To show that every probabilistic language can be accepted by a threshold Tm operating within the same time bound, it suffices to take the probabilistic Tm, and modify it in such a way that all its moves are probabilistic. This is easily

accomplished by adding extra states: if $\delta(q,a) = (p,b,d)$ is a deterministic move, define a new state p', identical to p, and make $\delta(q,a) = \{(p,b,d),(p',b,d)\}$. Doing this for every move, and applying the transformations simultaneously to identical states p, p', we have, after a finite number of steps, a modified machine M' that accepts the same language and has no non-probabilistic moves. Now consider the nondeterministic machine M" that has the same transition rules as M' (without probabilities attached to them). Let M" be bounded by the same polynomial as M' is. Then M' accepts a string with probability greater than 1/2 iff more than half of its possible computations within the time bound end in an accepting configuration; that is, (since M' and M" have identical computations) iff the threshold Tm M" with threshold function "more than half the leaves accept" accepts the string. Thus, probabilistic Tms are the special case of threshold Tms when the computation tree is a complete binary tree and the threshold function is "more than half of the leaves accept."

b) To prove the other direction, it suffices to show that every threshold machine may be simulated, at polynomial cost, by a threshold Tm with the characteristics above. This will be done by adding enough padding to the computation tree so that it will become a > 1/2 threshold tree.

We shall give separate proofs for the case k a constant and k a fraction. We shall show first that k accepting nodes can be turned into 1/2 of the nodes.

1) k a constant. Let T (T(|x|)) be the bound on the time used by M -- i.e. after T steps k of the leaves accept. Clearly we may pad M's instructions, so that the total number of leaves is 2^T (2^T - k non-accepting). Now we modify the padded machine so that its first new move results in two subtrees: the right subtree is identical to the old computation tree: the left subtree uses the input only for computing T and produces, after T steps, 2^T - k accepting leaves and k rejecting leaves. Thus, the total number of accepting leaves is greater than or e ual to half of the leaves in the new model if and only if at least k leaves of M were accepting. It is easy to see how to obtain >, <, or $\leq$ from $\geq$; =, i.e. the condition that exactly k leaves accept, is mapped into "exactly half of the leaves accept" and we shall see in the second half of the proof how to reduce that to the predicate "more than half of the leaves accept".

2) k is a fraction. Let k = p/q.

a) The condition is $\geq$. We have a tree (binary but not necessarily complete) and we accept whenever at least a fraction p/q of the leaves accept. We first transform p/q into 1/2, then the tree into a complete tree. Let each accepting node produce q - p accepting leaves, and each rejecting node produce p rejecting offsprings (this takes time $\log_2(\max(p,q-p))$, a constant). Then exactly half of the leaves accept in the new model iff exactly p/q leaves accepted in the old one, and if there was an excess over p/q of the accepting nodes in the old model, there will be more than 1/2 of the leaves accepting in the new one.

To make the tree binary, just add dummy transitions whenever the machine was deterministic that subdivide in such a way that exactly half of their offspring are accepting. Clearly leaves originating from such nodes will not influence the percentage of accepting leaves.

b) The condition is = (i.e. exactly half of the leaves accept). The problem, of course, is that we do not accept any excess over a half, and the techniques we saw were all monotone (linear). Get the complete tree of depth T that accepts by "exactly half" and build the following tree: the root generates two subtrees, both complete binary trees of depth 2T. The first simply has 3/4 of its leaves accepting and 1/4 rejecting, independently of the input. The second starts by imitating the computations of the old machine. After T steps this procedure terminates, the fact of whether the input was accepted by it or not is recorded in the finite control, and each node restarts the whole computation. After T more steps, the input is finally accepted if exactly one of the two attempts ended in an accepting configuration. The number of accepting leaves in the subtree is at most 1/4 of its leaves, and this happens iff exactly 1/2 of the leaves of the original tree accepted. This concludes the proof of the theorem.

Let us recapitulate what we did: we defined polynomially bounded threshold machines, showed that they contain NP and languages that have their complement in NP (co-NP), that they recognize the sets defined in the previous section, and that probabilistic polynomially bounded Tms are the special case of

threshold "more than half of the nodes". We also showed that this particular special case is sufficiently powerful to recognize any language recognized by any of the other models. We also showed that the power of such machines is bounded above by the power of polynomial tape bounded Tms. It is very difficult to get much more information about the class: for one thing, proof that any of these containments is proper would immediately imply that P ≠ PTAPE. Some easy additional properties are:

Lemma: The class of k-NP languages is complete for threshold machines.

Proof: follows immediately from Theorem 4.2 (since every accepting leaf in the tree corresponds to a different satisfying assignment). and the lemma that precedes Theorem 4.4.

As a consequence, all "customary" implications follow, e.g. if k-SATISFIABILITY is in P (NP, LOGTAPE), then every polynomial time bounded threshold machine can be simulated by some machine in that class. It is also clear that

Lemma : The class of k-NP languages is closed under complement.

Proof: Wlog assume the threshold to be > 1/2. Then the complement is accepted by the same machine (with accepting and nonaccepting states interchanged) with threshold $\leq$ 1/2.

In particular, NP = k - NP implies that NP is closed under complement. We do not know whether k-NP is closed under nondeterminism (projection). This may sound paradoxical, at first, since threshold machines may simulate nondeterministic

Tms without loss in efficiency -- but of course this does not imply that threshold machines may simulate nondeterministic threshold machines (where the machine would have a nondeterministic choice of what pair of transitions to choose). This model is better explained in terms of iterative arrays: we shall sketch it and come back to the discussion.

First, let us interpret our tree as represented Tms operating in parallel. The model can easily be formalized by defining Tms which may split into two, and each of these continue its computation independently of the other. In fact, we may assume that the machine has to describe the new machine completely (so that no "cheating" is done by getting large amounts of computing power too easily). It is clear that in this model nondeterminism corresponds to the model in which the only interaction between parallel processes is the OR-ing together of a single bit at the end of the computation (the bit is 1 iff the machine accepts) and threshold machines correspond to adding the bits up, and comparing the result with the threshold. The latter seems to demand somewhat more interaction, but we could not formalize this idea. It is unclear whether other models, defined similarly by "small interaction" can be simulated by threshold machines (for example, "an even number of nodes accept"). On the other hand, if every machine is allowed to communicate freely with any other, the only limiting resource will be tape -- it is an interesting question whether all languages in PTAPE may be obtained by some predicate on the leaves which corresponds to "small interaction" at the end,

among the parallel machines. Savitch's construction [60] shows that small interaction in the whole tree is enough to get the tape bounded class, i.e. the program for parallel Tms:

```
root machine
     write down the initial configuration
     write down the final configuration
     guess a midpoint configuration
     generate two sons with inputs
        1) initial configuration
           midpoint configuration
      and 2) midpoint configuration
           final configuration
     wait until both sons report success, then accept.
general node
     if leaf then if the 2 arguments represent successive
                   I 's of the Tm computation, report
                   success fi
          else guess a midpoint configuration
               generate two sons, with inputs
                  1) argument 1
                     midpoint
                and 2) midpoint
                     argument 2
               report success if both sons report
               success fi
```

Clearly, requiring that this computation tree have all the leaves successful will ensure that the original Tm accepted the

input in the given tape bound.

Each of these machines can be simulated, at polynomial cost, by an array of finite automata, capable of describing other finite automata. In these terms we have

Lemma : An iterative array of self-reproducing nondeterministic finite automata accepts in polynomial time

a) P, if the array is a fixed n-dimensional structure and operates deterministically.

b) NP, if the array is a fixed n-dimensional structure.

c) PTAPE, if the array may have tree structure.

A colorful statement of the above is that biological reproduction-like structures seem much more powerful than fixed n-dimensional structures.

4.5 Polynomial bounded hierarchies

In the first part of this section we shall review the definitions of the arithmetical hierarchy of Kleene [40], [55] and of the polynomial hierarchy of Meyer and Stockmeyer [47].

As we pointed out before, there is a well-known analogy between nondeterministic computation and existential quantification [61]. Given a nondeterministic Tm T' and an input x we add, as a separate input, the string of choices y that T' makes in an accepting computation. We may then easily build a Tm T which operates deterministically and accepts exactly the same set as T', in the same number of moves, whenever y is (an encoding of) the series of choices that T' uses. T rejects if y does not represent an accepting sequence of choices of moves. Thus the set accepted by the nondeterministic machine is

$\mathcal{L}(T') = \{x \mid (\exists y)\ T(x,y)$ halts and accepts$\}$.

This motivates us to try to identify, somehow, "recursive" with "P" and "r.e."* with "NP", since a set A is r.e. if it can be expressed as

$$A = \{x \mid (\exists y)\ T(x,y)\},$$

T a recursive predicate.

The restriction that T be a predicate computable in P still gives us all r.e. sets, but if we replace y by

$$y: |y| < p(|y|)$$

where $|y|$ denotes the length of y and $p(\ )$ is a polynomial, fixed for any given set, we get exactly the languages in NP.

Unhappily the analogy is not string enough to enable us to prove the theorems corresponding to the facts that there exist sets that are r.e. but not recursive (i.e. NP $\supsetneq$ P) or that if a set and its complement are both r.e. the set is recursive (i.e. NP = co-NP $\rightarrow$ P = NP). The reason is that in both cases the classical proof uses an exponential amount of information for each string.

We may try to extend the analogy in another direction -- in logic we have Kleene's definition of the arithmetical hierarchy, the classes of sets Σ_i, Π_i, Δ_i defined as follows:

$\Delta_0 = \Sigma_0 = \Pi_0$ = the class of recursive sets.

A set B is Σ_{n+1} if it can be expressed as

$$B = \{x \mid (\exists y)\ (P(x,y))\}$$

where P is a relation computable in Π_n. Similarly C is Π_{n+1} if

---

* recursively enumerable

$C = \{x \mid (\forall y)\ R(x,y)\}$

where R is a relation computable in Σ_n. (By that we mean that the language of pairs (x,y) such that R(x,y) is true is a Σ_n set.)

D is a Δ_{n+1} set if it can be recognized by a Tm with a Σ_n oracle.

It is easy to see that Σ_1 sets are the r.e. sets while Π_1 sets are the sets whose complement is r.e. Other facts about the arithmetical hierarchy are:

1) Σ_n sets (Π_n sets) are exactly the ones which may be represented as

$$\{y \mid Q_n x_n Q_{n-1} x_{n-1} \ldots Q_1 x_1 P(x_1 x_2 \ldots x_n y)\}$$

where P is recursive, each Q_i is $\exists$ or $\forall$, Q_n is $\exists$ (is $\forall$ for Π_n) and the quantifiers alternate, i.e. $\forall$ is always followed by a $\exists$ and a $\exists$ by a $\forall$.

2) The complement of a Σ_n set (Π_n set) is a Π_n set (Σ_n set).

3) A is Σ_{n+1} iff it is r.e. in a Π_n set B.

The fact that makes these classes of sets very interesting is the following hierarchy theorem:

Theorem 4.5: For all $i > 0$ there are languages A_i, B_i, C_i, D_i such that

$A_i \in \Sigma_i - \Sigma_{i-1}$

$B_i \in \Pi_i - \Pi_{i-1}$

$C_i \in \Sigma_i - \Pi_i$

$D_i \in \Pi_i - \Sigma_i$.

This says that the classes Σ_i and Π_i are incomparable, while the Σ_i and Π_i form an increasing collection of sets with increasing i. It is also true that for all i,

$$\Sigma_{i+1} \supsetneq \Pi_i$$
$$\Pi_{i+1} \supsetneq \Sigma_i .$$

It also implies that the class

$$A = \bigcup_i \Sigma_i = \bigcup_i \Pi_i$$

properly contains any Σ_i.

One way of looking at this hierarchy is as an attempt to close the recursive sets under the operations of $\exists$ (projection) and $\neg$ (negation), and the existence of the hierarchy as a statement of the fact that no finite amount of such closure operations will be enough.

Now, we saw that if we restrict our predicates to deterministic polynomial time computable predicates and quantification to polynomial length bounded quantification, we can define P and NP in a similar way. We may extend this analogy [47]: Let

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = P$$

$$\Sigma_{i+1}^P = \text{class of languages definable as } \{y \mid \exists^P x\ R(x,y),\ R \text{ in } \Pi_i^P\}$$

$$\Pi_{i+1}^P = \text{class of languages definable as } \{y \mid \forall^P x\ S(x,y),\ S \text{ in } \Sigma_i^P\}$$

where $\exists^P x$ ($\forall^P x$) means that there is a polynomial $p(\ )$ such that the quantification ranges only over strings x with $|x| \leq p(|y|)$.

Δ^P_{i+1} = class of sets definable by

{y| y is accepted by a polynomial bounded Tm with a Σ^P_i oracle}.

Then Σ^P_0 sets are languages in P, Π^P_1 sets are languages in NP, and Π^P_1 sets are languages whose complement is in NP. Also facts 1-e (with "nondeterministic polynomial computable in" substituted for "r.e. in") hold for the collection of classes Σ^P_i, Π^P_i.

Many other such theorems may be proven by analogy. It is, however, not known whether there is a hierarchy theorem analogous to Theorem 4.5. Such a theorem would certainly imply $P \neq NP$, but it is actually a stronger conjecture: all that is known is that

$$\Sigma^P_{i+1} \supseteq \Sigma^P_i \cup \Pi^P_i \qquad \Delta^P_i \subseteq \Sigma^P_i \cap \Pi^P_i$$

$$\Pi^P_{i+1} \supseteq \Pi^P_i \cup \Sigma^P_i \qquad \Delta^P_{i+1} \supseteq \Sigma^P_i \cup \Pi^P_i$$

and if these inclusions become equalities at any level, they will collapse at every higher level; i.e., if, for example, $\Sigma^P_{i+1} = \Sigma^P_i$, then $\Sigma^P_{i+k} = \Sigma^P_i$ for all $k \geq 0$, and $\Pi^P_{i+\ell} = \Pi^P_{i+1}$ for all $\ell \geq 1$; moreover, $\ell^P_{i+1} = \Sigma^P_i$. Also, $\Pi^P_i = \Sigma^P_i$ implies that $\Sigma^P_{i+1} = \Sigma^P_i$. So it is quite possible that $P \neq NP$ but the hierarchy collapses at some finite point. Questions about these classes are thus extremely hard to answer in general, but this might be a fruitful approach to the problem P =? NP.

It is also natural to ask what happens if we consider languages of the form

$$L = \{y|\ Q^P_n x_n Q^P_{n-1} x_{n-1} \cdots Q^P_1 x_1\ R(x_1, \ldots, x_n, y)\}$$

where, as before, the Q_i^p are polynomial bounded quantifiers, R is a deterministic polynomial time computable predicate, but n depends on y. If the dependence is arbitrary, one may obtain big recursive sets, by using so many variables x_i that R becomes non-polynomial in x. We may prevent this by requiring that $n \leq p(|y|)$ where p is a polynomial. Let H be the class of all such languages. We have then the rather surprising result

Theorem 4.6: H = PTAPE.

Proof: It is not hard to prove that PTAPE $\supseteq$ H, by direct programming: one reserves a slot for each variable. Since the quantification is polynomial bounded, a polynomial space will suffice for each slot; moreover, since there are at most a polynomial number of quantifiers the storage required is polynomial. Now verify the formula nondeterministically: for every $\forall^p$ quantifier, run through all possible values, for every $\exists^p$ quantifier guess a value for it. Verify that R is true (since R may be checked in polynomial time it uses at most polynomial tape for every set of inputs).

The proof of the containment H $\supseteq$ TAPE is similar to Savitch's proof that PTAPE = NPTAPE [60]. It is easy to write an expression $A_0(u,v)$ which denotes the predicate which is true iff u and v are IDs of a given polynomial tape bounded Tm T on a given input x, and v follows from u in at most one move. Now define

$$A_{i+1}(u,v) = (\exists^p w)(\forall^p y)(\forall^p z)$$
$$[(((u = y) \wedge (w = z)) \vee ((w = y) \wedge (v = z))) \Rightarrow A_i(y,z)] .$$

where, as before, the Q_i^p are polynomial bounded quantifiers, R is a deterministic polynomial time computable predicate, but n depends on y. If the dependence is arbitrary, one may obtain big recursive sets, by using so many variables x_i that R becomes non-polynomial in x. We may prevent this by requiring that $n \leq p(|y|)$ where p is a polynomial. Let H be the class of all such languages. We have then the rather surprising result

Theorem 4.6: H = PTAPE.

Proof: It is not hard to prove that PTAPE $\supseteq$ H, by direct programming: one reserves a slot for each variable. Since the quantification is polynomial bounded, a polynomial space will suffice for each slot; moreover, since there are at most a polynomial number of quantifiers the storage required is polynomial. Now verify the formula nondeterministically: for every $\forall^p$ quantifier, run through all possible values, for every $\exists^p$ quantifier guess a value for it. Verify that R is true (since R may be checked in polynomial time it uses at most polynomial tape for every set of inputs).

The proof of the containment H $\supseteq$ TAPE is similar to Savitch's proof that PTAPE = NPTAPE [60]. It is easy to write an expression $A_0(u,v)$ which denotes the predicate which is true iff u and v are IDs of a given polynomial tape bounded Tm T on a given input x, and v follows from u in at most one move. Now define

$$A_{i+1}(u,v) = (\exists^p w)(\forall^p y)(\forall^p z)$$
$$[(((u = y) \wedge (w = z)) \vee ((w = y) \wedge (v = z))) \Rightarrow A_i(y,z)] .$$

In other words, $A_i(u,v)$ is true iff there is a sequence of valid transitions of the Tm T which lead from the configuration denoted by the ID u to the one of ID v in at most 2^i steps. Now, if T is n^k tape bounded, then it accepts x if and only if

$$A_{n^k}(i,f)$$

is true, where i is the initial configuration with x as input, and f an accepting configuration. Since $A_{n^k} \in H$, we are done.

Again, in recursive function theory, the class corresponding to H properly contains the arithmetical hierarchy. The similar result is of course not known to be true for polynomial classes. The results above are due to Meyer and Stockmeyer [48].

The Meyer-Stockmeyer hierarchy may be viewed as an attempt to close the class of deterministic polynomial time computable functions under the operations of $\exists^p$ (nondeterminism) and (complement). In this case, the previous theorem shows that a polynomial number of operations is sufficient. However, it may be true that the hierarchy collapses much sooner.

Lemma : If NP = co-NP (P = NP) and the reduction is very efficient, then NP = PTAPE (P = PTAPE).

By very efficient we mean that if L's complement is accepted by a nondeterministic Tm in time p(n), then L is accepted by a deterministic Tm in time p(n) + c.

The proposition follows from the normal form of the previous theorem: we just eliminate each quantifier at a constant cost.

The same sort of hierarchy could be defined for threshold machines, with the alternating quantifiers being threshold and

existential quantifiers, or simply successive threshold quantifiers. Again, a polynomial number of quantifiers is necessary to describe PTAPE and an efficient closure property would ensure that PTAPE = k - NP.

Unfortunately we were unable to prove any relationships between the two hierarchies (except that $\Delta_1^P \subset \exists_k$). We conjecture that they are in fact independent.

Let us now present a hierarchy of nondeterministic exponential time bounded sets that orders the elementary sets. This will be done by a generalization of the polynomial hierarchy by using quantifiers over polynomially bounded objects of higher type, in a manner analogous to the definition of the analytical hierarchy generalizing the arithmetical hierarchy.

We build up some machinery in order to present our result.

Definition: A set of strings, A, is polynomially bounded by y iff $\exists p(\ )$, $p(\ )$ a polynomial, $\forall x \in A \quad |x| \leq p(|y|)$.

A string is an object of type 0. An object of type i+1 is a collection of objects of type i. For i > 1 we say that an object of type 1 is polynomially bounded iff the objects that it is a collection of are polynomially bounded.

The functions $t_i(n)$ are defined by

$$t_0(n) = n \ , \quad t_{i+1}(n) = 2^{t_i(n)} \ .$$

Thus $t_i(n)$ stands for i levels of exponentiation. It is well known that the elementary languages are exactly the languages in

$$\bigcup_{i=1}^{\infty} \mathrm{DTIME}[t_i(n)]$$

and it is easy to show that

$$\bigcup_{i=1}^{\infty} \text{NTIME}[t_i(n)] = \bigcup_{i=1}^{\infty} \text{DTIME}[t_i(n)]$$

since

$$\text{NTIME}[t_i(n)] \subseteq \text{DTIME}[t_{i+1}(n)].$$

Also, for all polynomials p( ), for sufficiently large n,

$$t_i(p(n)) < t_{i+1}(n)$$

We saw in the preceding pages that NP (i.e.

$$\bigcup_{p \text{ a polynomial}} \text{NTIME}[t_0(p(n))])$$

can be characterized as the class of languages L which could be expressed as

$$L = \{y \mid \exists x: |x| < p(|y|) \text{ and } R_L(x,y)\}$$

for some predicate $R_L \in P$.

Our result is an extension of this characterization to the classes

$$S_i = \bigcup_{p \text{ a polynomial}} \text{NTIME}[t_i(p(n))]$$

namely, that any language $L \in S_i$ may be obtained by a single polynomially bounded existential quantification of a simple predicate. In the case i = 0, the quantifier ranges over polynomially bounded objects of type 0 and the predicate is computable in deterministic polynomial time, while in the general case, to describe a language $L \in S$, we quantify over polynomially bounded objects of type i, and allow a constant number of polynomially bounded quantifiers over objects of type at most i-1. In addition, the matrix (the quantifier-free portion of the formula) becomes a fixed boolean combination of deterministic polynomial time computable predicates and membership

predicates -- or, in other words, if we make the convention that predicates of the form "u ∈ V" may be evaluated in unit time, the matrix is deterministic polynomial time computable in this new sense.

Let us use $\exists^p_i x$ to denote "there is a polynomially bounded object x of type i". With these conventions, our main result is:

Theorem 4.7: A language L is in the class S_i (= NTIME[p(n)]) if and only if it may be written as

(**) $L = \{y \mid \exists^p_i z \exists^p_{n_1} x_1 \ldots \exists^p_{n_k} x_k \; R_L(y,A,x_1,\ldots,x_k)\}$

where $n_j < i$ for $j = 1,2,\ldots,k$, k and R_L are fixed for each language L, and the predicate R_L is a deterministic polynomial time computable predicate (in the sense just defined above).

Proof: First, let us prove the special case i = 1.

Let $L \in \text{NTIME}[2^{p(n)}]$, i.e. S_1. For clarity, let us assume that p(n) = n -- it will become clear that our argument does not use the fact that we use this particular polynomial, but the formulas are somewhat cleaner. For every $y \in L$, we shall describe an accepting computation of the Tm M that recognizes y, i.e. our predicate will be

$L = \{y \mid \exists X$: X represents an accepting computation of M on input y taking exponential time at most$\}$.

X will be the set of ID's of M's computation (i.e. a description of M's tape -- including the square scanned and the state of the finite control). The only problem is that the tape used may be of exponential length and our quantifiers

(except for $\exists X$) range over strings of polynomially bounded length.

We overcome this difficulty by representing each ID as 2^n elements, where each element stands for a single square of the Tm tape. Each element will be a triple (i.e. a string with the components separated by markers) of the form

time # position # square.

(t,p,s) will represent the contents of the p-th square of the Tm tape at time t. We assume the tape to be one-way infinite (so that position is well-defined) and if at time t the Tm is scanning the p-th square in state q, we shall represent the contents of that square as the pair (σ,q), where σ is the symbol that the square contains. It is easily seen that the length of t and p will be polynomially bounded.

In order to ensure that X is the set of ID's as claimed, we shall write a predicate expressing the fact that for all strings $w \in X$,

a) w is of the form (t,p,s) as described above (ensured by the formula A defined below)

b) for all times we have a complete description of M's ID (and only one) denoted by B

c) the set of such ID's is an accepting computation of M on input y, guaranteed by C.

Detailed descriptions of each of these are:

$A = \forall^p w \in X \ \exists^p t \exists^p p \exists^p s$ w = t#p#s and $s \in \Gamma$ where Γ = {M's tape alphabet} $\cup$ {M's tape alphabet} $\times$ {M's state set}

Actually $\exists^p s$ is not necessary -- one could simply write out

all possibilities for s since Γ is finite -- we used it only for conciseness of exposition. We shall use this abbreviation throughout, remembering that quantification over s is not essential.

$$B = \forall^p t \forall^p p \ (t \leq 2^{cn} \wedge p \leq 2^{cn}) \rightarrow \exists s \exists^p w \ [w = t\#p\#s \wedge w \in X]$$
$$\wedge \ \forall^p t \forall^p p \forall^p w \forall^p w' \forall s \forall s' \ [w = t\#p\#s \wedge w' = t\#p\#s'] \rightarrow w =$$

c) An accepting computation is characterized by

c1) at time t = 0 we have the initial ID of M with input y

c2) at time $t = 2^{cn}$ M is in an accepting state

c3) if a configuration is assumed at a time t > 0, it is attained as the result of a sequence of legal moves from the initial configuration. Such descriptions of accepting computations were used by Stockmeyer and Meyer [47] for the study of lower bounds.

c1) is ensured by

$$c1 = 0\#1\#y_1 \times q_i \in X \wedge 0\#2\#y_2 \in X \ \wedge$$
$$0\#3\#y_3 \in X \wedge \ldots \wedge 0\#n\#y_n \in X \wedge$$
$$\forall^p p \ (p > n \wedge p \geq 2^{cn}) \rightarrow 0\#p\#b \in X$$

where the input $y = y_1 y_2 \ldots y_n$, q_i is the initial state of M, and b denotes the symbol for blank in M's tape alphabet.

c2): $c2 = \exists^p w \exists^p t \exists^p p \exists s \ [w \in X \wedge w = t\#p\#s \wedge s = a \times q_f]$ where a stands for some symbol in M's tape alphabet and q_f denotes the unique final state of M, such that M upon entering q_f accepts and stops (it can be assumed wlog that M has such behavior).

c3): To make the expressions easier, let us assume that

M may either rewrite a square or move its head but not both. This causes the running time to double at most. Now (t,p,s) is part of M's configuration at time t if

(1) s does contain a state component (s = a × q), at the previous moment M was either scanning the p-th square in state q', read b on it, and one of the valid moves of M upon reading a b in state q' is to rewrite it as an a and go to state q; or scanning one adjacent square, and an allowable transition was to move the head to the p-th square.

(2) s does not contain a state component (s = a) and at the previous move either M was scanning the p-th square, reading a, and its state caused the head to move; or M was scanning another square and did not move onto the p-th square, which at time t-1 contained a.

We must, in addition, require that only one move was made (e.g. if $\delta(q,a) = \{(c,q',0),(a,q'',L),(a,q''',R)\}$ then it is not the case that $(t,p,q \times a) \in X$ and more than one of $(t+1,p,q' \times c),(t+1,p-1,b \times q''),(t+1,p+1,q''' \times d) \in X$. It can be seen easily that this requirement is satisfied (given our previous restrictions) by

(3) at all times exactly one square has a state specified in it.

The expressions that stand for the facts above are:

$C3(1) = \forall^p t \forall^p p \forall s \in \Sigma \times Q, 0 < t \leq 2^{ct}, t\#p\#s \in X \rightarrow$

$[s = a \times q \rightarrow ((t-1\#p\#b \times q' \in X \wedge (a \times q, s) \in \delta(b,q')$

$\vee (\exists c,q' \; c \in \Sigma, q'' \in Q, (q,R) \in \delta(c,q'')$

$\wedge t-1\#p-1\#c \times q'' \in X \wedge t-1\#p\#a \in X)$

$\vee (\exists d,q''' \; d \in \Sigma, q''' \in Q, (q,L) \in \delta(d,q''')$

$\wedge t-1\#p+1\#d \times q''' \in X \wedge t-1\#p\#a \in X))]$

$C3(2) = \forall^p t \forall^p p \forall s \in \Sigma, 0 < t \leq 2^{ct}, t\#p\#s \in X \rightarrow$

$[s = a \rightarrow [(t-1\#p\#a \times q) \in X, q \in Q \wedge [\delta(a,q) \ni (a,q',R)$

$\wedge (t-1\#p+1\#c) \in X \wedge (t,p+1\#c \times q') \in X]$

$\vee [\delta(a,q) \ni (a,q'',L) \wedge (t-1\#p-1\#d) \in X$

$\wedge (t\#p-1\#d \times q'') \in X]]$

$\vee [(t-1\#p\#a) \in X \wedge ((t-1\#p\#b) \in X$

$\vee (t-1\#p-1\#b \times q') \in X \wedge$

(did not move to p) $(t\#p-1\#c \times q'') \in X, q'' \in Q, c \in \Sigma \vee$

$(t\#p-2\#c') \times q''') \in X, q''' \in Q, c' \in \Sigma')$

$\vee ((t-1)\#p+1\#b) \in X \vee ((t-1)\#p+1\#c \times q') \in X \wedge$

$((t\#p+1\#c'' \times q'') \in X \vee (t\#p+2\#c''' \times q''') \in X))]$

$C3(3) = \forall^p t \exists^p p \forall^p p', \exists a \times q \in \Gamma$

$(t\#p\#a \times q) \in X \wedge ((t\#p'\#s) \in X \wedge p \neq p' \Rightarrow s \in \Sigma).$

Now the language L may be expressed as

$L = \{y \mid \exists^p X \; A \wedge B \wedge C1 \wedge C2 \wedge C3(1) \wedge C3(2) \wedge C3(3)\}.$

It is clear that each of A,B,...,C3(3) contains only predicates testable in polynomial time (remember that $w \in X$ is counted as a unit operation).

Also, only 4 quantifiers were used -- if we count alternation of quantifiers, only 3 have been used. This proves that every language recognizable in $NTIME[2^{p(n)}]$, p( ) a polynomial,

has the claimed representation.

Some comments about the encoding:

1) Non-alternating quantifiers may be collapsed into one, at a slight cost of the polynomial bound (both the time necessary to evaluate the predicate and the bound on string length go up). For example, $\exists^{P}x\, \exists^{P}y\ R(x,y)$ is equivalent to $\exists^{P'}w\ \ w = P(x,y) \wedge R(H(w),T(w))$ where $P(\ )$ is a polynomial time computable pairing function and $H(\ )$ (head) and $T(\ )$ (tail) are the first and second projection functions respectively, and p' is a polynomial that bounds the length of $P(x,y)$ if the lengths of x and y are both bounded by $p(\ )$.

The argument above holds for polynomially bounded quantifiers over higher type objects.

2) The string quantifier prefixes we used were all of the form $\forall\exists$, except for C3(3) which is $\forall\exists\forall$. However, C3(3) may be rewritten into the form $\forall\exists$ by stating

$\forall t \exists p$ [at time t the p-th square is scanned]

$\forall t \forall p \forall p'$ [if at time t the p-th and p'th squares are scanned then $p = p'$].

Let us tackle the general case now. We want to show that all the languages acceptable by a Tm in nondeterministic time $2^{\cdot^{\cdot^{\cdot^{2^{p(n)}}}}}\}k$ (for inputs of length n) can be expressed by a predicate (**), which contains a single polynomially bounded existential quantifier over objects of type i, and a fixed finite number of polynomially bounded quantifiers over objects of type at most i-1, and the predicate preceded by these quantifiers is

computable in deterministic polynomial time (assuming all membership predicates to take unit time).

We shall proceed by induction. The case i = 1 has just been proved.

Lemma : The number of objects of type i polynomially bounded by p is: $2^{2^{\cdot^{\cdot^{\cdot^{2^{p(|y|)}}}}}}$ (i 2's)

$n_i = 2^{2^{\cdot^{\cdot^{\cdot^{2^{n}}}}}}$ (binary alphabet) .

Proof: for sets $n_1 = 2^{p(|y|)}$. The number of objects of type i+1 is the cardinality of the power set of the objects of type i, i.e. 2^{n_i}.

Our approach to the general case will be analogous to the case of exponential time: we shall have an object X, of type i, representing an accepting computation of the Tm M (that recognizes the language within the given bound) on input y. Again, we shall want to describe an ID as a triple (time, position, square) and write down the predicates A - C3(3) that will ensure that X indeed represents such a sequence. The problem is that the first two components now have length

$$2^{2^{\cdot^{\cdot^{\cdot^{2^{p(n)}}}}}} \text{ (i-1 2's)} = t_{i-1}(p(n))$$

and all strings have polynomially bounded length.

The trick we will use is to use encodings for the numbers, just as we used encodings for the ID's. Intuitively, the numbers "stay one level below" so that the procedure may be carried out inductively.

More precisely, we shall prove that with quantifiers of

type i-1 we may obtain objects of type i-1 that will represent numbers. Also, the operations of forming pairs and triples of such objects can be described within the same formalism, and we may express the relations

$a = b$

$a = 0$

$a = b+1$

a is the first (second) component of a pair $w = (a,b)$ among objects a, b, w representing numbers up to $t_i+1(n)$, using a finite number of quantifiers of type at most i-1, over polynomial time bounded predicates.

It is only a matter of going over our previous proof (for the case i = 1) to check that only the predicates above were used to write down formulae A - C3(3), so that if we prove the claim above we shall have proven the theorem.

Lemma : If A and B are polynomially bounded objects (pbos) of type i, and $x \in \Gamma$, there are pbos of type i (perhaps with a higher bound) C and D such that

1) C represents (A,B)

2) D represents (Λ,x).

Moreover, the predicates $C = (A,B)$, $(D = (\Lambda,x))$

x = first (second) component of C (D)

are expressible using polynomially bounded quantifiers of types at most i-1 (except for i = 0 when an existential quantifier is needed for the last one).

Proof: By induction on i.

For i = 0, $C = \Lambda\#B$, $D = \Lambda\#x$; $w = \Lambda\#B$ is deterministic

polynomial time computable, as is the predicate $(\ ,y)[w = x\sharp y]$.

For $i > 0$, $A = \{a_j\}$, $B = \{b_k\}$ where a_j, b_k are objects of type i-1. By induction $c_{jk} = (a_j,b_k)$ is defined, as well as $d_j = (a_j,x)$. Define $C = \{c_{jk}\}$, $D = \{d_j\}$. Now $C = (A,B)$ iff $\forall w \in C$, $w = (a_j,b_k) \wedge a_j \in A$, $b_k \in B$ and

$A = \{a_j \mid \exists b_k\ [b_k \in B \wedge (a_j,b_k) \in C]\}$.

This lemma justifies the use of triples at all levels. Now let us define representations for numbers. For $i = 0$ we represent a number by writing it down. Since our numbers were bounded by 2^{cn} (remember the simplifying assumption that our polynomials were linear), this could be done in polynomial space.

For $i = 1$, a number will be represented as a set {(position,digit)}, with the obvious meaning: position will be a string of polynomial length. This will enable us to write $\sim 2^n$ positions (i.e. our numbers will have length $\sim 2^n$) and represent values up to 2^{2^n}.

In general, we will assume as inductive hypothesis that the representation of numbers up to $t_{i+1}(n)$ (i.e. of length $t_i(n)$) can be done using an object T_i of type i ($i \geq 1$) and the elements of T_i are of the form (p_{ij},digit) where p_{ij} represents a number (the j-th position of the number represented by T_i) and digit is the corresponding digit. Moreover, we assume that the set T_i may be defined using a single existential quantifier of type i (and quantifiers of smaller types). (Since we shall use T_i in a formula which has a quantifier of type i+1, this will cause no problems.) Then define numbers of length $t_{i+1}(n)$

as sets $\{(T_i,\text{digit})\}$ where T_i will denote a position. In order to get numbers we must

a) ensure that all digits are defined.

b) be able to deal with these objects as numbers $\leq t_{i+2}(n)$,

i.e. i) test for $=0$

ii) test for $=t_{i+2}(n)$

iii) test for equality of two "numbers"

iv) test for the successor relationship

v) test for $<$.

Note that if we can prove that a) and b) can be done using quantifiers of type i+1 at most, we not only shall have proven that we can define numbers but will have proven our theorem, since the predicates b)i) - b)iv) are exactly those needed for the proof of the main theorem.

<u>basis</u> (i=1)

a) $(\forall^p z)[(z,0) \in T_\ell \vee (z,1) \in T_\ell \wedge \neg((z,0) \in T_\ell \wedge (z,1) \in T_\ell)]$ ("T_ℓ is a number")

b) i) $T_i = 0 \overset{\text{def}}{=} (\forall^p z)\,[(z,0) \in T_i]$

ii) $T_i = t_i(n) \overset{\text{def}}{=} (\forall^p z)\,[(z,1) \in T_i]$

iii) $T_i = T_j \overset{\text{def}}{=} (\forall^p z)\,[((z,0) \in T_i \leftrightarrow (z,0) \in T_j)]$

iv) $T_i = T_j + 1 \overset{\text{def}}{=} (\exists^p z)\,[(z,1) \in T_i \wedge (z,0) \in T_j \wedge (\forall^p z')\,[z' < z \rightarrow ((z',0) \in T_i \wedge (z',1) \in T_j) \wedge (z' > z \rightarrow ((z',0) \in T_i \leftrightarrow (z',0) \in T_i))]$

(i.e. there is a position z which becomes 1 -- all digits higher than z remain the same, while the

digits below change from 1 to 0. In T_j all digits up to the z-th were 1's.)

v) $T_i > T_j$ $(\exists^p z)$ $[(z,1) \in T_i \wedge (z,0) \in T_j \wedge (\forall^p z')$ $[z' < z \rightarrow ((z',0) \in T_i \longleftrightarrow (z',1) \in T_j)]$.

This concludes the proof for i = 1. Let us suppose the theorem holds for i.

induction (i+1)

a) $(\forall_i^p z)$ $[((z,0) \in T_\ell \vee (z,1) \in T_\ell) \wedge$ $((z,0) \in T_\ell \longleftrightarrow (z,1) \notin T_\ell)]$.

b) it is easy to see, in a similar way, that i) - v) still hold -- of course whenever we use < for example, we shall have to substitute the appropriate formula, but by the induction hypothesis such a formula exists and has quantifiers of type i at most.

This completes the proof of the existence of number representations. It also shows that <, =, +1, etc. may be defined for objects representing numbers smaller than $t_{i+2}(n)$, and such a formula may be written by using quantifiers of type i at most.

Now we may substitute these number representations into formulae A - C3(3) of our proof of the representation theorem for i = 1, and obtain the theorem for general i, i.e.

if a language L is accepted by a nondeterministic Tm within time $t_i(n)$, it may be written as

$$L = \{y \mid \exists_i^p X\ z\ Q_1^p x_1 \ldots Q_k^p x_k\ R_L(x_1,\ldots,x_k),z,y)\}$$

where R_L is deterministic polynomial time computable, and the Q_j^p's are polynomially bounded quantifiers over objects of type at most i-1.

It is fairly easy to show the converse, i.e.

Theorem 4.8: if a language L can be characterized by

$$L = \{y \mid \exists_i^p x \; \forall Q_1^p x_1 \ldots Q_k^p x_k \;\; R_L(x_1,\ldots,x_k),y)\}$$

then $L \in NTIME[t_i(p(n))]$ for some polynomial $p(\,)$.

Proof: we saw that the cardinality of an object of type i is $n_i \leq t_i(p'(n))$. The Tm M that recognizes L writes down (nondeterministically) the set X, in tape and time n_i. For the quantifiers of lower types, it acts as follows: if the quantifier is existential, it writes down a choice for the object; if it is universal it goes into a loop -- starting with the empty set, it tests whether the formula is true, if so adds a new element to the object (which this is possible) and repeats the procedure. If the formula is false, M rejects. If all possible values for the set satisfy the formula, the universal quantifier was checked. Since the predicate itself is computable in polynomial time, the validity of $a \in X$ can be checked by scanning the tape, and the enumeration of an object of type i-1 takes at most $t_i(n)$ steps, the running time of the nondeterministic procedure is bounded by $t_i(p(n))$ for some polynomial $p(\,)$, as claimed.

4.2 Approximate solutions

"Someone asked: 'If you form an idea about TT, it's only a more resemblance, isn't it?'

The Master said: 'Even if you don't form an idea, it's still a more resemblance'.

The Master said: 'You're a man who doesn't feel pain and itching, aren't you?'."

Te'ao-chan

This section is scarcely more than a comment on some recent work and a conjecture, motivated by our definition of parsimonious and transparent reductions.

When we presented the NP-complete problems, we stated that much of their importance is due to the fact that there are many "useful" combinatorial problems (knapsack, integer programming, clique, etc.) that are NP-complete. Thus, solving NP-complete problems is necessary in order to answer real life questions.

This last statement, however, is somewhat more than the truth: many of the NP-complete problems that arise from modeling real situations appear as minimization problems, where the user might be perfectly satisfied with an approximate "good" solution, instead of the absolute best. Also, data is often approximate, so that any solution must be approximate to within the error in the input. For all these reasons, it makes sense to ask for approximate solutions to problems in NP, that operate in deterministic polynomial time. There are quite a few recent papers that examine this problem: [43,10,12,36,29,26,] [17]. It turns out that in many cases very good approximations exist and one can bound the error of the approximation a priori. The intuitive reason is that one may sometimes eliminate very bad solutions at reasonable cost. As an illustration, we shall

present the SUBSET - SUM problem and its efficient approximation (from [36]).

Problem: given a set T of n positive rationals and a bound b, try to maximize

$$m(T) = \max_{T' \subseteq T} \{ \Sigma q \mid \Sigma q < b \} .$$
$$q \in T' \qquad q \in T'$$

The k-th approximation algorithm A_k is given by

1) Find the best possible solution among the "very big" elements by any (inefficient) method. By "very big" we mean $b/k+1$.

2) If the best solution is not equal to b, let y be the difference between b and the current sum. Find the biggest element in the set not yet in the solution and smaller than y. Put it in the solution and go to 2). (If no such element exists, halt).

It is not hard to prove that A_k will give better and better approximations with increasing k, while operating in deterministic n^k time, and that in general the ratio of the best solution to the solution found by A_k is at most $(k+1)/k$. Essentially this happens because a really bad solution must do very stupid things with "big" elements and step 1) prevents this from happening.

Many of the "good" approximation algorithms follow the same lines. One notable exception is Lin's traveling salesman algorithm, which is based on local optimization at a few randomly selected points.

Although many times the practical results obtained using these methods are quite impressive, and approximation algorithms

are certainly necessary, we claim that research in this area will not contribute to our understanding of the P = NP problem. The reason is that the original problems were difficult (at least apparently) because it seems necessary to examine many combinations in order to obtain a solution -- it is not necessarily true that this will still hold if we deal with approximate problems. In fact, algorithms A_k show exactly that doing a complete combinatorial search on a very small portion of the data will produce very good approximations to the knapsack problem. Of course, in order to define what we mean by a good approximation, we must impose some topological structure on the P-complete problem and speak about points "close" to the solution. Since this is done <u>after</u> the problem is shown to be P-complete, there is no reason to expect that "easily approximable" will have any invariant meaning (if, in the above problem, we chose for metric 2^{Δ} where $\Delta = b - \Sigma$ elements in solution, then instead of an asymptotic ratio $(k+1)/k$ we would have had a ratio of $O(\log n)$). Thus, trying to classify P-complete problems on the basis of their approximation properties [36],[59] is quite pointless if we want to get insight into P-completeness (just because 5 bitter lemons and 5 cars share the attribute "5", we should not discuss whether 5 is bitter). This is not to say that such studies are not useful for solving approximate P-complete problems -- in fact they may suggest that solving P =? NP may not be important after all.

Thus approximation problems as usually defined are useless to the general problem because

1) an approximation to one P-complete problem does not help us to solve other P-complete problems equivalent to it.

The reason is that

2) an approximation may be completely unrelated to the exact solution.

As an illustration of 2), consider the knapsack problem:

$T = G \cup B$,

where $G = \{1/2i \mid i = 0,1,\ldots,m\}$

$B = \{1/2i + \varepsilon \mid i = 0,1,\ldots,m\}$ $\varepsilon = 1/2^{2m}$

$b = 2 - 1/2^{m}$.

The correct solution uses all the elements of G (good) and none of B (bad). However, all algorithms A_k that do not examine all the elements of T in step 1) will include only elements from B. Of course the approximation is quite good in terms of the error committed, but the "good" approximation does not include any element of the correct solution. The reader should find it easy to confirm such behavior in the similarly modified P-complete problem [38] or in the "good" approximations of the literature. In the case of the traveling salesman problem, [69] shows that for the deterministic versions of the sort of algorithms used in [43], the suboptimal and truly optimal solutions must differ by many arcs, and this is the behavior of the worst-case examples in [56].

We propose a definition of approximation to the solution of combinatorial problems which does not suffer from 1) and 2) (although it does not express what "real-life" people would call an approximation). We shall be quite informal.

Let M be a polynomial time bounded nondeterministic Tm accepting an input of length n. Let y be (an encoding of) the sequence of nondeterministic choices that M made. Then y represents the nondeterminism, and knowledge of y would eliminate the necessity for it. Moreover, if we use transparent reductions, in P-complete problems y will be encoded into the solution to these problems. So it seems natural to define a solution to be good, if many of its bits are right. More precisely,

Definition: Let b be a boolean formula in the variables $x_1,\ldots,x_n$. Any assignment $\bar{x}_1,\ldots,\bar{x}_n$ will be an approximation to the satisfying assignment. For satisfiable formulas the goodness of the approximation is $f = k/n$, where $n-k$ is the minimal number of bits that must be changed in order to obtain a satisfying assignment.

Clearly, $0 \leq f \leq 1$, $f = 1$ denotes a solution, and good approximations to a boolean formula will yield a good approximation to any problem which is related to it by a transparent reduction. For example, in the case of the knapsack problem, it would ensure that some elements did belong to the solution, and in the case of the traveling salesman problem it would ensure that some arcs were indeed taken, etc. Thus, a good approximation in this sense is a good approximation to every natural P-complete problem.

As one would expect, no good approximation algorithms (in the sense above) are known; in fact

Lemma : There are effective good approximations to the satisfiability problem iff P = NP. By effective we mean that not only the approximation is given, but also which bits are guaranteed to be correct of a solution exists, and the procedure works in deterministic p(n) time.

Proof: trivial -- since it is easy to substitute the known values into the boolean expression and repeat the procedure. Every step (by assumption) takes a polynomial time for the guess and another polynomial for the substitution.

This does not exclude the fact that non-effective approximation algorithms could exist, and P ≠ NP. We exhibit an algorithm that operates in time n^k and is guaranteed to produce k correct bits. We shall consider the case when the formula is satisfiable (otherwise by definition, every procedure produces perfect approximations).

```
proc assign (b) returns x
   for i = 0 to k do
       for all assignments with exactly i 1's in them do
          if b(assignment) = 1 then
                                 return(assignment)
                                 fi
                                                    end
                   end
   return(1^n)
end
```

The algorithm operates in time proportional to

$$\sum_{i=0}^{k} \binom{n}{i} \sim n^k$$

and clearly returns an assignment with k correct bits (either by finding a solution or by proving that every solution must contain k 1's, and therefore the solution with all 1's has at least k correct bits).

Conjecture: If P ≠ NP there are no better approximation methods than the above (within a polynomial factor).

Certainly, any approximation method which obtains k correct values out of n in asymptotic time less than n^{k^m}, where m is a rational, would solve an important open problem. For example, if there was such an approximation algorithm taking deterministic time $n^{(\log k)^m}$ (m an integer), that would imply that deterministic and nondeterministic $2^{p(n)}$ time bounded languages (p( ) a polynomial) were the same, by a straightforward translation argument.

Chapter 5 Conclusions and remaining problems

We started this thesis by stating five "super" questions and by claiming that they must be solved before a reasonable theory of computational complexity could fully develop. In Chapter 3 we showed that Question 3 (i.e. the power of RAM operations, such as multiplication) can be reduced to the question of what are the trade-offs between time and tape bounded computations, our Question 2. The descriptive power of formal languages was extensively studied by others [65, 32,34] and we presented some of their results in Chapter 2. Finally, in Chapter 4 we showed that answers to many of the questions about probabilistic and parallel machines would imply answers to the other hard problems. Thus, although more and more relationships among these problems become known, the big questions remain unanswered. While it is customary to present in the last chapter a list of open problems touched on but unanswered by a thesis, we felt that concluding by simply asking

P = NP?

P = PTAPE?

would be unfair. We present therefore a short summary of what methods were tried to directly attack the problem, and which were the difficulties met in each case.

To prove equality, it suffices to produce a good algorithm for any complete problem. Quite a bit of effort has gone into investigating some of these problems, without any results -- in fact our intuition gives us no clues as to how such

algorithms would operate, and a major insight into the nature of computations seems required to get such an algorithm.

To prove inequality, we may try using powerful diagonal arguments from recursion theory. However, not only were such diagonal proofs not found yet, but there is an argument against using them, that runs as follows:

in general,diagonal arguments generalize easily to relativized classes (i.e. classes of oracle Tms). For these problems, however, Baker, Gill and Solovay [2] have proven that

Theorem 5.1: There exist sets E and D such that

(1) $P^E = NP^E$

and

(2) $P^D \neq NP^D$

i.e. there are oracle sets for which deterministic polynomially bounded Tm's with oracle E (for equal) accept the same languages as nondeterministic polynomially bounded Tm's, while the sets are different if we use oracle D (for different). In other words, the relativized P = NP question in meaningless.

The idea of the proof is the following: to prove (1), take E to be any P-complete set, say L_U. Then $P^{L_U} = NP$ by the arguments of Proposition 2.2. On the other hand, $NP^{L_U} = NP$, since each call of the oracle saves at most some computation do-able in nondeterministic polynomial time, and a polynomial of a polynomial is a polynomial.

To prove (2) we formalize the following idea: "We have

a complete binary tree of depth n and we hide an object in one of the leaves. We have deterministic and nondeterministic machines that try to find the object by traversing the tree. Then the worst case running time of the deterministic algorithm is 2^n, while the nondeterministic algorithm takes a linear time." Similar ideas were used in comparative schematology [50],[67].

Formally, define D to be the set, built in stages by the program below.

Let P_i be an enumeration of the polynomial bounded deterministic oracle Tm's. We suppose that the running time of P_i is bounded by polynomial p_i. We shall, at stage i, put some numbers into D -- the set of all numbers put into D before stage i is denoted by $D(i)$, and $D = \bigcup_{i=0}^{\infty} D(i)$

stage 0 $D(0) = \emptyset$

$n_{-1} = 0$

stage i choose an n_i that satisfies the following:

a) $n_i > n_{i-1}$

b) $2^{n_i-1} > p_i(n_i)$

c) n_i is greater than the length of any number that was used in an oracle call earlier in the construction.

/* We "hide" a number among the 2^{n_i-1} numbers of length n_i */

compute $p_i^{D(i)}(2^{n_i-1})$

if it is accepted then $D(i+1) = D(i)$

/* no number of length n_i is put */

/* into D -- p_i does not accept D */

else

x = smallest number of length n_i not used in the computation.

/* exists by condition b) */

$D(i+1) = D(i) \cup \{x\}$

/* x is in D , but P_i does not know */

/* about it */

fi

go to stage i+1

Claim: the language

$L_B = \{x \mid \exists y \quad y \in D \quad |y| = |x| \}$

is not in P^D .

Note that $L_B \in NP^D$, since a nondeterministic machine, given x , guesses a y of the same length and uses the oracle to decide whether $y \in D$, all of which can be done in linear time.

Proof: The construction insures at stage i that $P_i^D(2^{n_i-1})$ rejects iff there is a $y \in D$ such that $y = n_i = |2^{n_i-1}|$, since the computation of $P_i^D(2^{n_i-1})$ is identical to the computation of $P_i^{D(i)}(2^{n_i-1})$.

The question of $NP = \overline{NP}$ may be treated similarly and there are oracle sets that realize all consistent set inclusion relations among P , NP and $\overline{NP}$.

If we apply the same techniques to the question of tape vs. time, we also see that the relativized problems can go either way. More precisely,

Corollary: There are oracle sets A, B such that

i) $PTAPE^A = P^A$

ii) $PTAPE^B \neq P^B$

Proof: i) is realized by taking A to be any tape-complete set. It is easily seen that the same argument that showed P^E to be equal to NP^E works here. As for ii), take B to be any oracle set for which $P^B \neq NP^B$. Since the containment $P^X \subseteq NP^X \subseteq PTAPE^X$ holds for any oracle set X, $P^B \neq NP^B$ implies $P^B \subsetneq NP^B \subseteq PTAPE^B$.

Another approach is through combinational circuit complexity. Fischer [12] proved that if a P-complete problem of length n requires a non-polynomial number of gates for its minimal realization for infinitely many n, then no deterministic polynomial time bounded Tm can recognize the set. The problem is that the best known lower bound for combinational circuit complexity of "easy" problems is 2.25n [51], and it seems very hard to get an nlogn or n^2 lower bound, much less a non-polynomial one.

Finally, we may just try to get non-polynomial lower bounds for specific problems. In particular, recognition of tautologies (theorem-proving in the propositional calculus) was attacked and, for specific procedures such as restricted resolution methods, exponential lower bounds were obtained. (regular resolution [66],[16]; bounded resolution [16], and Davis-Putnam procedure [39]. For more details, see [16]). In all these cases the proof consists of exhibiting a set of

boolean expressions, giving an easy argument to show that they are tautologies, and then proving that the original method must use non-polynomial time to prove them. But, by formalizing the argument that they are tautologies (which was straightforward), we get a new system, in which they are easily recognized. Therefore, all such results will fail to prove $P \neq NP$, unless some sort of a diagonal procedure is built into them, which we do not know how to do at this time.

We gave an argument in Chapter 4, to the effect that if we could show that one is required to produce deterministically the computation tree of a nondeterministic Tm, this will take exponential time. All known equality proofs (PTAPE = NPTAPE, PTIME - MRAM = NPTIME - MRAM, etc.) work by simulation of all possible moves. Such equivalence proof cannot exist in the P = NP case, but we have no ideas as to how to prove that the computation tree must be produced.

The comments above should give an idea of why P = NP? is so difficult. Although some of the comments apply to P = PTAPE? and P = DLOG - TAPE? , we expect these to be more tractable. In particular, we conjecture that k - NP $\supsetneq$ DLOG - TAPE , and that a proof of this fact should be easier than of $P \neq NP$. (Certainly no enumerative procedure could work there, since the number of states of a log-tape bounded Tm is less than the number k of solutions to count up to.)

Some technical, easier open problems, are:

1) Are there any nice relationships between k-NP-complete problems (or, in general, problems with several $\exists k$ quantifiers)

and the Meyer-Stockmeyer polynomial hierarchy?

2) If we add the instructions

$R_i \leftarrow R_j$ shift right register R_i by the contents of register R_j

$R_i \leftarrow R_j$ shift left register R_i by the contents of register R_j

to a RAM, what is the power of the resulting machine [53]?

3) Is there a nice set of predicates on the leaves of computation trees, other than $\exists^k$? In particular, what is the power of the predicate $\exists$ on even number of accepting leaves? Is there a set that yields PTAPE?

Bibliography

[1] Aho, A., J.E. Hopcroft and J.D. Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass. 1974.

[2] Baker, T., J. Gill and R. Solovay: Relativization of the P =? NP question. to be published in SICOMP.

[3] Bauer, M., D. Brand, M. Fischer, A. Meyer and M. Patterson: A note on disjunctive form tautologies. SIGACT News 52 (1973), 17-20.

[4] Blum, M.: A machine independent theory of the complexity of recursive functions. JACM 14:2 (1967),332-336.

[5] Blum, M.: On the size of machines. Inf. and Contr. 11:3 (1967), 257-265.

[6] Book, R.V.: On languages accepted in polynomial time. SIAM J. Comp. 1:4 (1972), 281-287.

[7] Book, R.V.: On the structure of complexity classes, in Automata, Languages and Programming (2nd Coll, 1974), Springer Verlag Lecture Notes in Computer Science no. 14, Berlin, 1974. 437-445.

[8] Cobham, A.: The intrinsic computational difficulty of functions. Proc. 1964 Congress for Logic, Math. and Phil. of Science. North Holland, Amsterdam. 24-30.

[9] Cook, S.A.: The complexity of theorem-proving procedures. Proc. 3rd STOC (1971), 151-158.

[10] Cook, S.A.: Linear time simulation of two-way deterministic pushdown automata. Proc. IFIP Congr. 71, North Holland, Amsterdam. 174-179.

[11] Cook, S.A.: Characterizations of pushdown machines in terms of tape-bounded computers. JACM 18:1 (1971), 4-18.

[12] Fischer, M.: Lectures on network complexity. Unpublished.

[13] Fischer, M. and M.O. Rabin: Super-exponential complexity of Pressburger arithmetic. Project MAC TM 43 (1974).

[14] Floyd, R.W.: Nondeterministic algorithms. JACM 14 (1967), 636-644.

[15] Galil, Z.: On direct encodings of nondeterministic Turing machines operating in polynomial time into P-complete problems. SIGACT News 6:1 (1974), 19-23.

[16] Galil, Z.: On the complexity of restricted resolution procedures. Dept. Comp. Sci. TR 74-223. Cornell University (1974).

[17] Garey, M.R., R.L. Graham and J.D. Ullman: Worst-case analysis of memory allocation algorithms. Proc. 4th STOC (1972).

[18] Garey, M.R., D.S. Johnson and L. Stockmeyer: Some simplified NP-complete problems. Proc. 6th STOC (1974) 47-63.

[19] Gill, T. III: Computational complexity of probabilistic Turing machines. Proc. 6th STOC (1974) 91-95.

[20] Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. 17:2 (1969), 416-429.

[21] Hartmanis, J. and H.B. Hunt III: The LBA problem and its importance in the theory of computing. TR 73-171 Dept. Comp. Sci., Cornell University. (1973) To be published by the AMS.

[22] Hartmanis, J. and H. Shank: Two memory bounds for the recognition of primes by automata. MST 3 (1969), 125-129.

[23] Hartmanis, J. and H. Shank: On the recognition of primes by automata. JACM 15 (1968), 382-389.

[24] Hartmanis, J. and J.E. Hopcroft: An overview of the theory of computational complexity. JACM 18:3 (1971), 444-475.

[25] Hartmanis, J. and J. Simon: On the power of multiplication in random access machines. Proc. 15th SWAT (1974), 13-23.

[26] Hartmanis, J. and J. Simon: On the structure of feasible computations. TR 74-210, Dept. of Comp. Sci. Cornell University. To be published in Proc. GI-74, Lecture Notes in Computer Science, Springer Verlag.

[27] Hartmanis, J. and R.E. Stearns: On the computational complexity of algorithms. TAMS 117 (1965), 285-306.

[28] Hopcroft, J.E. in panel discussion in R.E. Miller (ed.): Complexity of Computer Computations, Plenum Press (1972), 178.

[29] Hopcroft, J.E. and J.D. Ullman: Formal Languages and their Relation to Automata. Addison-Wesley, Reading, Mass. (1969).

[30] Hunt, H.B. III: On time and tape complexity of languages. Proc. 5th STOC (1973), 10-19.

[31] Hunt, H.B. III: The equivalence problem for regular expressions with intersection is not polynomial in tape. TR 73-161 Dept. of Comp. Sci., Cornell University (1973).

[32] Hunt, H.B. III: On time and tape complexity of languages. Ph.D. Dissertation, Cornell University, Ithaca, N.Y. (1973).

[33] Hunt, H.B. III: Stack languages, Rice's theorem and a natural exponential complexity gap. TR 142, Dept. of Elect. Eng., Princeton University.

[46] Meyer, A.R.: The inherent computational complexity of theories of ordered sets: a brief survey. Project MAC TM 54 (1974). To appear in Proc. Int. Congr. of Mathematicians in Canada (Aug. 1974).

[47] Meyer, A.R. and L.J. Stockmeyer: The equivalence problem for regular expressions with squaring requires exponential space. Proc. 13th SWAT (1972), 125-129.

[48] Meyer, A.R. and L.J. Stockmeyer: Word problems requiring exponential tape. Proc. 5th STOC (1973), 1-9.

[49] Minsky, M.: Form and content in computer science. JACM 17:2 (1970), 197-215.

[50] Patterson, M.S. and C.E. Hewitt: Comparative schematology. Record of Project MAC Conf. on Concurrent Systems and Parallel Computation. ACM, New York. 119-128.

[51] Paul, W.: 2.25N-lower bound on the combinational complexity of boolean functions. TR 74-222, Dept. of Comp. Sci. Cornell University (1974).

[52] Pratt, V.R.: Every prime has a succinct certification. Unpublished manuscript (January 1974).

[53] Pratt, V.R., L.J. Stockmeyer and M.O. Rabin: A characterization of the power of vector machines. Proc. 6th STOC (1974), 122-134.

[54] Rabin, M.O.: Real-time computation. Israel I. of Math. 1 (1963), 203-211.

[55] Rogers, H. Jr.: Theory of Recursive Functions and Effective Computability. McGraw-Hill, N.Y. (1967).

[34] Hunt, H.B. III and D.J. Rosenkrantz: Computational parallels between the regular and context-free languages. Proc. 6th STOC (1974), 64-74.

[35] Johnson, D.S.: Fast allocation algorithms. Proc. 13th SWAT (1972).

[36] Johnson, D.S.: Approximation algorithms for combinatorial problems. Proc. 5th STOC (1973), 38-49.

[37] Jones, N.D. and W.T. Laaser: Complete problems for deterministic polynomial time. Proc. 6th STOC (1974), 40-46.

[38] Karp, R.M.: Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher (eds.) Complexity of Computer Computations, Plenum Press, N.Y. (1972), 85-104.

[39] Kirkpatrick, D.: Ph.D. Dissertation, University of Toronto, forthcoming.

[40] Kleene, S.C.: Recursive predicates and quantifiers. TAMS 53 (1943), 41-73.

[41] Knuth, D.E.: A terminological proposal. SIGACT News 6:1 (1974), 18.

[42] Ladner, R., N. Lynch and A.L. Selman: Comparison of polynomial time reducibilities. Proc. 6th STOC (1974), 110-121.

[43] Lin, S.: Computer solution of the traveling salesman problem. BSTJ 44:10 (1965).

[44] McNaughton, R. and H. Yamada: Regular expressions and state graphs. In E.F. Moore (ed.) Sequential Machines: Selected Papers. Addison-Wesley, Reading, Mass. (1964).

[45] Meyer, A.R.: Weak second order theory of successor is not elementary recursive. MIT Project MAC TM 38 (1973).

[67] Walker, S.A. and H.R. Strong: Characterization of flow-chartable recursions. Proc. 4th STOC (1972), 18-34.

[68] Warkentin, J.C. and P.C. Fischer: Predecessor machines and regressive functions. Proc. 4th STOC (1972), 81-87.

[69] Weiner, P., S.L. Savage and A. Bogchi: Neighborhood search algorithms for finding optimal traveling salesman tours must be inefficient. Proc. 5th STOC (1973), 207-213.

[56] Rosenkrantz, D.J., R.E. Stearns and P.M. Lewis: Approximate algorithms for the traveling salesperson problem. Proc. 15th SWAT (1974), 33-42.

[57] Ruby, S. and P.C. Fischer: Translational methods and computational complexity. Proc. 6th SWAT (1965), 173-178.

[58] Sahni, S.L.: Some related problems from network flows, game theory, and integer programming. Proc. 13th SWAT (1972), 130-138.

[59] Sahni, S. and T. Gonzales: P-complete problems and approximate solutions. Proc. 15th SWAT (1974), 28-32.

[60] Savitch, W.L.: Relationships between nondeterministic and deterministic tape complexities. JCSS 4:2 (1970), 177-192.

[61] Scott, D.: Some definitional suggestions for automata theory. JCSS

[62] Sethi, R.: Complete register allocation algorithms. Proc. 5th STOC (1973), 182-195.

[63] Simon, J.: On the power of multiplication in random access machines. TR 74-205, Dept. of Comp. Sci., Cornell University. (1974).

[64] Stearns, R.E., J. Hartmanis and P.M. Lewis: Hierarchies of memory limited computations. Proc. 6th SWAT (1965), 179-190.

[65] Stockmeyer, L.J.: The complexity of decision problems in automata theory and logic. Project MAC TR 133 (July 1974).

[66] Tseitin, G.S.: On the complexity of derivation in propositional calculus, in A.O. Slisenko (ed.) Studies in Constructive Mathematics and Mathematical Logic, II.