ON THE STRUCTURE OF FEASIBLE COMPUTATIONS†

J. Hartmanis and J. Simon††

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract:

During the last four years research on lower level computational complexity has yielded a rich set of interesting results which have revealed deep and unexpected connections between various problems and thus brought new unity to this area of computer science. This work has also yielded new techniques and insights which are likely to have further applications, and it has identified some very central problems in the quantitative theory of computing. The purpose of this paper is to give the reader an overview of these developments, an insight into some of these results and applications, as well as an appreciation of the unity and structure which has emerged in this area of research.

1. Introduction

In theoretical computer science we can identify several super problems whose solution is bound to contribute extensively to our understanding of the quantitative aspects of computations and could be of considerable practical value. Among these problems we certainly must include the problems dealing with :

1. The quantitative differences between deterministic and

---

† This research has been 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.

†† On leave of absence from Universidade Estadual de Campinas, Campinas, S.P., Brazil.

nondeterministic computations.

2. The time and memory trade-offs in computations.

3. The computational speed gained by adding new operations to random access machines, such as multiplication.

4. The change in descriptive power of formal languages as we add new features (operations).

The first problem area could also include the related problem of characterizing the quantitative differences between parallel and sequential computations as well as understanding the quantitative differences between "finding a proof" and "verifying that a given proof is correct" in a formal system.

Considerable research effort has been dedicated to problems of this type during the last decade as part of a systematic development of the theory of computational complexity. The real progress, though, in this area of research has come since 1970. During the last four years research in this area has yielded a rich set of new results which have revealed unexpected and deep connections between different problems, including the four mentioned above. These results have brought new unity to the study of the computational complexity of feasible problems and have identified and isolated specific problems on whose solution many others depend. Furthermore, these results have yielded some powerful new techniques, interesting applications outside of computer science, and even insights about the nature of theoretical computer science.

In this paper we will try to give the reader an overview of these developments, an insight into some of these results and their applications as well as an appreciation of the unity and structure which has emerged in this area of research.

2. Feasible computations and nondeterminism

From the early research on effective computability emerged in the first half of this century a consensus 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 <u>effectively computable</u> (or simply <u>computable</u>) 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 in principle 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. Thus we shall define below a computation to be <u>feasible</u> iff it can be computed in polynomial time on a Turing machine. This definition of feasibility of computations, besides being intuitively acceptable, has some very nice mathematical properties and shows a very rugged invariance under changes of the underlying computing model. The last point will be particularly emphasized in Chapter 5 when we study random access machines with different sets of operations.

To make these concepts precise and to simplify our notation, we will consider all through this paper the computational problem of accepting languages or solving suitably encoded problems with "yes" and "no" answers, such as "Does the binary input string represent a prime number in binary notation?" or "Does the given regular expression designate the set of all finite binary sequences?" In all these problems the length of the input sequence is the number of symbols in the sequence and we express the amount of computing resource used in terms of the length of the input sequence.

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

To be able to talk about the class of all feasible computations we introduce

Definition: Let PTIME, or simply P, designate the family of languages accepted in polynomial time by deterministic Turing machines.

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. We will return to the last point when we discuss random access machines in a later section and encounter the same polynomially bounded classes. 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

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

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 nondeterministic 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 i $L(R_i)$ or $L(R_j)$ but not in both. So far no deterministic polynomial time algorithm has been discovered for this problem.

The multitude of problems and languages of this type has led to the definition of the corresponding nondeterministic class of languages.

Definition: Let NPTIME, or simply NP, denote the family of languages accepted in polynomial time by nondeterministic Turing machines.

To emphasize the importance of this class of problems or languages we list some such problems. In all these problems we assume that we have used a straightforward and simple encoding of the problem. For a detailed discussion of such problems see [6,16].

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 propositional calculus involving only variables plus connectives (or 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 easily seen that all of these problems are in NPTIME by a straightforward "guessing and verifying" method. On the other hand, no deterministic polynomial time algorithm is known for any of these problems. This list of problems can easily be extended and it is clear that it contains many practical problems for which we would very much like to have deterministic polynomial time algorithms [16, 1]. The question whether such algorithms exist is by now known as the

P = NP?

problem and it has to be considered as one of the central problems in computational complexity [6].

Intuitively, we feel that $P \neq NP$, though all attempts to prove it have failed. 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 has a deterministic polynomially time bounded algorithm. This simplifies the P = NP? problem considerably, but it still seems quite unlikely that such a deterministic polynomial time algorithm could exist. Of course, a proof that any one of these seven problems is <u>not</u> in P would prove that $P \neq NP$.

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 [26]. 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 [10,11].

A very important property of the class NP was discovered by S.A. Cook [6] when he proved that there exists a language L in NP such that if there exists a deterministic polynomial time algorithm for the recognition of L, then P = NP and we can effectively find (from the deterministic polynomial time algorithm for L) deterministic polynomial time algorithms for every L' in NP.

To make these concepts precise we define complete languages in NP as those languages to which all other languages in NP can be "easily" reduced. Note that the concepts of complete languages and reducibility will be used repeatedly in this study and that they play important roles in recursive function theory and logic [30].

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

$$w \text{ is in } L_i \text{ iff } f_i(w) \text{ is in } L.$$

Proposition 2.1: 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

$$w \in L_i \text{ iff } f_i(w) \in L.$$

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 prove 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 2.2: 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 non-deterministic 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
   $\#M_i\#\mathrm{CODE}(x_1x_2\ldots x_n)\#^{3|M_i|n^q}$
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

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

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

The previous proof shows 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 $0[n^{pq}]$. 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 can not recursively determine the degree of the polynomial.

Our next result shows that, nevertheless, we can get a general translation result. For a related result see [9].

Theorem 2.3: 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 $0[T_i(n)^k]$.

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 $0[T_i(n)^k]$ for a fixed k.

In our construction we use two auxiliary languages:

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

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

Clearly, the languages B_i' and $\Sigma^* - B_i''$ can be accepted in non-deterministic 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 $0[n^p]$. 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 $\#w\#$, $\#w\#^2$, $\#w\#^3, \ldots$.

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.

We conclude by observing that L_U is an NP-complete problem, as defined above.

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

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

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.

<u>Theorem 2.4</u>: L_R is NP-complete and so are all the languages associated with the problems 1-7.

<u>Proof</u>: We prove only that the first problem on our list is NP-complete; for the other proofs see [16].

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 Tm's. 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 $x_1x_2...x_n$. Note that the input $x_1x_2...x_n$ 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 $x_1x_2...x_n$ is not the set of all sequences (of a given length). Thus the test of whether $x_1x_2...x_n$ is accepted by M_i can be reduced to a test of whether the regular expression, describing the invalid computations of M_i on $x_1x_2...x_n$, 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 (we assume without loss of generality that M_i halts after exactly n^k steps). 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 <u>instantaneous description</u> 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$(x_1x_2...x_n)$ denotes the set of <u>valid computations</u> of M_i on input $x_1x_2...x_n$. A valid computation consists of a sequence of instantaneous descriptions

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

such that:

a) $ID_1 = (x_1, q_0)x_2x_3...x_nb^{n^k-n}$

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 <u>invalid</u> set of computations be given by

$$\text{NVCOMP}(x_1 x_2 \ldots x_n) = \Gamma^{n^{2k}+n^k+1} - \text{VCOMP}(x_1 x_2 \ldots x_n).$$

We show next that there exists a deterministic Tm, M_D , which constructs for every input $x_1 x_2 \ldots x_n$ in polynomial time a regular ex- expression using only $\cdot, \cup$, denoting the set of sequences 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

$$\text{NVCOMP}(x_1 x_2 \ldots x_n) = R_1 \cup R_2 \cup R_3.$$

Let $\bar{x}$ designate any 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

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

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

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

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 at most one square, the set CORRECT suffices to charactrrize valid transitions. For example if the Tm in state q , upon reading a 0 may either print a 1 or a 0 and move right, then

$$\mathrm{CORRECT}(\sigma_1(0,q)\sigma_3) = \{(\sigma_1 0(\sigma_3,q)),(\sigma_1 1(\sigma_3,q))\}$$
$$\mathrm{CORRECT}((0,q)\sigma_2,\sigma_3) = \{(0(\sigma_2,q)\sigma_3),(1(\sigma_2,q)\sigma_3)\}$$
$$\mathrm{CORRECT}(\sigma_1\sigma_2(0,q)) = \{(\sigma_1\sigma_2 0),(\sigma_1\sigma_2 1)\}.$$

It is easily seen that 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)

$$\mathrm{NVCOMP}(x_1x_2\ldots x_n)$$

and then checks (using M_C) whether the expressions are unequal. Clearly, the regular expressions are unequal 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.

For the sake of completeness, we mention that it is not known whether the language

$$L_P = \{w \mid w \in 1(0 \cup 1)^* \text{ and } w \text{ designates a composite number}\}$$

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,

$$\tilde{L}_P = \{w \mid w \in 1(0 \cup 1)^* \text{ and } w \text{ designates a prime}\}$$

is also in NP [26]. 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. Since L_P and $\tilde{L}_P$ (or $\bar{L}_P = (0 \cup 1)^* - L_P$) are in NP, and for no NP-complete problem L is it known that $\bar{L}$ is in NP, it seems unlikely that either L_P or $\bar{L}_P$ could be NP-complete.

Note that, if we could show for an NP-complete problem L that $\bar{L}$ is not in NP, then we would have a proof that $P \neq NP$, since L' in P implies that $\overline{L'}$ is in P and thus in 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. Our current understanding of these problems is so limited that we cannot rule out either of the two extremes: a) that $P = NP$ and we need only polynomial time bounded deterministic algorithms, or (b) that there exist L in NP which require an exponential amount of time for their recognition.

As stated before, it appears that a proof that $P = NP$ will have to reveal something new about the nature of computation. Similarly, a proof that for all L in NP, $\bar{L}$ is in NP, which could happen even if $P \neq NP$, would have to reveal something unexpected about the process of computation. To emphasize this, consider again the set of unequal regular expressions over $0,1,\cdot,\cup,($, $)$:

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

As observed before, L_R is easily seen to be in NP. On the other hand, 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.

It may be shown that $L_{R\neg}$ has the same "completeness" property with respect to NP^c, i.e. the set of languages L such that $\Sigma^* - L$ is in NP, as L_R has with respect to NP; we will discuss such "complete" sets throughout the paper.

We conclude this section by observing that if P $\neq$ NP and NP is not closed under complementation, that then P and NP show a good low complexity level analogy to the recursive and recursively enumerable sets; P corresponding to the set of recursive sets and NP to the set of recursively enumerable sets.

3. Memory bounded computations

In the study of the complexity of computations there are two natural measures: the time or number operations and the tape or memory space used in the computation. It is strongly suspected that there exist interesting and important connections between these two complexity measures and that a central task of theoretical computer science is to understand the trade-offs between then [23].

In this section we discuss the problem of how much memory or tape is required for the recognition of the classes P and NP and some related problems. It will be seen that this study again leads us very quickly to some interesting open problems and reveals some interesting analogies with previous problems which will be further pursued in the chapter dealing with random access machines.

In analogy to the time bounded Tm computations we define memory bounded complexity classes.

Definition: Let PTAPE (NPTAPE) denote the family of languages accepted by deterministic (nondeterministic) Turing machines in polynomial tape.

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 3.1: $NP \subseteq PTAPE$.

On the other hand, it is not known whether there exists an L 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.

Thus we are led to another central problem in computational complexity: is NP = PTAPE or possibly P = PTAPE?

At the present time we have to conjecture that NP $\neq$ PTAPE, since nothing in our knowledge of memory bounded computations suggests that PTAPE computations could be carried out in polynomial time. Furthermore, NP = PTAPE would have, as we will see later, some very strong and strange implications.

As stated in the previous chapter, it is not known whether $P = NP$, and this is a very important problem in complexity theory as well as for practical computing. The situation for tape bounded computations is different [31].

<u>Theorem 3.2</u>: Let $L(n) \geq \log n$ be the amount of tape used by a nondeterministic Tm M_i. Then we can effectively construct an equivalent deterministic Tm $M_{\sigma(i)}$ using no more than $[L(n)]^2$ tape.

From this result we get immediately

<u>Corollary 3.3</u>: PTAPE = NPTAPE.

In the last chapter we will give a new proof (without using Savitch's result) that PTAPE = NPTAPE, as part of our characterization of the computational power of multiplication in random access machines.

Next we will show that PTAPE has complete problems, just as NP did, and thus to show that NP = PTAPE we only have to show that one specific language in PTAPE is also in NP.

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

$$w \in L_i \text{ iff } f_i(w) \in L.$$

From this we immediately get

<u>Proposition 3.4</u>: 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:

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

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

<u>Theorem 3.5</u>: L_{UT} and L_R^* are tape complete languages. Thus

$$NP = PTAPE \ (P = PTAPE) \text{ iff } L_{UT} \in NP\ (P) \text{ iff } L_R^* \in NP\ (P) .$$

<u>Proof</u>: 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 [19]) check that the sequence is not in R_i. 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

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

just as in Chapter 2, with $|ID_j| = n^k$. Since we now 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 proof in Chapter 2 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.

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

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

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

where $\mathrm{CORRECT}(\sigma_1\sigma_2\sigma_3)$ 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

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

since w is accepted by M_i iff there exists a valid computation of M_i on w, but this happens iff $\mathrm{NVCOMP}(w) \neq \Gamma^*$. 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} in in NP (P).

It should be pointed out that L_R^* is just one example of tape complete problems about regular expressions. We can actually state a very general theorem which characterizes a large class of such problems (or languages) [9], which shows that, for example,

$\{R \mid R$ regular expression and $L(R) \neq L(R^*)\}$

$\{R \mid R$ regular expression and $L(R)$ is cofinite$\}$

are two such tape complete languages; many others can be constructed using this result. It is interesting to note 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 $\neq$ 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.

To emphasize a further analogy between NP and PTAPE we take a quick look at logic. Recall that all the expressions in propositional calculus which for some assignment of variables become true form an NP-complete language. Thus we have

Theorem 3.6: The problem of recognizing the satisfiable formulas of the propositional calculus is an NP-complete problem [6]. Similarly, the set of true sentences (tautologies) of the propositional calculus is a complete language for $\overline{NP}$.

The next simplest theory, the first order propositional calculus with equality (1EQ) is a language that contains quantifiers but no function symbols or predicate symbols other than =. The following result characterizes the complexity of this decision problem.

Theorem 3.7: The problem of recognizing the set of true sentences for the first order predicate calculus (1EQ) is complete in PTAPE.

Proof: See [22].

Again we see that the difference in complexity between these two decision problems is directly related to the difference, if any, between NP and PTAPE.

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 simplify 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 [21]. In particular, as we will see the addition of the complementation operator makes the decision problem for equivalence of regular expressions practically unsolvable [34,14,22].

Theorem 3.8: 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 $2^{2^{\cdot^{\cdot^{2^n}}}}\Big\}k$ 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 Tm's 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 [33,13]. 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.

The reason why the addition of complementation permits us to describe very long Turing machine computations economically is also easy to see, though the details of the proof are quite messy. The descriptive power is gained by using the complement to go from regular expressions describing invalid computations to regular expressions of (essentially) the same length describing valid computations. For example, consider a Tm M_i which, on any input of length n, counts up to 2^n and halts. Using the techniques from the proof of Theorem 3.5 we can write down

$$NVCOMP(x_1x_2 \ldots x_n)$$

for this machine on cn tape squares, where c is fixed for M_i. But then

$$\neg NVCOMP(x_1x_2 \ldots x_n) = VCOMP(x_1x_2 \ldots x_n)$$

and we have a regular expression of length $\leq cn + 1 < Cn$, describing a computation which takes 2^n steps, and thus

$\text{VCOMP}(x_1x_2\ldots x_n)$ consists of a single string whose length is $\geq 2^{2n}$.

Next we indicate how the above regular expression is used to obtain a short regular expression for $\text{NVCOMP}(x_1x_2\ldots x_n)$ of Tm's using 2^n tape squares.

A close inspection of the proof of Theorem 3.5 shows that

$$\text{NVCOMP}(x_1x_2\ldots x_n) = R_1 \cup R_2 \cup R_3$$

and that the length of R_1 and R_2 grows linearly with n and does not depend on the amount of tape used by the Tm. Only R_3 has to take account of the amount of tape used in the computation since R_3 takes care of all the cases where an error occurs between successive instantaneous descriptions. In the proof of Theorem 3.5 we simply wrote out the right number of tape symbols between the corresponding places where the errors had to occur. Namely we used the regular expression

$$\Gamma^{n^k - 1}$$

as a "yardstick" to keep the errors properly spaced. The basic trick in this proof is to have short regular expressions for very long "yardsticks".

As indicated above, by means of complements we can write a regular expression for $\text{VCOMP}(x_1x_2\ldots x_n)$ of M_i which grows linearly in n and consists of a single sequence of length $\geq 2^n$. With a few ingenious tricks and the accompanying messy technical details we can use this regular expression as a yardstick to keep the errors properly spaced in a Tm computation using more than 2^n tape. Thus a regular expression which grows linearly in length can describe Tm computations using 2^n tape.

By iterating this process, we can construct for any k a regular expression whose length grows polynomially in n and which describes computations of Tm's using more than

$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\Big\}k$ tape squares for inputs of length n. From this we conclude by the previously outlined reasoning, that $\tilde{L}_R$ cannot be recognized by any Tm using tape bounded by an elementary function.

For the sake of completeness we will mention a result [14] about regular expressions without $\neg$, but with $\cup, \cdot, *$ and $\cap$.

Theorem 3.9: 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.

Finally we note that the rather surprising result that for unrestricted regular expressions the equivalence is not decidable in elementary tape cannot be extended to single letter alphabets [29].

Theorem 3.10: The language

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

can be recognized on

$$L(n) \leq 2^{2^{2^{2^{cn\log n}}}}$$

tape.

In the previous proofs we established the complexity of the recognition of unequal pairs of regular expressions by the following method: we described valid or invalid Tm computations by regular expressions and then related 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 describe with expressions of length n or less 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 [14] it is shown that regular expressions over $0,1,\cup,\cdot,\cap,*$ are such a formalism -- from which Theorem 3.9 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 it 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 in our formalism make assertions about Tm computations, 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.

Some of the most interesting applications of this principle have yielded the computational complexity of decision procedures for decidable logical theories. The results are rather depressing in that even for apparently simple theories the decision complexity makes them practically undecidable. We cite two such results.

We first consider the decision procedure for Pressburger arithmetic. We recall that Pressburger arithmetic consists of the true statements about integer arithmetic which can be expressed by using successor function S, addition, and equality. More formally, the theory is given by the axioms of first order predicate logic augmented by:

$(x = y) \to ((x) = S(y)$

$S(x) = S(y) \to (x = y)$

$S(x) \neq 0$

$x + 0 = x$

$x + S(y) = S(x + y)$

$A_{[x]}(0) \to [(\forall x)\ A \to A_{[x]}(S(x)) \to (\forall x)A]$ where x is not free in A and $A_{[x]}(y)$ means y substituted for every occurrence of x in A (induction scheme)

The theory can express any fact about the integers that does not involve multiplication. For example, by writing

$\underbrace{(S\ldots S}_{i \text{ times}}(0)\ldots)$ we get a formula that denotes the integer i,

by writing $\underbrace{x + x + \ldots + x}_{n \text{ times}}$ where x is a formula, we may denote nx (i.e. multiplication by a constant), by writing $(\exists x)[s + x = t]$ we express the fact that $s \leq t$, and by writing

$$(\exists x)[(r = s + \underbrace{x + x + \ldots + x}_{n \text{ times}}) \vee (s = r + \underbrace{x + x + \ldots + x}_{n \text{ times}})]$$

we are stating that $r \equiv s \pmod{n}$

It is a famous result of Pressburger's [28] that this theory is decidable: basically the reason is that all sentences of the theory can be effectively put into the form of a collection of different systems of linear diophantine equations, such that the original sentence is true iff one of the systems has a solution. Since linear diophantine equations are solvable, the theory is decidable. The transformation into and solution of the equations is costly in terms of space and time: the best known algorithm [25] has an upper bound of

$$2^{2^{2^{pn\log n}}}$$

on the deterministic time and storage required for a sentence of length n (p is a constant greater than 1).

Recently it has been shown [7] that any decision procedure will require at least a super exponential number of steps. More precisely

Theorem 3.11: There is a constant $c > 0$, such that for every (possibly nondeterministic) decision procedure A for Pressburger arithmetic, there is an integer n_0, such that for all $n > n_0$ there is a formula of length n which requires $2^{2^{cn}}$ steps of the procedure A to decide whether the formula is true.

The proof of this result is technically quite messy but again follows the principle of describing by short formulas in Pressburger arithmetic long Tm computations, thus forcing the decision procedure to be complex.

Next we look at a surprising result due to A. Meyer about the decision complexity of a decidable second order theory [20].

A logical theory is second order if we have quantifiers ranging over sets in the language. It is weak second order if set quantifiers range only over finite sets. All second order theories have, in addition to first order language symbols, a symbol, e.g. $\in$, to denote set membership. The weak monadic second order theory of one successor has the two predicates

$$[x = S(y)] \text{ (or } x = y + 1) \text{ and } [y \in X]$$

with the usual interpretation. It was shown to be decidable by Buchi and Elgot [3],[4]. We shall abbreviate the theory by WS1S, and the set of sentences of its language by L_{S1S} .

Theorem 3.12: Let M be a Tm which, started with any sentence of L_{S1S} on its tape, eventually halts in a designated halting state iff the sentence is true. Then, for any $k \geq 0$, there are infinitely many n , for which M's computation requires more than $\left.2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\right\} k$

steps and tape squares for some sentence of length n .

In other words, the decision procedure is not elementary recursive.

These asymptotic results actually hold for small n (of the order of the size of the Tm). Since they hold for the amount of tape used, the same bounds apply to lengths of proofs, in any reasonable formalism. Therefore, there are fairly short theorems in these theories (less than half a page long) that simply cannot be proven--their shortest proofs are too long to write down.

The implications of these "practical undecidability" results are not yet well understood, but their philosophical impact on our ideas about formalized theories may turn out to be comparable to the impact of Goedel's undecidability result.

4. Nondeterministic tape computations and the lba problem

It is known, as pointed out before, that PTAPE = NPTAPE and that a nondeterministic L(n)-tape bounded computation ($L(n) \geq \log n$)

can be simulated deterministically on $L^2(n)$ tape [31]. On the other hand, it is not known whether we cannot do better than the square when we go from deterministic to nondeterministic tape bounded computations. As a matter of fact, we do not know whether we cannot eliminate nondeterminism in tape bounded computations by just enlarging the tape alphabet and not the amount of tape used.

This problem of how much memory we can save by using nondeterministic computations has been a recognized open problem since 1964 when it first appeared as a problem about context-sensitive languages or linearly bounded automata [24,18,17,9].

For the sake of completeness we recall that a <u>linearly bounded automaton</u> is a one-tape Turing machine whose input is placed between endmarkers and the Tm cannot go past these endmarkers. Thus all the computations of the lba are performed on as many tape squares as are needed to write down the input and since the lba can have arbitrarily large (but fixed) tape alphabet, we see that the amount of tape for any given lba (measured as length of equivalent binary tape) is linearly bounded by the length of the input word. If the Tm defining the lba operates deterministically we refer to the automaton as a <u>deterministic lba</u>, otherwise as a <u>nondeterministic lba</u> or simply an lba.

Since the connection between linearly bounded automata and context-sensitive languages is well-known [13], we will also refer to the languages accepted by nondeterministic and deterministic lba's as nondeterministic and deterministic context-sensitive languages, respectively. Let the corresponding families of languages be denoted by NDCSL and DCSL, respectively.

Then the lba <u>problem</u> is to decide whether NDCSL = DCSL. It is also an open problem to decide whether the nondeterministic context-sensitive languages are closed under complementation. Clearly if

NDCSL = DCSL then they are closed under complementation, but it still could happen that NDCSL $\neq$ DCSL and that the context-sensitive languages are closed under complementation.

We now show that there exist time and tape hardest recognizable context-sensitive languages. That is, the family NDCSL has complete languages and, as a matter of fact, we have already discussed such languages in this paper.

Recall that

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

and let

$$L_{LBA} = \{\#M_i\#CODE(x_1x_2\ldots x_n)\# \mid x_1x_2\ldots x_n \text{ is accepted by lba } M_i\}.$$

Theorem 4.1: 1. DCSL = NDCSL iff L_R^* is in DCSL iff L_{LBA} is in DCSL.

2. L in NDCSL implies $\bar{L}$ in NDCSL iff $\overline{L_{LBA}}$ is in NDCSL.

3. DCSL $\subseteq$ NP(P) iff L_{LBA} is in NP(P) iff L_R is in NP(P).

The proof is quite similar to the previous proofs that L_{UT} and L_R^* are complete in PTAPE.

It is interesting to note that if L_{LBA} of L_R^* can be recognized on a deterministic lba then all nondeterministic tape computations using $L_i(n) \geq n$ tape can be replaced by equivalent deterministic computations using no more tape. Furthermore, there is a recursive translation which maps the nondeterministic Turing machines onto the equivalent deterministic Turing machines.

Corollary 4.2: DCSL = NDCSL iff there exists a recursive translation σ such that for every nondeterministic Tm M_i which uses $L_i(n) \geq n$ tape, $M_{\sigma(i)}$ is an equivalent deterministic Tm using no more than $L_i(n)$ tape.

Proof: The proof is similar to the proof of Theorem 2.3. For details see [9].

From the above results we see that if DCSL = NDCSL then all other deterministic and nondeterministic tape-bounded computations using more than a linear amount of tape are the same. On the other hand, we have not been able to force the equality downward. For example, we have not been able to show that if all deterministic and nondeterministic tape-bounded computations using $L_i(n) \geq 2^n$ tape are the same, that then DCSL = NDCSL.

Similarly, it could happen that DCSL = NDCSL but that the logn-bounded deterministic languages are properly contained in the non-deterministic logn-bounded computations.

It is worth mentioning that Greibach [8] has recently exhibited a context-free language which plays the same role among context-free languages as L_{LBA} does for context-sensitive languages. Namely, this context-free language is the hardest time and tape recognizable cfl and there also exist two recursive translations mapping context-free grammars into Turing machines recognizing the language generated by the grammar in the minimal time and on the minimal amount of tape, respectively, though at this time we do not know what is the minimal time or tape required for the recognition of context-free languages.

5. Random access machines

In this section we study random access machines which have been proposed as abstract models for digital computers and which reflect many aspects of real computing more directly than Turing machines do. On the other hand, as it will be seen from the results in this chapter, the study of the computational power of random access machines with different instruction sets leads us right back to the central problems which arose in the study of Tm computations. Thus, quite surprisingly, we will show that the difference in computing power of polynomially time bounded RAM's with and without multiplication is characterized by the difference between PTIME and PTAME for Tm's.

More specifically, it is known that the computation time of random access machines without multiplication is polynomially related to the equivalent Tm computation time, and vice versa. Thus the question of whether the deterministic and nondeterministic polynomially time bounded random access machine computations are the same is equivalent to the question of whether P = NP for Tm computations, a problem we discussed before.

In contrast, when we consider random access machines with the power to multiply in unit time, the situation is completely different. We show that for these devices nondeterministic and deterministic computation time is polynomially related and therefore for random access machines with built-in multiplication, P = NP [32,12]. Furthermore, we give a complete characterization of the computational power of these devices: the family of languages accepted in polynomial time by random access machines with multiplication is exactly PTAPE, the family of languages accepted by Tm's in polynomial tape. Thus the additional computing power that a random access machine with multiplication has over such a machine without multiplication is characterized by the difference between PTIME and PTAPE for Tm computations. Recall that we do not know whether PTIME $\neq$ PTAPE and therefore, multiplication could be simulated in polynomial time by addition and boolean operations iff PTIME = PTAPE; again, an open problem which we have already discussed.

For related results about other random access machine models and for more detailed proofs than given in this paper see [12,32,27].

To make these concepts precise we now describe random access machines, RAM's, with different operation sets and step counting functions. Note that we again consider these devices as acceptors.

<u>Definition</u>: A <u>RAM acceptor</u> or <u>RAM with instruction set O</u> 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 <u>program</u> of (possibly labeled) <u>O-instructions</u>. If no two labels are the same, we say that the program is <u>deterministic</u>, otherwise it is <u>nondeterministic.</u> We call a RAM model deterministic if we consider only deterministic programs from the instruction set.

Our first instruction set consists of the following:

$\underline{O_1}$

$R_i \leftarrow R_j \quad (=k)$	(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)
$R_i \leftarrow R_j$ <u>bool</u> R_k	(boolean operations)
<u>if</u> R_i <u>comp</u> R_j label 1 <u>else</u> label 2	(conditional jump)
<u>accept</u>	
<u>reject</u>	

<u>comp</u> 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 <u>nand</u> 10 = 1. <u>bool</u> may be any binary boolean operation (e.g. $\wedge, \vee$, <u>eor</u>, <u>nand</u>, $\supset$, etc.). <u>accept</u> and <u>reject</u> have obvious meanings. An operand of =k is a literal and the constant k itself should be used.

The computation of a RAM starts 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 is the condition is satisfied and one of the instructions with label "label 2" is executed otherwise. Execution stops when an <u>accept</u> or <u>reject</u> 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 complexity measures defined for RAM's 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 of bits used at a given time is the sum of the number of significant bits of all registers in use at that time.)

Unless otherwise stated, time measure will mean unit time measure. We shall call RAM's with instruction set O_1 RAM_1's, or simply RAM's. For a discussion of RAM complexity measures, see [5] or [1].

We will consider another instruction set:

O_2 is O_1 plus the instruction

$$R_i \leftarrow R_j \cdot R_k \qquad \text{(product)}$$

which computes the product of the two operands (which may be literals) and stores it in R_i. RAM's with instruction set O_2 will be called MRAM's (M for multiplication).

We denote by PTIME - MRAM and by NPTIME - MRAM, respectively, the families of languages accepted in polynomial time by deterministic and nondeterministic MRAM's.

We shall outline below the proof of the main results about MRAM's (for more detailed proofs and related results see [12,32,27])

Theorem 5.1: PTAPE $\supseteq$ NPTIME - MRAM

Theorem 5.2: PTIME - MRAM $\supseteq$ PTAPE.

Thus for MRAM's we have that deterministic and nondeterministic polynomial time computations are the same.

Corollary 5.3: PTAPE = NPTAPE.

This follows from the fact that the proofs of Theorems 5.1 and 5.2 actually imply that PTAPE $\supseteq$ NPTIME - MRAM and that PTIME - MRAM $\supseteq$ NPTAPE.

We now sketch a proof of Theorem 5.1.

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 previous one 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 part. 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: $a, a^2, a^2 \cdot a^2 = a^4, a^4 \cdot a^4 = a^8, \ldots, a^{2^t}$, 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 length, that is , up to $\log_2(a^{2^t}) = 2^t \log_2 a$, which may be done in space $\log_2(2^t \log_2 a)$. In particular, for $t=n^k$, space n^{k+1} will suffice, so that b may be written down in polynomial tape. Clearly, t may also be written down in polynomial tape. There is a small difficulty with r: due to indirect addressing M might use high-numbered registers, even though it uses only a polynomial number of them. However, by using a symbol table, at a cost of a squaring of the running time, we may assume that a machine operating in time t uses only its t first registers. It is clear that in that case r may be written down in polynomial tape. 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 previous values of FIND -- clearly the only interesting case is when r was altered in the previous move. For example, if the move at t-1 was $r \leftarrow p \vee s$, then FIND(r,b,t) = FIND(p,b,t-1) $\vee$ 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 FIND's for time up to t-1, we have the recurrence $\ell_t = \ell_{t-1} + c$ $(\ell_0 = cn^{k+1})$ which has the solution $\ell_t = c'n^{k+1}$.

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

$$\ell_t = \ell_{t-1} + cn^{k+1} \text{ with } \ell_0 = n$$

which implies $\ell_t \leq cn^{2k+1}$, and the simulation of multiplication 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.

With the above comments in mind it is easy to write out a complete simulation program and see that it runs in polynomial time. This ends the proof of our theorem, i.e.

Theorem 5.1: Polynomial time bounded nondeterministic MRAM-recognizable languages are recognizable in polynomial tape by Turing machines.

Now we sketch the ideas behind the proof of Theorem 5.2. They are basically a set of programming tricks that enable us to do operations in parallel very efficiently.

To simplify our proof we will use a special RAM model referred to as CRAM (for concatenation). A CRAM is a RAM with the ability to concatenate the contents of two registers in one operation, and also has the operator SUBSTR(A,B) which replaces A by the string obtained from A by deleting the initial substring of the length ℓ, where ℓ is the length of B.

It can be seen that CRAM computations may be simulated easily by MRAM's, and that SUBSTR is not essential to the construction.

For any given Tm T operating in polynomial tape on input x, a CRAM can first generate all possible configurations of this Tm computation (a configuration of T on input x consists of the state of T, the contents of the work-tape and the positions of T's heads). From this set of all possible configurations, the CRAM can 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.

First we indicate how to compute efficiently the transitive closure of a matrix A. We suppose that initially the whole matrix is in a single register. Remember that $A^* = I \vee A \vee A^2 \vee A^3 \ldots \vee A^n \vee \ldots$, where A is n by n and A^i is the i-th power of A in the "and-or" multiplication (i.e. if $C = A \cdot B$, $c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$).

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 $\log n$ of these ($(I \vee A)^{n+1} = (I \vee A)^n$), transitive closure of n by n matrices can be done in time $\log n$ times the time for multiplication. Throughout this proof, "multiplication" will mean "Λ" 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 "Λ" 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

$$(\text{row } 0 \text{ of } A)^n (\text{row } 1 \text{ of } A)^n \ldots (\text{row } n-1 \text{ of } A)^n =$$

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

in one register (where $(\text{row } i)^n$ means n-fold concatenation) and

$$[(\text{column } 0 \text{ of } b)(\text{column } 1 \text{ of } B)\ldots(\text{column } n-1 \text{ of } B)]^n =$$

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

in the other, the "Λ" of the two registers yields all terms $a_{ik} \Lambda b_{kj}$. 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. To collect terms, if we are able to take advantage of the parallel operations at their fullest, we should not have to do more than $\log n$ operations, since each c_{ij} is the sum of n products. Note that in our case $c_{0,0}$ 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 $i \cdot n + (j-1)n$ to $i \cdot n + jn - 1$.

We use the following idea: to add up a row vector of bits, take the second half of the row, add it in parallel to the first half and call the procedure recursively for the new first half. The reader is encouraged to write a routine, using the mask $M' = 0^{n/2}1^{n/2}$ to select the second half (n is the length of the vector) and prefixing strings of 0's to registers to get proper alignment. It is possible to design the algorithm in such a way that this procedure may be done in parallel for several vectors, stored concatenated to each other in a single register. In particular if one starts with n^2 copies of the mask M', then the following procedure obtains all terms of the matrix product C = A B from all the products $a_{ik} \wedge b_{kj}$:

```
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) V B) ∧ M
            K = K/2
            M = (0^K •M) ∧ M
               end
     end:ADDUP
```

ADDUP uses 0^K and K/2 as primitive operations, but K/2 = SUBSTR(K,1) and 0^K may be obtained by successive concatenations of a string with itself: after p steps we get a string of 0's of length 2^p.

In order to perform matrix multiplication one must be able to expand matrices from some standard input form into the two forms we needed in forming the product. In addition, for transitive closure, we must "pack" the result back into standard form. We do not give the messy details: the sort of programming is illustrated by ADDUP. Basically one uses masks and logical operations to get the required bits from their original places, then using concatenations one "slides" a number of them simultaneously to where they belong. The masks and "sliding rules" are updated and the process repeated. It can be shown that all these operations require only time polynomial in the logarithm of the size of the matrix. In fact transitive closure of $n \times n$ boolean matrices may be found in $O(\log n^2)$ CRAM 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 closures.

If a Tm operates on an input of length n in tape n^k, there are at most $O(2^{cn^k})$ different configurations. 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.

Now, in the operation of the 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 (i.e. CORRECT$(\sigma_1\sigma_2\sigma_3)$).

This is the sort of localized change that may be checked by boolean operations. More precisely, it is not hard to write a CRAM routine that checks that configuration c_i follows from configuration c_j in $O(\ell)$ moves, where ℓ is the length of the configurations. Moreover, the operations executed by the program do not depend on the contents of c_i or c_j -- in particular it may be adapted to check whether for vectors of configurations c_{i_t} $t = 0,1,\ldots,p$, c_{jk} $k = 0,1,\ldots p$ c_{jk} follows from c_{i_t} still using only $O(\ell)$ moves. Now the way to generate the transition matrix in time $O(n^{2k}$) where n is the length of the input is: 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 $(c_0)^m(c_1)^m\ldots(c_{m-1})^m$ where $m = 2^{n^k}$ and $(c_i)^m$ means m-fold concatenation, and $(c_0c_1\ldots c_{m-1})^m$ in $O(\log m) = O(n^k)$ 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 completes the description of our simulation algorithm: putting everything together we have a procedure which runs in polynomial time, since the matrix may be computed in $O((\log 2^{cn^k})^2)$ moves and its transitive closure in $O((\log 2^{cn^k})^2) = O(n^{2k})$ moves.

This completes the outline of the proof for the special CRAM used.

Let us restate the results of this chapter. We defined a reasonable RAM model -- the MRAM -- that has 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, i.e. PTIME - MRAM = NPTIME - MRAM .

2) time-bounded computations are polynomially related to Tm tape, i.e. 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, i.e. PTIME = PTAPE iff PTIME - MRAM = PTIME - RAM_1.

This last observation is interesting, since it seems to imply that the elusive difference between time and memory measures for Tm's might perhaps be attacked by "algebraic" techniques developed in "low level" complexity theory.

We also note that RAM's may simulate MRAM's in polynomial time, as long as MRAM's operate in polynomial space and time. Therefore MRAM's are more powerful than RAM's 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 corollaries follow, in the same vein, from 1), 2) and 3). For example:

Corollary 5.4: The set of regular expressions whose complements are non-empty (i.e. L_R^* of section 3) 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 sound quite surprising at first.

Minsky suggested [23] 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 Tm's by MRAM's) 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, since 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 5.5: PTIME $\neq$ PTAPE iff there exists a language L which can be recognized by MRAM's in polynomial time and polynomial memory, but not simultaneously.

Note that if such an L exists, any tape complete problem may be chosen to be it, for example L_R^*.

As we saw, if MRAM's are different from RAM's, they must use more than a polynomial amount of storage (in our simulation it was an exponential amount). This suggests asking whether it is sufficient to have a RAM 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 is that almost anything that expands the length of the registers fast enough will do, as long as we have parallel bit operations: multiplication, concatenation or shifting all have this property. In particular, concatenation, tests and parallel bit operations (no indirect addressing) will do. On the other hand, 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 RAM's makes them a nice characterization of memory-bound complexity classes. We also think they might be useful for studying parallelism.

Since we believe that P $\neq$ NP (and therefore PTIME - RAM $\neq$ NPTIME-RAM) but PTIME - MRAM = NPTIME - MRAM it seems interesting to ask what happens to the P = NP? question in an abstract setting, when we allow a fixed but arbitrary set of recursive operations in a single step. The surprising result [2] is that there are instruction sets for which P = NP and instruction sets for which P $\neq$ NP -- in other words the problem becomes meaningless when asked in such a general setting. Since we have argued that P = NP? is a central problem of theoretical computer science, the result appears to us like a general warning that, by becoming too formal too soon, we can "generalize away" the problems of interest to computer science,and wind up with uninteresting abstractions.

In particular, a proof that P $\neq$ NP would have to deal somehow with the nitty-gritty combinatorics of the problem. We note that our technique for proving inclusion among sets -- diagonalization -- is usually insensitive to such details. The requirement that diagonal arguments be extremely efficient is peculiar to computer science, and the discovery of such a technique may be as big a breakthrough as the discovery of priority methods (nonrecursive but r.e. diagonal methods) was in recursion theory.

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] Büchi, J.R.: Weak second order arithmetic and finite automata. Zeit. f. Math. Log. und Grund. der Math., 6(1960) 66-92.

[4] Büchi, J.R. and C.C. Elgot: Decision problems of weak second order arithmetics and finite automata, Part I. AMS Notices, 5(1959) Abstract 834.

[5] Cook, S.: Linear time simulation of deterministic two-way pushdown automata. Information Processing 71. North Holland, Amsterdam 1972. 75-80.

[6] Cook, S.: The complexity of theorem-proving procedures. Proc. 3rd Ann. ACM Symp. Th. Comp. 1971 151-158.

[7] Fisher, M.J. and M.O. Rabin: Super-exponential complexity of Pressburger arithmetic. Project MAC TM 43 (1974).

[8] Greibach, S.: The hardest context-free language. SIAM J. Comp. v.2, (1973) 304-310.

[9] Hartmanis, J. and H.B. Hunt III: The lba problem and its importance in the theory of computing. TR-171 Dept. Comp. Sci. Cornell University (1973). To be published by the AMS.

[10] Hartmanis, J. and H. Shank: Two memory bounds for the recognition of primes by automata. MST v.3 (1969) 125-129.

[11] Hartmanis, J. and H. Shank: On the recognition of primes by automata. JACM v.15(1968) 382-389.

[12] Hartmanis, J. and J. Simon: On the power of multiplication in random access machines. Conf. Rec. IEEE 15th SWAT (1974).

[13] Hopcroft, J.E. and J.D. Ullman: Formal languages and their relation to automata. Addison-Wesley, Reading, Mass. 1969.

[14] Hunt, H.B. III: On time and tape complexity of languages. Ph.D. Dissertation, 1973, Cornell University, Ithaca, N.Y.

[15] Hunt, H.B. III: On time and tape complexity of languages. 5th Ann. ACM Symp. Th. Comp. (1973)10-19.

[16] Karp, R.: Reducibilities among combinatorial problems. R. Miller and J. Thatcher (eds), Complexity of Computer Computations. Plenum Press (1972) 85-104.

[17] Kuroda, S.Y.: Classes of languages and linear bounded automata. Information and Control v.3 (1964) 207-223.

[18] Landweber, P.S.: Three theorems on phrase structure grammars of type 1. Information and Control v.2(1963) 131-136.

[19] McNaughton, R. and H. Yamada: Regular expressions and state graphs. E.F. Moore (ed) Sequential Machines: Selected Papers. Addison-Wesley, Reading, Mass. 1964.

[20] Meyer, A.: Weak monadic second order theory of successor is not elementary recursive. M.I.T. Project MAC TM 38(1973).

[21] Meyer, A. and L. Stockmeyer: The equivalence problem for regular expressions with squaring requires exponential space. Conf. Rec. IEEE 13th SWAT(1972) 125-129.

[22] Meyer, A. and L. Stockmeyer: Word problems requiring exponential tape. Proc. 5th Ann. ACM Symp. Th. Comp. (1973) 1-9.

[23] Minsky, M.: Form and content in computer science. JACM v.17 n.2 (1970) 197-215.

[24] Myhill, J.: Linearly bounded automata. WADD Technical Note 60-165 (June 1960).

[25] Oppen, D.C.: Elementary bounds for Pressburger arithmetic. Proc. 5th Ann. ACM Symp. Th. Comp.(1973) 34-37.

[26] Pratt, V.R.: Every prime has a succinct certification. Unpublished manuscript (January 1974).

[27] Pratt, V., L. Stockmeyer and M.O. Rabin: A characterization of the power of vector machines. <u>Proc. 6th Ann. ACM Symp. Th. Comp.</u> (1974) 122-134.

[28] Pressburger, M.: Über die Vollstandigkeit eines gewissen System der Arithmetik ganzen Zahlen, in welchen die Addition als einzige Operation hervortritt. <u>Comptes-Rendus du I Congres des Mathematiciens des Pays Slavs</u>. Warsaw, 1929.

[29] Rangel, J.L.: The equivalence problem for regular expressions over one letter alphabet is elementary. <u>Conf. Rec. IEEE 15th SWAT</u> (1974).

[30] Rogers, H. Jr.: <u>Theory of recursive functions and effective computability</u>. McGraw-Hill, New York. 1967.

[31] Savitch, W.J.: Relations between nondeterministic and deterministic tape complexities. <u>JCSS</u> v.4(1970) 177-192.

[32] Simon, J.: On the power of multiplication in random access machines. TR 74-205 Dept. of Comp. Sci. Cornell University (1974).

[33] Stearns, R.E., J. Hartmanis and P.M. Lewis: Hierarchies of memory limited computations. <u>Conf. Rec. IEEE 6th SWAT</u> (1965) 179-190.

[34] Stockmeyer, L.J.: The complexity of decision problems in automata theory and logic. Project MAC TR 133 (July 1974).