Title	Computational complexity of multitape Turning machines and Random Access Machines(Dissertation_全文)
Author(s)	Kasai, Takumi
Citation	Kyoto University (京都大学)
Issue Date	1977-03-23
URL	http://dx.doi.org/10.14989/doctor.r3265
Right	
Type	Thesis or Dissertation
Textversion	author

学位申請論文

笠井琢美

主論文

Computational Complexity of Multitape Turing Machines and Random Access Machines

by

Takumi KASAI

§1 Introduction

In recent years there has been increasing interest in analyzing the computational complexity of programs. The multitape Turing machine has become the standard model used for evaluating time and storage complexity, even though such machines are not much like any existing comupters. Some authors, however, implement there algorithms not on Turing machines but on random access machines. In 1972 S.A. Cook introduced a formal model of a random access machine. This model is closer to real computer, for real computers have to calculate the address of desired storage cell before fetching its content.

Notation. Let N denote the set of natural numbers and let $[k] = \{0,1,\ldots,k-1\}$ for each $k \in N$. Hence $[0] = \phi$. We regard $[k]$ as an alphabet consisting of k symbols. Thus, a language is a subset of $[k]^*$ for some $k \in N$.

Let I and O be sets. We denote by $[I \to O]$ the set of all partial functions from I to O.

Definition 1.1. A computing machine is a 3-tuple $M = (L,I,t)$, where

(i) L is a language,

(ii) I is a function from L to $[I \to O]$, and

(iii) t is a function from L to $[I \to N]$ satisfying the following condition: for each $P \in L$ and $x \in I$,

(3.1) $I(P)(x)$ is defined iff $t(P)(x)$ is defined.

The function I is called the interpretation of M and $I(P)$ is the partial function realized by P under M. We say that $t(P)$ is the time complexity of P, and sometimes write $t(P,x)$ instead of $t(P)(x)$. The set I is the input domain and O is the output domain.

Definition 1.2. Let $M = (L,I,t)$ and $M' = (L',I',t')$ be computing machines with the same input domain I and output domain O. Let $f:N \to N$ be a function. Then M is said to be $f(n)$-translatable to M' if and only if for each $P \in L$, there exist $P' \in L'$ and constant c satisfying the following conditions:

(1.2) $I(P) = I'(P')$,

(1.3) for each $x \in I$, if $t(P,x)$ is defined, then

$$t'(P',x) \leq cf(t(P,x)),$$

that is, if a program P in L takes time $\tau(x)$ for its execution, then there is a program P' in L' which computes the same partial function as P within time $cf(\tau(x))$. If $f(n) = n$, we say that M is linearly translatable to M'. If f is a polynomial, then M is polynomially translatable to M'.

In this paper, we consider the following types of computing machines:

RAM ... the random access machine with indirect addressing,

RAMR... the random access machine without indirect addressing

SM ... the step machine with indirect addressing,

SMR ... the step machine without indirect addressing,

TM ... the multitape Turing machine.

We compare these models on the basis of their ability to reflect the complexity of an algorithm. The results obtained in this paper are summarized in Fig 7.1. In [7], Cook has shown that the RAM is n^2-translatable to the TM. In Section 5, we show that this upper bound cannot be improved, that is, we show that the RAM is not $n^{2-\varepsilon}$-translatable to the TM for any $\varepsilon > 0$. This yields a negative answer to an open problem suggested by Borodin [5] and Aho, Hopcroft and Ullman [2].

One of the purpose of this paper is to construct a good model to use in the theory of computational complexity. We maintain that the SM is a good model, since both RAM and TM (and hence, any restricted type of these machices) are linearly translatable to SM.

§2 Random Access Machine

Definition 2.1. Let D be the set of functions $d:N \to N$. Each element d of D is called a memory. For each $i \in N$, $d(i)$ represents the contents of register i. For each $d \in D$ and $i,j \in N$, let $d(i \leftarrow j)$ be the memory defined by

$$d(i \leftarrow j)(k) = \begin{cases} d(k) & \text{if } k \neq i \\ j & \text{if } k = i \end{cases}$$

For each $n \in N$, let

$$\text{Log } n = \begin{cases} \lceil \log_2 n \rceil & \text{if } n \geq 2 \\ 1 & \text{if } n < 2. \end{cases}$$

Definition 2.2. The RAM instructions, together with their meanings and execution times, are given in Table 1.1, where n is an element of N and d represents a current memory.

Instruction	next memory	execution time for RAM
1. LOAD n	d(0←d(n))	Leg n+Log d(n)
2. SETC n	d(0←n)	Log n
3. STORE n	d(n←d(0))	Log n+Log d(0)
4. READ n	d(0←"input")	Log n+Log "input"
5. WRITE n	d	Log n+Log d(n)
6. JZERO n	d	Log n
7. ADD n	d(0←d(0)+d(n))	Log n+Log d(0)+Log d(n)
8. SUB n	d(0←d(0)-d(n))	Log n+Log d(0)+Log d(n)
9. INCR n	d(0←d(0)+1)	Log d(0)
10. DECR n	d(0←d(0)-1)	Log d(0)
11. LOAD *n	d(0←d(d(n)))	Log n+Log d(n)+Log d(d(n))
12. STORE *n	d(d(n)←d(0))	Log n+Log d(n)+Log d(0)

TABLE 2.1

RAMA Instructions and Execution Times

Definition 2.3. (a) A RAM program is a finite sequence of RAM instructions. (b) A RAMR program is a RAM program without the instruction types LOAD *n and STORE *n. (c) A SM program is a RAM program with neither ADD nor SUB. (d) A SMR program is a RAM program without ADD, SUB, LOAD *n and STORE *n. Thus, it is a SM program without LOAD *n and STOR *n.

Definition 2.4. An element (i,x,y,d) of $N\times N^*\times N^*\times D$ is called a configuration of random access machines. Let $P = s_1 s_2 \ldots s_k$ be a program with s_i being instructions. Let $\vdash_P$ be the relation over the configurations defined as follows. We write

$$(i,x,y,d) \vdash_P (i',x',y',d')$$

if and only if the following conditions are satisfied:

(i) $1 \leq i \leq k$,

(ii) if s_i is JZERO n and $d(0) = 0$ then $i' = n$ else $i' = i+1$,

(iii) if s_i is READ n then $x = a\cdot x'$ for some $a \in N$ else $x' = x$

(iv) if s_i is WRITE n then $y' = y\cdot d(n)$ else $y' = y$,

(v) d' is the next memory determined by Table 2.1.

Let $\vdash_P^*$ be the reflexive transitive closure of $\vdash_P$. If $\alpha \vdash_P^* \beta$ and there is no γ such that $\beta \vdash_P \gamma$, then we write $\alpha \hat{\vdash}_P \beta$.

Let d_0 be the memory defined by

$$d_0(i) = 0 \text{ for all } i \in N.$$

Let $I(P):N^* \to N^*$ be the partial function defined by

$$I(P)(x) = y \text{ iff } (1,x,\lambda,d_0) \hat{\vdash} (i,\lambda,y,d')$$

for some $i \in N$ and $d' \in D$. $I(P)$ is called the partial function realized by P.

Definition 2.5. (a) Time complexity of RAM and RAMR: The time complexity of a RAM program (or a RAMR program) P is the function $t_{RAM}(P):N^* \to N$ such that $t_{RAM}(P)(x)$ is the sum of the execution time taken by each instruction executed on input x, where the time required by each instruction is shown in Table 2.1.

(b) Time complexity of SM and SMR: The time complexity of a SM program (or a SMR program) P is the function $t_{SM}(P):N^* \to N$ such that $t_{SM}(P)(x)$ is the number of instruction steps executed by P on input x. That is, in these machine, each instruction requires one unit of time.

Henceforth, the subscript M on t_M is dropped whenever M is understood.

Definition 2.6. Let $x = x_1 \cdot x_2 \cdot \ldots x_n$ be an element of N^* with each x_i being in N. The proper length of x, denoted by ln(x), is defined by

$$ln(x) = \sum_{i=1}^{n} \text{Log } x_i.$$

Let $f:N \to N$ be a monotone increasing function and let P be a program. Then P executes within time f (alternatively, P is said to be f(n) time bounded) if and only if

$$t(P,x) \leq f(ln(x)) \quad \text{for all} \quad x \in N^*.$$

A language $L \subset [k]^*$ is recognized by a program P if $L = \text{Dom } I(P)$. L is recognizable within time f, abbreviated f-recognizable, if there is a program P recognizing L which executes within time f.

Definition 2.7. Let f be a partial function from N^* to N^*. Then f is said to be of rank k if

$$\text{Dom } f \subset [k]^* \quad \text{and} \quad \text{Im } f \subset [k]^*.$$

A program P is said to be of rank k if the partial function realized by P is of rank k. In this paper, unless stated otherwise, any program is supposed to be of finite rank.

Remark. Note that any partial function realized by a Turing machine is of finite rank. Now we show that the condition of Definition 2.7 is not too severe, that is, we show that any RAM program of infinite rank can be simulated within an $n \log n$ factor by a RAM program of finite rank. Let $A = (1(0 \cup 1)^*2 \cup 02)^*$ Let $\xi: N^* \to A$ and $\nu: A \to N^*$ be the functions defined by

$$\xi(x_1 \cdot x_2 \cdot \ldots x_n) = \overline{x}_1 2 \overline{x}_2 2 \ldots \overline{x}_n 2$$

$$\nu(\overline{x}_1 2 \overline{x}_2 2 \ldots \overline{x}_n 2) = x_1 \cdot x_2 \cdot \ldots x_n,$$

where $\overline{x}_i$ is the binary representation of the integer x_i.

Then, by the proof of Theorem 4.1 in Section 4, it follows that for any RAM program P, there exist a constant c and a RAM program $\overline{P}$ of rank 3 such that

$$I(P) = \nu \cdot I(\overline{P}) \cdot \xi, \quad \text{and}$$

$$t(\overline{P}, \xi(x)) \leq c \cdot t(P,x) \cdot \log t(P,x).$$

§3 Relationship between the RAM and the SM

Theorem 3.1. Let P be a SM program. Then there exists a constant c such that

$$t_{RAM}(P,x) \leq c\ t_{SM}(P,x)\ \log\ t_{SM}(P,x).$$

Proof. Let q be the largest constant appearing as the argument of SETC instruction in P. Let P be of rank k. Then, a number appearing in any register during the computation is less than $q+k+t_{SM}(P,x)$. Hence one instruction costs at most $O(\log\ t_{SM}(P,x))$ time under the logarithmic cost criterion.

Corollary 3.1. The SM is n log n translatable to the RAM. The SMR is n log n translatable to the RAMR.

Notation. Let L_o be the language defined by

$$L_o = \{w2w^R2 \mid w \in \{0,1\}^*\}$$

where w^R denotes the reveral of word w.

Lemma 3.1. L_o is recognizable by a SM program which executes within time $f(n) = cn$ for some constant c.

Proof. Evident

From Theorem 3.1 and Lemma 3.1, we have the following:

Corollary 3.2. L_0 is recognizable by a RAM program which executes within time $f(n) = cn \log n$ for some constant c.

The SMR can be views as a Neuman-type model realization for counter machines [10,11]. The following lemma is an immediate consequence of the result obtained by Fischer, Meyer and Rosenberg [11].

Lemma 3.2. If L_0 is recognizable by a SMR program which executes within time $f(n)$, then $f(n) \geq c^n$ for some constant $c > 1$ and for all n.

Combining Lemmas 3.1 and 3.2, we have the following result.

Corollary 3.3. The SM is not polynomially translatable to the SMR.

Lemma 3.3. If L_0 is recognizable by a RAMR program P which executes within time $f(n)$, then $f(n) \geq cn^2$ for some constant c and for all n.

Proof. Let q be the largest constant appearing as the argument of a SETC instruction in P. First we show that if m is the largest number appearing in any register after a computation of duration τ, then

$$(3.1) \quad \tau \geq \tfrac{1}{2}(\mathrm{Log}^2 m - \mathrm{Log}^2 q).$$

The proof will proceed by induction on the length of a computation. It is trivially true for computations of length 0, since a computation begins with all registers set to zero. Assuming that it is true for a computation

$$(1,u,\lambda,d_0) \vdash_P^* (i,v,\lambda,d),$$

consider the next move of this computation. We may assume that the i-th instruction of P is of the form ADD p. Since

$$\tau \geq \tfrac{1}{2}(\mathrm{Log}^2 \max\{d(0),d(p)\} - \mathrm{Log}^2 q),$$

it follows that

$$\tau+\text{Log } d(0)+\text{Log } d(p)+\text{Log } p$$

$$\geq \frac{1}{2}(\text{Log}^2\max\{d(0),d(p)\}-\text{Log}^2 q)+\text{Log } d(0)+\text{Log } d(p)$$

$$> \frac{1}{2}(\{\text{Log } \max\{d(0),d(p)\}+1\}^2-\text{Log}^2 q)$$

$$\geq \frac{1}{2}(\text{Log}^2(d(0)+d(p))-\text{Log}^2 q)$$

Therefore (3.1) holds for all computations.

Let ℓ be the length of P and let k be the number of registers used in P. Let m be the largest number appearing in any register after reading a word of length $\frac{n}{2}$. Then, for two distinct binary word u and v of length $\frac{n-1}{2}$, if

$$(1,u2u^R2,d_0) \vdash_P^* (i,u^R2,\lambda,d) \quad \text{and}$$

$$(1,v2u^R2,d_0) \vdash_P^* (i',u^R2,\lambda,d'),$$

then either $i \neq i'$ or $d \neq d'$. Hence we have

$$\ell\cdot(m+1)^k \geq 2^{\frac{n}{2}} \tag{3.2}$$

From (3.1) and (3.2), it follows that

$$\tau \geq cn^2$$

for some constant $c > 0$.

Corollary 3.4. If the SM is $f(n)$ translatable to the RAMR, then

$$\sup_{n\to\infty}\frac{f(n)}{n^2} > 0 .$$

If the RAM is f(n) translatable to RAMR, then

$$\sup_{n\to\infty} \frac{f(n)\log^2 n}{n^2} > 0$$

Since the language L_0 can be recognizable in real time by a Turing machine, we have the following result.

Corollary 3.5. If the TM is f(n) translatable to RAMR, then

$$\sup_{n\to\infty} \frac{f(n)}{n^2} > 0.$$

§4. Linear Simulation of the RAM by the SM

In this section, we show that the RAM is linearly translatable to the SM. Since the SM programs to do this are intolerably long, it will be convenient to describe them in a higher-level language called SM-ALGOL, instead of the "machine language" given in Senction 2.

Definition 4.1. A SM-ALGOL program can contain one-dimensional infinite array.

(a) An atomic statement is one of the followings

read v	write v	goto label
v ← w	v ← w + c	v ← w ∸ c

where c is a constant and v and w are either simple variables x or subscripted variables of the forms

a[x]	a[x + c]	a[x ∸ c].

(b) A condition is one of the followings

v = c	v ≠ c

where c is a constant and v is a simple variable or a subscripted variable.

(c) A SM-ALGOL program is a statement of one of the following types.

(1) atomic statement

(2) if condition then statement else statement

(3) if condition then statement

(4) while condition do statement

(5) repeat statement until condition

(6) label: statement

(7) begin statement: ...; statement end

(8) procedure name (list of parameters): statement

(9) procedure-name (arguments)

(d) Recursive procedures are not allowed in SM-ALGOL programs, and any procedure statement of type (9) should be previously defined by a procedure declaration of type (8).

The time complexity of a SM-ALGOL program P is the function $t(P)\colon N^* \to N$: such that $t(P)(x)$ is the number of executions of atomic statements and conditions executed by P on input x.

Lemma 4.1. Every SM-ALGOL program is linearly translatable to a SM program.

Outline of proof. Let P be a SM-ALGOL program. Without loss of generality we may assume that P contains no procedure call. To prove the lemma, it suffices to show that there exist a SM-ALGOL program $\bar{P}$ with exactly one array and constant c such that

$$t(\bar{P}, x) \leq ct(P, x)$$

for all inputs x.

Let the arrays used in P be $A_0, A_1, \ldots, A_{k-1}$, and let simple variables used in P be $X_1, \ldots, X_t$. The program $\bar{P}$ uses a single array A and simple variables $x_1, \ldots, x_t$,

$X'_1, \ldots, X'_t$. The program $\overline{P}$ computes values v and $2kv$ simultaneously whenever P computes the value v, that is, the program P can be constructed such that the following relations are satisfied during execution:

$$X'_i = 2k \cdot X_i$$

$$A[2ki + j] = A_j[i] \qquad 0 \leq j \leq k - 1$$

$$A[2ki + j + k] = 2k \cdot A_j[i] \qquad 0 \leq j \leq k - 1.$$

To do this, for example, the statement $X_i \leftarrow X_j + c$ in P is translated into

$$\underline{\text{begin}}\ \ X_i \leftarrow X_j + c;\ \ X'_i \leftarrow X'_j + 2kc\ \ \underline{\text{end}},$$

the statement $A_j[X_i] \leftarrow X_t$ is translated into

$$\underline{\text{begin}}\ \ A[X'_i + j] \leftarrow X_t;\ \ A[X'_i + j + k] \leftarrow X'_t\ \ \underline{\text{end}},$$

and the statement $X_t \leftarrow A_j[X_i]$ is translated into

$$\underline{\text{begin}}\ \ X_t \leftarrow A[X'_i + j];\ \ X'_t \leftarrow A[X'_i + j + k]\ \ \underline{\text{end}}.$$

It should be evident that the program P can be designed to simulate P faithfully within a constant factor.

Definition 4.2. Let m be a positive integer, and let $m_0, m_1, \ldots, m_t$ be elements of $\{0,1\}$ such that

$$m_t = 1, \qquad m = \sum_{i=0}^{t} m_i 2^t.$$

In this paper, the binary representation for m means the word $m_0 m_1 \cdots m_t 2$. The binary representation for zero is the word consisting a single letter 2.

Theorem 4.1. The RAM is linearly translatable to the SM.

Outline of proof. Let P be a RAM program. We now construct a SM-ALGOL program $\overline{P}$ which linearly simulates P. The program $\overline{P}$ uses arrays ACC, TEMP, INDEX, DATA and CONSm for each constant m appearing as argument of instructions in P. Initially, for each constant m appearing in P, the binary representation $m_0 m_1 \cdots m_t 2$ for m is stored in the array CONSm[0],..., CONSm[t+1].

The array ACC represents the register 0. The binary representation $a_0 a_1 \cdots a_{u+1}$ for the contents a of register x is stored in DATA in a contiguous set of subscripted variables

$$\text{DATA}[e] = a_0, \quad \text{DATA}[e+1] = a_1, \ldots, \text{DATA}[e+u+1] = a_{u+1}.$$

The integer e is called the entry corresponding to x. If a register x has been used thus far in the computation, then the entry e corresponding to x can be found by means of the array INDEX and the binary representation $x_0 x_1 \cdots x_{v+1}$

for x, that is, the integers $e_0 e_1 \cdots e_{v+1}$ can be found such that

$$\text{INDEX}[x_0] = e_0$$
$$\text{INDEX}[e_0 + x_1] = e_1$$
$$\vdots$$
$$\text{INDEX}[e_v + x_{v+1}] = e_{v+1} = e.$$

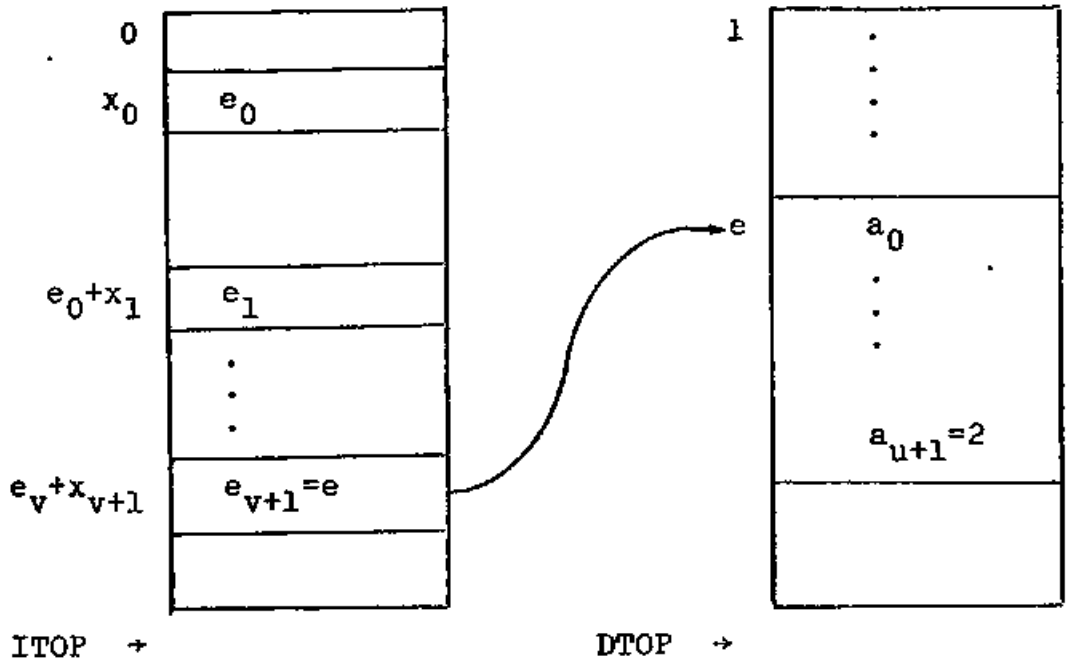


Fig. 4.1.

The procedure FIND(X,e) finds the entry e corresponding to X. The procedure LOAD(X,e) brings the binary representation $a_0 a_1 \cdots a_{u+1}$ to the array X. Precisely, these programs are not SM-ALGOL programs, since they contain

the statement of the form $e \leftarrow e + X[j]$. This type of statement, however, can be easily translated into a SM-ALGOL program, since $X[j] \leq 2$ holds whenever this statement is executed. Clearly, the time complexity of FIND(X,e) is O(v), and hence O(Log x). The time complexity of LOAD(X,e) is O(u), and hence O(Log a).

```
          procedure  FIND(X,e):
          begin
             e ← 0; j ← 0;
             repeat
                begin
                   e ← e + X[j];
                   if  INDEX[e] = 0  then goto notused;
                   e ← INDEX[e];
                   j ← j + 1
                end
             until  X[j - 1] = 2;
             goto  return;
notused:     e ← 0;
return:
          end
```

Fig. 4.2. Procedure FIND

```
          procedure  LOAD(X,e):
```

```
begin
   j ← 0;
   if  e ≠ 0  then
        repeat
           begin
              X[j] ← DATA[e];
              j ← j + 1;
              e ← e + 1
           end
        until  X[j - 1] = 2;
end                .
```

Fig. 4.3. Procedure LOAD

To complete the proof, it suffices to illustrate the simulation of indirect addressing. The statements LOAD *m and STORE *m are simulated by the following SM-ALGOL statements. Now, it should be clear that these statement simulate faithfully within a constant factor.

```
begin
   FIND(CONSm, e);
   if  e ≠ 0  then
        begin
           LOAD(TEMP, e);
           FIND(TEMP, e);
           if  e ≠ 0  then  LOAD(ACC, e)
```

```
        end
    end
```

Fig. 4.4 Simulation of LOAD *m by SM

```
begin
   FIND(CONSm, e)
   if  c ≠ 0  then
        begin
            LOAD(TEMP, e);
            e ← TEMP[0];  j ← 1;
            if  e = 2  goto return;
            repeat
                 begin
                    if  INDEX[e] = 0  then goto notused;
                    e ← INDEX[e] + TEMP[j];
                    j ← j + 1
                 end
            until  TEMP[j - 1] = 2;
            goto  store;
notused:    repeat
                 begin
                    INDEX[e] ← ITOP;
                    e ← ITOP + TEMP[j];
                    ITOP ← ITOP + 3;
                    j ← j + 1
                 end
```

```
            until  X[j - 1] = 2;
    store:  INDEX[e] ← DTOP;  j ← 0;
            repeat
                  begin
                     DATA[DTOP] ← ACC[j];
                     DTOP ← DTOP + 1;  j ← j + 1
                  end
               until  ACC[j - 1] = 2
         end
   return:
  end
```

Fig. 4.5. Simulation of STORE *m by SM

§5. Relationship between the TM and the SM

In this section we show that the SM is not $n^{2-\varepsilon}$ translatable to the TM for any $\varepsilon > 0$.

Definition 5.1. Let U be the subset of [4]* defined recursively as follows:

(5.1) $3 \in U$,

(5.2) If α is in U, then 0α and 1α are both in U,

(5.3) If α and β are in U, then $2\alpha\beta$ is in U.

For each $\alpha \in U$, let $\varphi(\alpha)$ be the language over {0, 1} defined as follows:

(5.4) $\varphi(3) = \lambda,$

(5.5) $\varphi(0\alpha) = 0\varphi(\alpha), \qquad \varphi(1\alpha) = 1\varphi(\alpha),$

(5.6) $\varphi(2\alpha\beta) = 0\varphi(\alpha) \cup 1\varphi(\beta),$

where α and β are elements of U.

Lemma 5.1. Let V be any nonempty subset of $\{0, 1\}^i$. Then there exists an element α in U such that

$$V = \varphi(\alpha) \quad \text{and} \quad |\alpha| \leq 2^{i+1} - 1$$

Proof. The proof will proceed by induction on i. It is trivially true for $i = 0$, since $\varphi(3) = \lambda = \{0, 1\}^0$. Suppose that the lemma is true for all $j < i$, $i > 0$. Let $V_0 = \{v | 0v \in V\}$ and $V_1 = \{v | 1v \in V\}$. Then, $V_k \subset \{0, 1\}^{i-1}$ for $k = 0, 1$. Thus, by the induction hypothesis there exist α and β in U such that

$$V_0 = \varphi(\alpha), \qquad V_1 = \varphi(\beta)$$

$$|\alpha| \leq 2^i - 1, \quad |\beta| \leq 2^i - 1.$$

Hence

$$V = 0V_0 \cup 1V_1 = 0\varphi(\alpha) \cup 1\varphi(\beta) = \varphi(2\alpha\beta),$$

and

$$|2\alpha\beta| = |\alpha| + |\beta| + 1 \leq 2^{i+1} - 1.$$

Therefore the lemma holds for all i.

Definition 5.2. Let L_1 be the language over [5] defined by $L_1 = U(4(0 \cup 1))^*4$. Let $g: [5]^* \to [2]^*$ be the partial function such that

(5.7) $g(y)$ is defined if and only if $y \in L_1$,

(5.8) $$g(\alpha 4 x_1 4 \cdots 4 x_k 4) = b_1 b_2 \cdots b_k,$$

$$b_j = \begin{cases} 0 & \text{if } x_j \in \varphi(\alpha) \\ 1 & \text{if } x_j \notin \varphi(\alpha) \end{cases}$$

where $\alpha \in U$, $x_j \in [2]^*$.

Theorem 5.1. The partial function g can be realized by a SM program in linear time.

Proof. Consider the program MAKETREE in Fig.5.2. The program MAKETREE terminates if and only if the input α is in U. If the program terminates, then the following condition is satisfied at the completion of the program execution:

(5.9) a string $b_0 b_1 \cdots b_k$, $b_i \in \{0, 1\}$, is in $\varphi(\alpha)$ if and only if there exist integers $e_0, e_1, \ldots, e_k$ such that

$$\text{TREE}[2 + b_0] = e_0$$

$$\text{TREE}[e_0 + b_1] = e_1$$

$$\vdots$$

$$TREE[e_{k-1} + b_k] = e_k$$

$$TREE[e_k] = 1$$

The program MAKETREE uses two stacks TREE and STAK with pointers TRTOP and TOP. It should be clear that the time complexity of MAKETREE is $O(|\alpha|)$. In this program, loop means "dead-end", that is, loop is an abbreviation of while 0 = 0 do.

```
procedure MAKETREE:
   begin
      TRTOP ← 2;  TOP ← 1;
      while  TOP ≠ 0  do
         begin
            read  x;
            if  x = 0 ∨ x = 1  then
                begin
                   TREE[TRTOP + x] ← TRTOP + 2;
                   TREE[TRTOP + |x - 1|] ← 0;
                   TRTOP ← TRTOP + 2
                end
            else
                if  x = 2  then
                   begin
                      TREE[TRTOP] ← TRTOP + 2;
                      STAK[TOP] ← TRTOP + 1;
                      TRTOP ← TRTOP + 2;
```

```
                    TOP ← TOP + 1
                end
            else
                if  x = 3  then
                    begin
                        TREE[TRTOP] ← 1;
                        TOP ← TOP - 1;
                        if  TOP ≠ 0  then
                            begin
                                temp ← STAK[TOP];
                                TREE[temp] ← TRTOP + 1;
                                TRTOP ← TRTOP + 1;
                            end
                    end
                else loop;
            end
        read  x;
        if  x ≠ 4  then loop;
    end
```

Fig.5.1. Procedure MAKETREE

The procedure TEST tests whether a given input x_j is in $\varphi(\alpha)$ or not, that is, writes 1 on the output tape if x_j is in $\varphi(\alpha)$, and writes 0 if x_j is not in $\varphi(\alpha)$. The time complexity of TEST is $O(|x_j|)$. Now it should be clear that the desired function g can be realized by a SM program within time $O(n)$, where n is the length of an input string.

```
procedure TEST:
begin
   e ← 2;
   repeat
      read x;
      if x = 0 ∨ x = 1 then
          begin
             e ← TREE[e + x]
             if e = 0 ∨ e = 1 then write 1;
          end
      else
          if x = 4 then
             if TREE[e] = 1 then write 0
                            else write 1
             else loop
   until x = 4 ∨ e = 0 ∨ e = 1
end
```

Fig.5.2. Procedure TEST

Now we show that any Turing machine realizing the partial function g requires at least $n^2/\log n$ steps. The Turing machine which we shall use is an ordinary on-line deterministic machine with a one-way read only input tape, a one-way write only output tape and a finite number of two-way, read and write working tapes of unbounded length.

A configuration of a m-tape Turing machine P is a 4-tuple

$$(q, x, y, d),$$

where q is a state, x is a input tape, y is a output tape and $d \in (N \times N^*)^m$. A pair (q, d) is called tape configuration. We denote by $\vdash_P$ the relation over the configurations which represents one move of the computation of P. For each $i \in N$ and configurations c and c', we write $c \vdash_P^i c'$ if there exists a computation from c to c' of length i, that is, if there exist configurations $c_0, \ldots, c_i$ such that

$$c = c_0 \vdash_P c_1 \vdash_P \cdots \vdash_P c_i = c'.$$

We write $c \vdash_P^* c'$ iff $c \vdash_P^i c'$ for some i, $c \vdash_P^\wedge c'$ iff $c \vdash_P^* c'$ and $c' \not\vdash_P c''$ for all c'', $c \vdash^{i\wedge} c'$ iff $c \vdash^i c'$ and $c \vdash^\wedge c'$. The partial function $I(P): N^* \to N^*$ realized by a Turing machine P is defined by

$$I(P)(x) = y \quad \text{iff}$$

$$(q_0, x, \lambda, d_0) \vdash_P^\wedge (q, \lambda, y, d') \text{ for some } q \text{ and } d',$$

where q_0 is the initial state of P and $d_0 = (0, \lambda)^m$.

The time complexity of P is defined by $t(P)(x) = i$ if and only if there exists a configuration c such that

$$(q_0, x, \lambda, d_0) \vdash_P^{i\wedge} c.$$

Therem 5.2. If a Turing machine P realizes the partial

function g within time $f(n)$, then

$$f(n) \geq c \frac{n^2}{\log n}$$

for some $c > 0$ and for all n.

Proof. Let P be an m-tape Turing machine which realizes g within time $f(n)$. Let A_i be the subset of U defined by

$$A_i = \{\alpha \mid \varphi(\alpha) \subset \{0, 1\}^i\}.$$

By Lemma 5.1,

$$\# A_i = 2^{2^i} - 1, \tag{5.10}$$

where $\#A$ denotes the number of elements in A.

For each $\alpha \in A_i$, let C_α be the set of tape configurations defined by

$$C_\alpha = \{(q, d) \mid (q_0, \alpha 4 x_1 4 \cdots x_\ell 4, \lambda, d_0) \vdash_P^{\wedge} (q, \lambda, b_1 \cdots b_\ell, d), x_1, \ldots, x_\ell \in \{0, 1\}^i\}$$

where q_0 is the initial state of P and $d_0 = (0, \lambda)^m$. Now we show that for $\alpha, \beta \in A_i$,

$$\text{if } \alpha \neq \beta, \text{ then } C_\alpha \cap C_\beta = \phi. \tag{5.11}$$

Assume, for contradiction, that $C_\alpha \cap C_\beta \neq \phi$. Let $(q, d) \in C_\alpha \cap C_\beta$. Then for each $x \in \{0, 1\}^i$,

$x \in \varphi(\alpha)$

iff $(q, x, \lambda, d) \vdash_P^{\wedge} (q', \lambda, 0, d')$ for some q' and d'

iff $x \in \varphi(\beta)$.

Therefore, $\varphi(\alpha) = \varphi(\beta)$. By definition, it should be clear that $\varphi(\alpha) = \varphi(\beta)$ if and only if $\alpha = \beta$. Hence, we have $\alpha = \beta$, contrary to assumption.

Let P have s states and at most k symbols per tape square. We may assume that $k \geq 2$. Let

$$(5.12) \qquad h(i) = \frac{2^i}{2m \log k + \log s} - 1.$$

Let H be the set of all tape configurations (q, d) which satisfy the following conditions:

(5.13) $(q, d) \in C_\alpha$ for some $\alpha \in A_i$, and

(5.14) for every $y \in \{0, 1\}^i$, there exist $t \leq h(i)$ and a configuration c such that

$$(q, y, \lambda, d) \vdash^{t^{\wedge}} c.$$

Next we show that

(5.15) there exists $\alpha \in A_i$ such that $C_\alpha \cap F = \phi$.

Assume, for contradiction, that $C_\alpha \cap F \neq \phi$ for all $\alpha \in A_i$. The only information in storage available to P in next t moves is the present state and the tape information within t squares of the head. From this information, at most $sk^{(2t+1)m}$ configurations can be distinguished in t moves. Hence, by (5.10) and (5.11) we have

$$s \cdot k^{(2h(i)+1)m} \geq 2^{2^i} - 1.$$

This, however, contradicts (5.12).

Now, consider the following input for P:

(5.15) $$z = \alpha 4 x_1 4 \cdots x_\ell 4,$$

where $C_\alpha \cap F = \phi$, $\ell = [2^i/i]$ and $x_1, \ldots, x_\ell$ are in $\{0, 1\}^i$. Then, by Lemma 5.1,

(5.16) $$|z| \leq 2^{i+1} + [2^i/i] \times i \leq 2^{i+2}$$

Consider the following computation:

$$(q_0, \alpha 4 x_1 \cdots 4 x_\ell 4, \lambda, d_0)$$

$$\vdash_P^{t_0} (q_1, x_1 4 \cdots x_0 4, \lambda, d_1)$$

$$\vdash_P^{t_1} (q_2, x_2 4 \cdots x_\ell 4, b_1, d_2)$$

$$\vdots$$

$$\vdash_P^{t_\ell} (q_{\ell+1}, \lambda, b_1 \cdots b_\ell, d_{\ell+1}).$$

Since $C_\alpha \cap F = \phi$, (q_j, d_j) is not in F for each j. Hence we have

$$f(|z|) \geq t_1 + \cdots + t_\ell$$
$$\geq h(i)[2^i/i]$$
$$\geq c_0 2^{2i}/i$$

for some constant c_0 and for all i. Hence

$$f(|z|) \geq c_1 |z|^2/\log |z|$$

for some c_1 and for all z. Since $f(n)$ is monotone increasing with n, we get

$$f(n) \geq c_1 n^2/\log n .$$

Corollary 5.1. If the ST is $f(n)$ translatable to the TM then

$$\sup_{n\to\infty} \frac{f(n)\log n}{n^2} > 0.$$

Combining Corollary 3.1 and Corollary 5.1, we have the following result.

Corollary 5.2. If the RAM is $f(n)$ translatable to the TM then

$$\sup_{n\to\infty} \frac{f(n)\log^3 n}{n^2} > 0.$$

Remark. Since it is proved by Cook and Reckhow that the RAM is n^2 translatable to the TM we can assert that this bound is close to best.

§6. Simulation of the TM by the RAMR

In this section we show that the TM is n^2 translatable to the RAMR.

Definition 6.1. The tape complexity of a turing machine P is the function $s_{TM}(P):N^* \to N$ such that $s_{TM}(P)(x)$ is the number of tape squares used in the computation on input x.

Definition 6.2. A multi-pushdown tape machine is a Turing machine with a read only input tape, a write-only output tape and a finite number of storage tapes with two storage tape symbols 0(blank) and 1. Whenever a head moves left on any one of its storage tape, a "blank" is printed of that tape. Thus, each multi-pushdown tape machine can be viewed as a finite sequence of the following statements (we call this a MPDM program):

(i) PUSHb[i]

(ii) POP[i]

(iii) IF TOP[i] = b THEN GOTO n

(iv) IF INPUT = c THEN GOTO n

(v) WRITE c

where $i,n,c \in N$ and $b \in \{0,1\}$.

The effect of most of the instructions should be evident. For example, PUSHb[i] causes to print the symbol b on top of the stack i. The instruction POP[i] causes to remove the top symbol of the stack i, that is, a "0" is printed on the tape cell scanned and then the head is moved left one cell.

Lemma 6.1. Let P be a Turing machine. Then there exists a multi-pushdown tape machine (a MPDM program) $\overline{P}$ such that

$$I(P) = I(\overline{P})$$

$$t_{TM}(\overline{P},x) \leq c\ t_{TM}(P,x)$$

$$s_{TM}(\overline{P},x) \leq c\ s_{TM}(P,x)$$

for some constant c and for all x.

Proof. Evident.

Definition 6.3. Let top:[2]* → {0,1,λ}, pop:[2]* → [2]*, push0 : [2]* → [2]*, push 1: [2]* → [2]* be functions defined as follows:

$$top(w) = \begin{cases} b & \text{if } w = vb, \quad b \in [2],\ v \in [2]^* \\ \lambda & \text{if } w = \lambda, \end{cases}$$

$$pop(w) = \begin{cases} v & \text{if } w = vb, \quad b \in [2],\ v \in [2]^* \\ \lambda & \text{if } w = \lambda, \end{cases}$$

$$\text{push0}(w) = w0$$

$$\text{push1}(w) = w1$$

Definition 6.4. For each $w \in [2]^*$, let x_w and y_w be the integers defined recursively as follows:

(i) $x_\lambda = 0,\ y_\lambda = 1$

(ii) if $w = v0$ then

$$x_w = x_v + 2y_v$$

$$y_w = x_v + y_v$$

(iii) if $w = v1$ then

$$x_w = x_v + y_v$$

$$y_w = x_v + 2y_v.$$

The following results are immediate consequences of the above definition.

Lemma 6.2. For each $w \in [2]^*$,

$$x_w > y_w \quad \text{iff} \quad \text{top}[w] = 0$$

$$x_w = 0 \quad \text{iff} \quad w = \lambda$$

$$0 < x_w < y_w \quad \text{iff} \quad \text{top}[w] = 1.$$

Lemma 6.3. If $w = vb$ with $b \in [2]$ and $v \in [2]^*$,

then

$$x_v = \underline{\text{if}}\ \ x_w > y_w\ \ \underline{\text{then}}\ \ 2y_w - x_w\ \ \underline{\text{else}}\ \ 2x_w - y_w$$

$$y_v = \underline{\text{if}}\ \ x_w > y_w\ \ \underline{\text{then}}\ \ x_w - y_w\ \ \underline{\text{else}}\ \ y_w - x_w$$

Lemma 6.4. For every $w \in [2]^*$,

$$x_w \leq 3^{|w|}, \qquad y_w \leq 3^{|w|},$$

Theorem 6.1. For any Turing machine P, there exists a RAMR program $\overline{P}$ such that

$$I(P) = I(\overline{P})$$

$$t_{RAM}(\overline{P},x) \leq ct_{TM}(P,x)s_{TM}(P,x)$$

for some constant c and for all x.

Proof. By Lemma 6.1, we may assume that P is a MPDM. Let P have m stacks. If the contents of i-th stack is w, then the integers x_w and y_w are strored in registers $2i + 1$ and $2i + 2$. Let X_i denote the contents of register i. The simulation of P proceeds as follows:

(i) PUSH0[i] is simulated by

$$X_{2i+1} \leftarrow X_{2i+1} + 2X_{2i+2}$$

$$X_{2i+2} \leftarrow X_{2i+1} + X_{2i+2}$$

(ii) PUSH1[i] is simulated by

$$X_{2i+1} \leftarrow X_{2i+1} + X_{2i+2}$$

$$X_{2i+2} \leftarrow X_{2i+1} + 2X_{2i+2}$$

(iii) POP[i] is simulated by

$$X_{2i+1} \leftarrow \underline{\text{if}}\ X_{2i+1} > X_{2i+2}\ \underline{\text{then}}\ 2X_{2i+2} - X_{2i+1}\ \underline{\text{else}}\ 2X_{2i+2} - X_{2i+1}$$

$$X_{2i+2} \leftarrow \underline{\text{if}}\ X_{2i+1} > X_{2i+2}\ \underline{\text{then}}\ X_{2i+1} - X_{2i+2}\ \underline{\text{else}}\ X_{2i+2} - X_{2i+1}$$

(iv) the condition TOP[i] is simulated by

$$X_{2i+1} > X_{2i+2}.$$

By Lemmas 6.2 and 6.3, it should be clear that the simulations above work correctly. By Lemma 6.3, each simulation requires at most $O(s(P,x))$ time. Hence the total time spend by $\overline{P}$ is

$$O(t_{TM}(P,x) \cdot s_{TM}(P,x)).$$

Corollary 6.1. The TM is n^2 translatable to the RAMR.

Proof. The proof follows from the fact that

$$s_{TM}(P,x) \leq t_{TM}(P,x).$$

In [7], Cook and Reckhow show that for each RAM program P, there exist a Turing mach $\overline{P}$ and a constant $c > 0$ such that

$$I(P) = I(\overline{P})$$

$$t_{TM}(\overline{P},x) \leq c\ t^2_{RAM}(P,x)$$

$$s_{TM}(\overline{P},x) \leq c\ t_{RAM}(P,x).$$

From this fact, we have the following result.

Corollary 6.2. The RAM is n^3 translatable to the RAMR.

§7. Conclusion

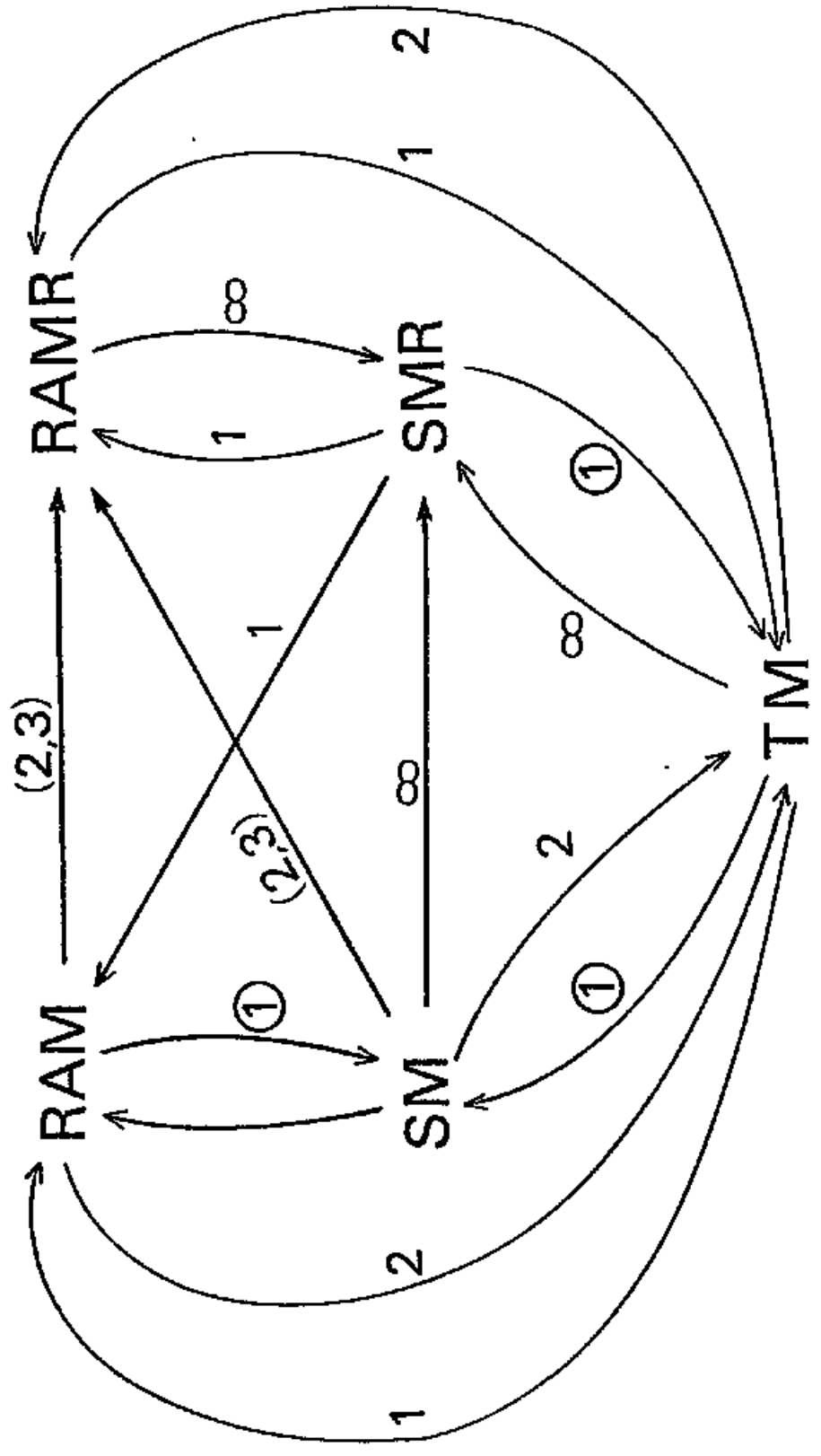
In this section, we summarize the results obtained in this paper.

Notation. Let M and M' be computing machines. We write (i) $M \xrightarrow{k} M'$ if and only if M is $n^{k+\varepsilon}$ translatable to M' for any $\varepsilon > 0$, but not $n^{k-\varepsilon}$ translatable to M' for any $\varepsilon > 0$, (ii) $M \xrightarrow{(1)} M'$ if and only if M is linearly translatable to M' (iii) $M \xrightarrow{(2,3)} M'$ if and only if M is $n^{3+\varepsilon}$ translatable to M' but not $n^{2-\varepsilon}$ translatable to M' for any $\varepsilon > 0$, (iv) $M \xrightarrow{\infty} M'$ if and only if M is not polynomially translatable to M'.

Fig. 7.1

Remark. Since the gap between $n^{k+\varepsilon}$ and $n^{k-\varepsilon}$ is small, the relation $\overset{k}{\longrightarrow}$ is practically optimal. However the gap between n^2 and n^3 is still wide, and the relation $\overset{(2,3)}{\longrightarrow}$ must be improved.

Open problem. Can the upper bound $O(n^3)$ or the lower bound $O(n^2)$ on the time for the RAMR to simulate the RAM be improved?


RAMR
SMR
TM
RAM
SM
2
1
8
1
1
(2,3)
1
8
(2,3)
8
2
1
2
1

Acknowledgements

The author wishes to express his gratitude to Professor Satoru Takasu for his advice. The author is also indebted to Professor Shigeru Igarashi and Mr. Takeshi Hayashi for their suggestions toward this paper.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., Time and tape complexity of pushdown automaton languages. Information and Control 13:3, 186-206 (1968).

[2] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The design and analysis of computer algorithms. Addison-Wesley (1974).

[3] Book, R.V., On languages accepted in polynomial time. SIAM J. Computing 1:4,(1972).

[4] Book, R.V., Greibach, S.A., and Wegbreit, B., Time- and tape-bounded Turing accepters and AFL's JCSS 4:6, 606-621 (1970).

[5] Borodin, A., Computational complexity: theory and practice. In "Currents in the theory of computing" (Aho, ed.). Prentice-Hall, Englewood Cliffs, N.J. (1973).

[6] Cook, S.A., Linear time simulation of deterministic two-way pushdown automata. Proc. IFIP Congress 71, TA-2. Horth-Holland, Amsterdam, 174-179 (1971).

[7] Cook, S.A., and Reckhow R., Time-bounded random access machines. JCSS 7, 354-375 (1973).

[8] Fischer, P.C., Predecessor Machines JCSS 8, 190-219 (1974).

[9] Miller, R.E., and Thatcher, J.W. (eds.), Complexity of Computer Computations. Plenum Press.

[10] Minsky, M.[1967], Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, N.J. 1967

[11] Fischer, P.C., Meyer, A.R. and Rosenberg, A.L., Counter machines and counter languages, Mathematical Systems Theory 2:3, 265-283.