A Canonical Form of Vector Machines

Kazuo Iwama* and Chuzo Iwamoto

Department of Computer Science and Communication Engineering, Kyushu University,
Hakozaki, Higashi-ku, Fukuoka 812, Japan

We introduce RS-vector machines (RS-VMs) as a canonical form of vector machines. They are based on vector operations called repeat and stretch. Repeat enlarges a vector \((a_1, a_2, ..., a_m)\) to \((a_1, a_2, ..., a_m, a_1, a_2, ..., a_m)\) and stretch enlarges \((a_1, a_2, ..., a_m)\) to \((a_1, a_1, a_2, a_2, ..., a_m, a_m)\), when the expansion factor \(d(m) = 2\).

It is shown that we can change the power of RS-VMs depending on this single parameter \(d(m)\): (i) Polynomial-time RS-VMs of \(d(m) = 2\) have the same power as polynomial-space TMs, and (ii) polynomial-time RS-VMs of \(d(m) = m\) have the same power as exponential-time, polynomial-alternation, alternating TMs. The more general results are: (iii) RS-VMs of \(d(m) = k\) (\(k \geq 2\) is constant) have the same power as RS-VMs of \(d(m) = 2\); (iv) a wide variety of \(d(m)\) is such as \(d(m) = m, m^2, ..., cm^k, 2^{\log m}, ..., A(2, m)/m\), have at least the same power as \(d(m) = m\), where \(A(i, j)\) is Ackermann’s function; and (v) any polynomial \(d(m)\) in \(m\) cannot surpass the power of \(d(m) = m\).

1. INTRODUCTION

Vector machines are one of the simplest and the most abstract models of parallel computation. They are totally uniform and are of SIMD-type in the strict sense. The local computation and the communication facility, the two main factors of parallel processing, are clearly distinguished and both are totally abstracted (see below). Vector machines have therefore been considered as an appropriate model for studying structure-based parallel complexity, in particular, of SIMD-type machines. In this paper, we introduce a “canonical form” of vector machines, called RS-vector machines (RS-VMs). It is shown that RS-VMs can change their computational power depending on a single parameter, called an expansion factor.

* This research was in part supported by Grant in Aid Scientific Research for the Ministry of Education, Science and Culture of Japan. Current address: Dept. of Information Science, Kyoto University, Kyoto 606, Japan. E-mail: iwama@kuis.kyoto-u.ac.jp.

† Current address: Dept. of Industrial and Systems Engineering, Hiroshima University, Kagamiyama, Higashi-Hiroshima 739, Japan. E-mail: iwamoto@ke.sys.hiroshima-u.ac.jp.
The notion of vector machines was first introduced by Pratt et al., (1974) and Pratt and Stockmeyer (1976), where it was shown that the instruction set of the shift operation (regarded as realizing the communication facility of parallel computation) and the bit-wise Boolean operations (the local computation) is as powerful as parallel computers of the second class (which, defined by van Emde Boas (1990), can simulate sequential polynomial space by polynomial time). Since then, we have seen a fairly large amount of literature discussing what kind of operation set is how powerful. Hartmanis and Simon (1974) showed that the shift operation above can be replaced by multiplication (by regarding a vector as an integer), concatenation, or other operations. Vector machines with multiplication and integer division but no bit-wise operations (we do not need the local computation as a pure form!) are also equivalent to machines of the second class (Bertoni et al., 1981).

The shift operation, division, and Boolean operation are much more powerful; they can speed up $A(2, n)$ sequential time to polynomial time (Simon, 1979), where $A(i, j)$ is Ackermann's function; i.e.,

$$A(2, n) = 2^{2^{n+1}}.$$  

Thus, this line of research gave us a lot of interesting knowledge on rather surprising powers of various operations on vector machines. Unfortunately, this knowledge was often given by tricky proofs, which did not describe more regular structures of vector operations, corresponding computational powers, or their relations to parallel architectures of real machines. From a practical point of view, van Leeuwen and Wiedermann (1987) proposed array processor machines as a more realistic model.

RS-VMs are introduced as a “canonical form” of vector machines. Vector operations are divided into two categories, local operations (corresponding to the local computation) and global operations (corresponding to the communication facility). The local operations are element-wise addition and subtraction. The global operations include four operations called repeat, stretch, fold, and contract (RS stands for Repeat and Stretch). Repeat enlarges a vector $(a_1, a_2, \ldots, a_m)$ to $(a_1, a_2, \ldots, a_m, a_1, a_2, \ldots, a_m)$, and stretch enlarges $(a_1, a_2, \ldots, a_m)$ to $(a_1, a_1, a_2, a_2, \ldots, a_m, a_m)$. Fold and contract are the inverse operations to repeat and stretch, respectively. Those operations are defined with a parameter $d(m)$, called the expansion factor. The above description of repeat and stretch is for $d(m) = 2$. When $d(m) = m$, repeat and stretch generate vectors of length $m^2$ in a similar manner. Note that vectors in this paper are not Boolean vectors but integer vectors and that those integers do not become too large within a polynomial number of the addition and subtraction operations. It is known that the power of other existing vector machines does not change if Boolean vectors are replaced by integer vectors.

Expectedly, the computational power of RS-VMs depends on the expansion factor. We mainly consider two specific values of $d(m)$: (i) RS-VMs of $d(m) = 2$ have the same power as the machines of the second class, and (ii) RS-VMs of $d(m) = m$ have the same power as exponential-time, polynomial-alternation, alternating Turing
machines (ATMs, see Chandra et al. (1981) for their definition). The results can be extended to: (iii) RS-VMs of $d(m) = k$ ($k \geq 2$ is constant) have the same power as RS-VMs of $d(m) = 2$; (iv) a wide variety of $d(m)$s such as $d(m) = m, m^2, ..., cm^k, 2^n/m, ..., A(2, m)/m$, have at least the same power as $d(m) = m$; and (v) any polynomial $d(m)$ in $m$ cannot surpass the power of $d(m) = m$. (Hence, RS-VMs of $d(m) = cm^k$ have the same power as RS-VMs of $d(m) = m$.) Note that all RS-VMs mentioned in (i) through (v) finish their computation within polynomial time.

It should be noted that the four global operations are unary operations while the conventional shift, multiplication, etc. are binary. We thus claim that the present architecture uses probably a minimum communication facility to exhibit the reasonable computational power as parallel models. An important reason for the above difference of the performance is the length of generatable vectors. When $d(m) = m$ RS-VMs can generate vectors of length $k^{2^{\alpha m}}$ in $t(n)$ steps, and when $d(m) = 2$ they can generate vectors of length $k^{2^{\alpha n}}$. However, this is not the complete story. Simon (1977) showed that although the unrestricted shift operation can generate vectors of length $A(2, n)$ in $n$ steps, its power is still the same as the second machine class.

Attention should be given to the class of languages which are accepted by exponential-time polynomial-alternation ATMs. This class can be characterized by means of RS-VMs quite naturally and robustly. (As mentioned above, the class is characterized by RS-VMs of not just a collection of special expansion factors but a wide range of them). Berman (1980) showed that the first-order theory of the real numbers with addition is complete for the class recognized by $2^{O(n)}$-time $n$-alternation ATMs. Also, Kozen (1980) showed that the elementary theory of Boolean algebra is complete for the same class. Although details are omitted, our result can be shifted from polynomial time to polylog time. For example, polylog-time RS-VMs when $d(m) = m$ are as powerful as polynomial-time, polylog-alternation ATMs.

In the next section, we shall introduce our model and fundamental programming techniques. In Section 3, we outline the major results and underlying ideas. Before giving the proofs of the results, we illustrate in Section 4 how to construct the fundamental vectors used in the simulation. The proofs of the results are given in Sections 5 and 6.

As a final remark, a little more should be mentioned about the SIMD model. Occasionally, the parallel random access machine (PRAM), the most popular shared-memory model, is also categorized as the SIMD model for the reason that it is fully synchronized and all processors execute the same program (e.g., JáJá, 1992; Quinn and Deo, 1984). However, each PRAM-processor can execute a conditional branching operation individually, by which processors might execute different segments of the same program, i.e., different instructions. A vector machine also has a single program, but it is completely free from the notion of different execution sequences. Conditional tests can only be applied to scalar values and cannot be applied to individual vector values. Van Leeuwen and Wiedermann (1987) provided a more realistic model in this “SIMD in the strict sense” category, called array processing machines.
2. RS-VECTOR MACHINES

2.1. Definitions

A vector $A$ of length $m$ is denoted by $A = (a_1, a_2, ..., a_m)$. Each $a_i$ (also denoted by $A[i]$) is a nonnegative integer called a scalar. Let $A = (a_1, a_2, ..., a_i)$, $B = (b_1, b_2, ..., b_j)$, and $\cdot$ be a binary operator for scalars. Then $A \cdot B$ is defined by $(a_1 \cdot b_1, a_2 \cdot b_2, ..., a_k \cdot b_k)$, where $k = \min(i, j)$. The concatenation of vectors $A$ and $B$, denoted by $A \cdot B$ (or sometimes denoted by $(A, B)$, or $AB$, as well) is $(a_1, a_2, ..., a_i, b_1, b_2, ..., b_j)$. $A^0$ is the empty vector; i.e., $A^0 = ( )$. For $i \geq 1$, $A^i = A^{i-1} \cdot A$. In this paper, we mainly use upper-case letters, $A$, $B$, ..., for vectors and lower-case letters, $a$, $b$, ..., for scalars.

The two local operations of RS-VMs are $A + B$ and $A - B$. The four global operations, repeat, stretch, fold, and contract, are denoted by $\downarrow$, $\rightarrow$, $\uparrow$, and $\leftarrow$, respectively. $d(m)$ denotes the expansion factor and is a nondecreasing function of $m$. Let $f$ be a function from vectors to scalars defined as follows (strictly speaking, $f(a_1, a_2, ..., a_i)$ should be written as $f((a_1, a_2, ..., a_i))$):

$$f(a_1, a_2, ..., a_i) = \begin{cases} 0 & \text{if } a_1 = a_2 = \cdots = a_i = 0, \\ a & \text{if every } a_i \text{ is either 0 or a and} \\ \text{at least one of } a_i \text{'s is } a, \\ \text{undefined otherwise.} \end{cases}$$

Note that this definition of $f$ is based on the same idea as the so-called COMMON rule on simultaneous memory write of CRCW-PRAMs (Karp and Ramachandran, 1990).

For $A = (a_1, a_2, ..., a_m)$ and $B = (a_1, a_2, ..., a_{d(m)} \cdot m)$, repeat, stretch, fold, and contract are defined as follows:

$\Downarrow A = A^{d(m)}$ (1)

$\rightarrow A = (a_1)^{d(m)}, (a_2)^{d(m)}, ..., (a_m)^{d(m)})$ (2)

$\Uparrow B = (f(a_1, a_{m+1}, ..., a_{(d(m)-1) \cdot m + 1}), f(a_2, a_{m+2}, ..., a_{(d(m)-1) \cdot m + 2}), ..., f(a_m, a_{2m}, ..., a_{d(m) \cdot m}))$ (3)

$\leftarrow B = (f(a_1, a_2, ..., a_{d(m)}), f(a_{d(m)+1}, a_{d(m)+2}, ..., a_{2d(m)}), ..., f(a_{(m-1) \cdot d(m)+1}, a_{(m-1) \cdot d(m)+2}, ..., a_{md(m)}))$ (4)

Now we are ready to introduce an RS-VM formally. An RS-VM is defined as a program $V$ recognizing a language over $\{0, 1\}$. $V$ can use:
1. A finite number of scalar variables: \(x, y, z, \ldots\)

2. A finite number of scalar constants: \(a, b, c, \ldots\)

3. Scalar instructions:
   \[x := y + z, \quad x := y - z \quad (=0 \text{ if } z > y), \quad \text{if } (x > 0) \text{ goto label, accept, reject}.
   \]

4. A finite number of vector variables: \(X, Y, Z, \ldots\)

5. A finite number of vector constants: \(A, B, C, \ldots\)

6. A special vector constant \(0 = (0, 1, 2, 3, \ldots). \) (None of \(a, b, A, \) and \(c\) can be applied to \(0\). \(0\) is assumed to have an unlimited number of elements.)

7. A special vector variable \(IN\). When the input string is \(i_1i_2 \cdots i_n \in \{0, 1\}^*, \) \(IN[j]\) holds \(i_j\) for \(j = 1, 2, \ldots, n.\) We assume that \(IN[j]\) holds large integers (2) for \(j > n.\)

8. Local and global vector operations:
   \[X := Y + Z, \quad X := Y \cdot Z \quad (\text{this} \, \, \text{"\cdot" \text{is \text{the} \, \text{same} \, \text{definition as that of the scalar instructions; i.e.,} \, \, Y[i] - Z[i] = 0 \text{if} \, \, Z[i] > Y[i]}, \, \, X := \downarrow Y, \, \, X := \rightarrow Y, \, \, X := \uparrow Y, \, \, \text{and} \, \, X := \leftarrow Y).
   \]

9. Vector-scalar translation:
   \[x := X[1].\]

Remark 1. There are several cases of undefined operations such as \(\uparrow(1, 2, 0, 3)\) and \(\downarrow(0, 0, 1)\) for \(d(m) = 2.\) It is the responsibility of the RS-VM programmer to ensure that undefined operations are never executed.

Remark 2. It would be better if we could remove the special vector \(\Omega.\) (i) When \(d(m)\) is a constant, \(\Omega\) is not needed at all for all the results in this paper. (ii) When \(d(m) = m, \) \(\Omega\) can also be removed if we allow element-wise multiplication. See Section 4.5 for more details.

2.2. Useful Subroutines

Frequently used subroutines in RS-VM programs are as follows:

Element-wise doubling operation. By executing \(A := A + A\) \(i\) times, we can enlarge each element \(a\) of \(A\) to \(a \cdot 2^i.\)

a-Inverse operation. Let \(A\) be a vector whose elements are either 0 or \(a.\) Subtracting \(A\) from vector \((a, a, \ldots, a)\) of the same length as \(A\) switches 0’s and \(a\)’s of \(A,\) which is denoted by \(\neg.\) The vector \((a, a, \ldots, a)\) can be obtained by applying the repeat operation logarithmic times. \(1\)-inverse is simply denoted by inverse.

Booleanization operation. Let \(A\) be a (possibly not 0/1) vector. Then, we can change all nonzero elements of \(A\) into 1’s by first computing \((1, 1, \ldots, 1) \cdot \neg A. \) The vector \((a, a, \ldots, a)\) can be obtained by applying the Booleanization operation \(\neg A.\)

Element-wise logical OR operation. Denoted by \(\lor.\) Let \(A\) and \(B\) be 0/1-vectors of the same length. We can obtain \(A \lor B\) by applying the Booleanization operation to \(A + B.\)

Element-wise logical AND operation. Denoted by \(\land.\) Let \(A\) and \(B\) be 0/1-vectors of the same length. We can obtain \(A \land B\) by applying the Booleanization operation to \(A \cdot B.\)

Coincide operation. For vectors \(A\) and \(B,\) we occasionally need to extract the positions where these two vectors have the same value. We first compute the
“difference” between $A$ and $B$ by computing $D := (A - B) + (B - A)$. The answer is obtained by applying the Booleanization operation to $D$ and then applying the inverse operation to the result.

**Mask operation.** Suppose that there are two vectors of the same length, say $A$ and $MASK$, where $MASK$ is a 0/1-vector. Now we want to mask $A$ by $MASK$, namely, to change the elements of $A$ into 0’s in all the positions where $MASK$ has 0’s. We apply the element-wise doubling operation on $\neg MASK$ in order to make non-zero elements of $\neg MASK$ sufficiently large. Subtracting the resulting vector from $A$ gives the answer. Suppose for example that $A = (2, 4, 1, 5)$ and $MASK = (1, 1, 0, 0)$. Then masking $A$ by $MASK$ yields $(2, 4, 0, 0)$.

**Operation $\&*$.** A different kind of subtraction for scalar is defined by

$$x \&* y = \begin{cases} x - y & \text{for } x \geq y, \\ x & \text{for } x < y. \end{cases}$$

$X \&* Y$ can be computed as follows: We first execute $Z := Y - X$ and $Z := (1, 1, \ldots, 1) - Z$. (Namely, $Z[i] = 1$ if and only if $X[i] \geq Y[i]$.) We then make each nonzero element of $Z$ sufficiently large using doubling operation. The answer is given by $(X - Y) + (Z - Z)$. This operation is often useful. Suppose that we want to extract 1’s in a (possibly not 0/1) vector $A$ (i.e., we want to change all $A$’s elements greater than 1 into 0’s). We first apply the Booleanization operation to $A$ and then change all 1’s of $A$ into 0’s and then change all $A$’s elements greater than 2 into 2’s, which gives a 0/2-vector, say $A_2$. The answer is given by $(\& A_1) + \neg (\neg A_2)$. For example, suppose that $A(\ell) = 4$ and $A = (2, 2, 0, 0, 2, 1, 0, 1, 2, 1, 1, 2, 2, 2, 2)$. Then $A_1 = (0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0)$ and $A_2 = (2, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2)$. Therefore, applying the $\&$-contract operation to this $A$ yields $(\& A_1) + \neg (\neg A_2) = (0, 1, 1, 0) + (0, 0, 0, 2) = (0, 1, 1, 2)$.

**$\&$-contract operation.** Defined by (4) and

$$f(a_1, a_2, \ldots, a_\ell) = \begin{cases} 2 & \text{if } a_i \geq 2 \text{ for every } i, \\ 1 & \text{if } a_i = 0 \text{ for at least one } i, \\ 0 & \text{otherwise (i.e., each } a_i \neq 0 \text{ and at least one } a_j = 1). \end{cases}$$

Note that $f$ computes $\ell$-bit OR if ignoring 2’s and larger values. Let $A$ be a vector. We change all $A$’s elements greater than 1 into 0’s by applying the 1-extracting operation, which gives a 0/1-vector, say $A_1$. We change all 1’s of $A$ into 0’s and then change all $A$’s elements greater than 2 into 2’s, which gives a 0/2-vector, say $A_2$. The answer is given by $(\& A_1) + \neg (\neg A_2)$. For example, suppose that $A(\ell) = 4$ and $A = (2, 2, 0, 0, 2, 1, 0, 1, 2, 1, 1, 2, 2, 2, 2)$. Then $A_1 = (0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0)$ and $A_2 = (2, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2)$. Therefore, applying the $\&$-contract operation to this $A$ yields $(\& A_1) + \neg (\neg A_2) = (0, 1, 1, 0) + (0, 0, 0, 2) = (0, 1, 1, 2)$.
function computes 1-bit AND. As above, we change all $A$’s elements greater than 1 into 1’s, which gives a vector, say $A_2$. $(\neg(\neg(\neg A_1)) - \neg(\neg(\neg A_2))) + \neg(\neg(\neg A_2))$ gives the answer. Consider the same $d(m)$ and $A$ as above. Then $A_2 = (1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$, and therefore $(0, 0, 1, 1) - (0, 0, 0, 2)) + (0, 0, 0, 2) = (0, 0, 1, 1)$.

### 3. RESULTS

Let $RSTIME(d(m), t(n))$ denote the class of languages that are accepted by RS-VMs of expansion factor $d(m)$ within $t(n)$ steps, and let $ATIMALT(t(n), a(n))$ denote the class of languages accepted by ATMs within $t(n)$ steps and $a(n)$ alternations between the universal and existential states (denoted by $\lor$ and $\land$-states, respectively).

Here are our major results:

**Theorem 1.** $RSTIME(2, \text{poly}) = \text{DSPACE}(\text{poly})$.

**Theorem 2.** $RSTIME(m, \text{poly}) = ATIMALT(2^{\text{poly}}, \text{poly})$.

Thus, we can change the power of RS-VMs only by changing the expansion factor. We extend these results as follows:

**Theorem 3.** $RSTIME(k, \text{poly}) = \text{DSPACE}(\text{poly})$ for an integer $k \geq 2$.

**Theorem 4.** For an expansion factor $d(m) \geq m$, let $d_0 = 2$, and $d_{i+1} = d_i \cdot d(d_i)$ for $i \geq 0$. Suppose that $d(m)$ is an expansion factor such that $d(d_i)/d_i$ is an integer for all $i \geq 0$. Then $RSTIME(d(m), \text{poly}) \subseteq ATIMALT(2^{\text{poly}}, \text{poly})$.

**Theorem 5.** $RSTIME(\text{poly}, \text{poly}) \subseteq ATIMALT(2^{\text{poly}}, \text{poly})$.

**Corollary 1.** $RSTIME(cm^k, \text{poly}) = ATIMALT(2^{\text{poly}}, \text{poly})$ for integers $c \geq 1$ and $k \geq 1$.

In this paper, we only show the proofs of Theorems 1 and 2 (see Sections 5 and 6, respectively). The proofs of Theorems 3 and 4 are omitted except for a brief description of the idea. Examples of the function $d(m)$ satisfying the condition in Theorem 4 are $d(m) = m, m^2, ..., cm^k, 2^m m, ..., A(2, m)$, and so on. For example, consider $d(m) = 2^m m$. Repeating (or stretching) a vector of length $d_0 = 2$ yields a vector of length $2 \cdot d(2) = 2^2$, which is equal to $d_1$. Then, repeating (or stretching) a vector of length $d_1 = 2^2$ yields a vector of length $d_2 = 2^3$. In general, by repeating the initial vector of length $2 \cdot i$ times, we obtain a vector of length $d_i = A(2, i)$. The reason $d(d_i)/d_i$ must be an integer will be mentioned in Section 4.3.

It may appear that RS-VMs of larger expansion factors should be able to simulate obviously those of smaller expansion factors (i.e., Theorem 4 looks like a corollary of Theorem 2). However, it is not true. Rather than this, the converse is more obvious; for example, RS-VMs of $d(m) = 2$ (of a smaller expansion factor) can simulate RS-VMs of $d(m) = 4$ (of a larger expansion factor) quite easily (although with a reasonable time penalty). Theorem 5 shows that the power of RS-VMs does not increase when the expansion factor stays polynomial.
We shall take a look at one of the fundamental programming techniques to understand why RS-VMs have such computational powers. Suppose that \( d(m) = m \). Then, if a vector, say \( X \), of length \( m \) is repeated (or stretched), then it becomes of length \( m \cdot d(m) = m^2 \). We can regard this vector of length \( m^2 \) as a square matrix, say \( M \), of \( m \) columns and \( m \) rows. The key point is that we can enlarge not only the size of the vector but also its “amount of content.” Suppose, for example, \( X = (0, 1, 2, 3) \) and \( Y \) is a vector obtained by applying the element-wise doubling operation to \( X \) twice; i.e., \( Y = (0, 4, 8, 12) \). Then, we can write \( \downarrow X \) and \( \rightarrow Y \) as:

\[
\downarrow X = (0, 1, 2, 3), \quad \rightarrow Y = (0, 0, 0, 0, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12)
\]

Note that each row of \( \downarrow X \) is equal to \( X \) and that each column of \( \rightarrow Y \) is equal to \( Y \). One can naturally pair each element of \( \downarrow X \) and each one of \( \rightarrow Y \) of the same position. Thus, we can obtain \( m^2 \) different pairs of elements (where \( m = 4 \) in the above case). For example, we can obtain 16 different integers, 0 through 15, by adding above \( \downarrow X \) and \( \rightarrow Y \).

To seek more computational powers, we could set \( d(m) = m^2 \). Now, when vectors of length \( m \) are repeated (or stretched), they become of length \( m \cdot d(m) = m^3 \). Again we can regard this vector as a cube vector and can try to extend naturally the above technique to obtain a vector containing \( m^3 \) different elements. Unfortunately, this approach does not work. Since RS-VMs have only two global operations, \textit{repeat} and \textit{stretch} (and their inverse ones), they can still pair only two different objects. Thus the similar pairing of \( \downarrow X \) and \( \rightarrow Y \) on an RS-VM of \( d(m) = m^2 \) still gives us only \( m^2 \) different elements, which is the same as before. We conjecture that the computational power given in Theorem 4 is the best possible for RS-VMs of any expansion factor.

4. CONSTRUCTION OF FUNDAMENTAL VECTORS

4.1. Overview of the Proof

As mentioned before, we mainly prove Theorems 1 and 2. It is well known (Chandra et al., 1981) that \( \text{DSPACE}(\text{poly}) = \text{ATIMALT}(\text{poly}, \text{poly}) \). Hence, all we have to prove is the following four lemmas:

**Lemma 1.** \( \text{RSTIME}(2, \text{poly}) \supseteq \text{ATIMALT}(\text{poly}, \text{poly}) \).

**Lemma 2.** \( \text{RSTIME}(2, \text{poly}) \subseteq \text{ATIMALT}(\text{poly}, \text{poly}) \).

**Lemma 3.** \( \text{RSTIME}(m, \text{poly}) \supseteq \text{ATIMALT}(2^\text{poly}, \text{poly}) \).

**Lemma 4.** \( \text{RSTIME}(m, \text{poly}) \subseteq \text{ATIMALT}(2^\text{poly}, \text{poly}) \).
One can see that Lemmas 1 and 3 (and Lemmas 2 and 4) are very similar. In fact, the proofs of the two Lemmas proceed analogously; the only difference is the length of involved vectors, \(2^{\log_2 d(m)} = 2\) and \(2^{2^{\log_2 m}} = m\). (Repeating a vector of length \(m = 2^i\) yields a vector of length \(m \cdot d(m) = 2^i \cdot 2 = 2^{i+1}\) when \(d(m) = 2\). When \(d(m) = m\), the length increases from \(m = 2^2\) to \(2^2 \cdot 2^2 = 2^{2i+1}\).)

It should also be noted that a single step of repeat \(A\) for \(d(m) = m\) can be simulated by \(\log |A|\) steps of repeat \(A\) for \(d(m) = 2\). That means that if we could prove Lemma 3 first then the proof of Lemma 1 could be greatly simplified. However, we will not take this approach, since the proof of Lemma 3 is fairly complicated and is much better understood after reading the (simpler) proof of Lemma 1. As for the proofs of Lemmas 2 and 4, we do omit the former, since there are no compelling reasons to treat the two cases separately.

In this section, we describe how to construct the fundamental vectors that are needed in the simulation of Lemmas 1 and 3. This will exhibit important differences between \(d(m) = 2\) and \(d(m) = m\) and will also contribute to increased transparency in describing the simulation.

4.2. Construction of All the Different \(n\)-Bit Patterns \((d(m) = 2)\)

In this section, we construct a vector, say, \(T_n(h)\), which is the concatenation of all lexicographically sorted \(n\)-bit patterns; i.e,

\[
T_n(n) = \begin{cases} 
0 \ldots 000 \\ n \\
0 \ldots 001 \\ n \\
0 \ldots 010 \\ n \\
0 \ldots 011 \\ n \\
\cdots \\
0 \ldots 111 \\ n 
\end{cases}.
\]

\(T_n(n)\) plays a key role to simulate all the different computations by the ATM. It can be constructed as follows. Let \(n\) be a power of two, and let

\[
T_n(h) = (s_0(0), s_0(1), \ldots, s_0(2^n - 1))^{(2^n/2^n)}
\]

\[
= (0 \ldots 00 0 \ldots 00 \ldots 0 \ldots 00 \\ n \\
0 \ldots 01 0 \ldots 01 \ldots 0 \ldots 01 \\ n \\
\cdots \\
0 \ldots 11 0 \ldots 11 \ldots 1 \ldots 11 \\ n )^{2^n/2^n},
\]

\(T_n(h)\) contains all the different \(2^h\)-bit patterns. Note that we only consider the value of \(h\) up to \(\log n\). The length of \(T_n(h)\) is \(n \cdot 2^h\), which is independent of \(h\). \(T_n(h)\) consists of \(2^n/2^h\) subvectors, all of which are the same. Each section consists of \(n/2^h\) subvectors of length \(2^h\), all of which are again the same.
The construction is by induction on $h$. We start with constructing

$$T_n(0) = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \cdots & 1 & 0 & \cdots & 0 \end{pmatrix}^{2^n/2^n}. $$

By repeating $(0, 1)$ $n-1$ times, we obtain $(01)^{2^n/2^n}$. By stretching it log $n$ times, we obtain $T_n(0)$.

Suppose that we have already constructed $T_n(h)$. To construct $T_n(h+1)$, we will "double" the length of all the different bit-patterns. We first repeat $T_n(h)$ (of length $n \cdot 2^h$) log$(n \cdot 2^h)$ times, which gives a vector, say $T'_n(h)$, of length $n \cdot 2^h \times n \cdot 2^h$. We also stretch $T_n(h)$ log$(n \cdot 2^h)$ times, which gives a vector, say $T''_n(h)$. The matrix-like representation of vectors is useful (see Fig. 1). If we regard $T'_n(h)$ as a matrix of $n \cdot 2^h$ columns and $n \cdot 2^h$ rows, then each row is the same as $T_n(h)$. Similarly, each column of $T''_n(h)$ is the same as $T_n(h)$. We construct $T_n(h+1)$ from $T'_n(h)$ and $T''_n(h)$ using a mask matrix $Z$ as follows. $Z$ has the same size as $T'_n(h)$ (and as $T''_n(h)$). Figure 1 shows the left upper part of $Z$. (This figure illustrates the case for $n=8$ and $h=1$; namely, we have already obtained all bit-patterns of length $2^h=2$.) Each shaded portion of $n$ rows and $n$ columns is called a box. Each box has diagonal elements from left-top to right-bottom, where $2^h$ 1’s and $2^h$ 2’s appear alternately and all the elements except for the diagonal elements are 0’s.

Now we construct a new matrix by extracting the diagonal elements of $T'_n(h)$ or $T''_n(h)$ depending on $Z$, namely, extract $T'_n(h)$’s element if the value of $Z$ is 1 and $T''_n(h)$’s element if the value of $Z$ is 2. (See the left-upper four shaded boxes in the figure. 00 in $s_2(0)$ of $T'_n(h)$ is concatenated with each of 00, 01, 10, and 11 of $T''_n(h)$, and thus 0000, 0001, 0010, and 0011 are obtained. In the next four shaded boxes, 0100, 0101, 0110, and 0111 are obtained, and so on.) More precisely, we first construct vectors $Z_1$ and $Z_2$ such that $Z'_1[i] = 1$ (resp. $Z'_2[i] = 1$) iff $Z[i] = 1$ (resp. $Z[i] = 2$); i.e., $Z_1 = Z - (1, 1, ... , 1)$ and $Z_2 = (Z - Z_1) - Z_2$. We can extract the diagonal elements by masking $T'_n(h)$ and $T''_n(h)$ with $Z_1$ and $Z_2$, respectively. By adding the resulting two vectors together, we obtain the new vector, say $T_n(h+1)$.

![Mask vector Z of length $n \cdot 2^h \times n \cdot 2^h$.](image-url)
By folding $T_d(h+1) \log(n \cdot 2^h)$ times, we can compress each column of $T_d(h+1)$ to a single element, which is equal to the diagonal element of the box. The resulting vector is what we wanted; i.e., $T_d(h+1)$.

The next problem is how to construct the mask vector $Z$. We need five vectors $A_d(h)$, $B_d(h)$, $D_n$, and $E_d(h)$, which have the same length as $T_d(h)$.

$A_d(h) = (00 \ldots 0 11 \ldots 1 22 \ldots 2 \cdots (2^h - 1) \cdots (2^3 - 1))^{2^h/2^h}$. \hspace{1cm} (5)

$B_d(h) = (00 \ldots 0 11 \ldots 1 (2^h - 1) \cdots (2^3 - 1))^{2^h/2^{2^h}}$. \hspace{1cm} (6)

$D_n = (0, 1, 2, \ldots, n - 1)^2$. \hspace{1cm} (7)

$E_d(h) = (11 \ldots 1 22 \ldots 2 11 \ldots 1 22 \ldots 2 22 \ldots 2 11 \ldots 1 22 \ldots 2)^2$. \hspace{1cm} (8)

We first construct a matrix called $Z_{BOX}$ which has 1’s inside the shaded boxes and 0’s outside. We stretch $A_d(h) \log(n \cdot 2^h)$ times and repeat $B_d(h) \log(n \cdot 2^h)$ times. Then $Z_{BOX}$ is obtained by applying the coincide operation to the resulting two vectors. We next construct a matrix called $Z_{DIA}$, which has 1’s on the diagonal lines (not only inside the shaded boxes but everywhere) in Fig. 1. We repeat $D_n \log(n \cdot 2^h)$ times and stretch $D_n \log(n \cdot 2^h)$ times. By applying the coincide operation to the resulting two vectors, we obtain $Z_{DIA}$. We then compute $Z_{BD} := Z_{BOX} \land Z_{DIA}$. We repeat $E_d(h) \log(n \cdot 2^h)$ times. Now one can see that $Z$ is finally obtained by masking the resulting vector by $Z_{BD}$.

It remains to show how to construct $A_d(h)$, $B_d(h)$, $D_n$, and $E_d(h)$. Let $Q(l) = (0, 1, 2, \ldots, l - 1)$. (We will construct $Q(l)$ in Section 4.5.) We first repeat $Q(2^h)$ ($n - 2^h$) times, which gives a vector of length $2^h$. By stretching the resulting vector log $n$ times, we obtain $A_d(h)$. $B_d(h)$ and $D_n$ may be omitted. $E_d(h)$ can be constructed as follows. We subtract $(n/2)^{l - n}$ from $D_n$ by the $-^*$ operation, and then subtract $(n/2)^{l - 2^h}$ by the $-^*$ operation, and so on, until we subtract $(2^{h+1} - 1)^{l - 2^{h+1}}$. We obtain $((0, 1, 2, \ldots, 2^{h+1} - 1))^{2^{h+1}}$. By subtracting $(2^{h+1} - 1)^{l - 2^{h+1}}$ from it and by applying the Booleanization operation to the result, we obtain $((0)^2, (1)^2)^{n/2^{h+1}}$. $E_d(h)$ is obtained by adding $(1)^{n - 2^h}$ to it.

4.3. Construction of All the Different $2^h$-Bit Patterns $(d(m) = m)$

Recall that what we want to get is a vector like $T_d(n)$ in Section 4.2, which includes $2^h$ different subvectors of length $k$. If we let $k = 2^h$, the length of the whole vector is $2^n \cdot 2^h$. However, when $d(m) = m$, the length of the vector (if it is obtained by repeats and stretches from an original vector of length two) must be $2^h$ for some $i$ ("lack of flexibility" as mentioned below). Hence, to meet the condition that $2^h \cdot 2^h \leq 2^h$, we have to set $i = n + 1$. Subsequently we can solve our problem by constructing $2^n$ different subvectors consisting of $2^n$ “important” bits plus $2^{h-2^n}$ “padding” bits. Now let
where $0 \leq h \leq n$ and $n$ is a power of two. $T_2(h)$ contains all the different $2^h$-bit patterns in its sections of length $2^n$.

Recall that a single application of $\text{repeat } A$ (stretch, etc.) for $d(m) = m$ is equal to the log $|A|$ repetitions of $\text{repeat } A$ for $d(m) = 2$, and two applications for $d(m) = m$ is equal to the $3 \log |A|$ repetitions of $\text{repeat } A$ for $d(m) = 2$. This means that the larger expansion factor $d(m) = m$ lacks a lot of flexibility (only rather specific numbers of repetitions are possible) compared with the smaller $d(m)$. This makes the simulation more difficult than before.

The construction of $T_2(h)$ is again by induction on $h$. We first generate the following vector, say $V$, which plays a key role in constructing $T_2(0)$.

$$V = \begin{pmatrix} 0 & 0 & \cdots & 0 & 2^0 & \cdots & 2^n \\ 2^n & 2^n & \cdots & 2^n & 0 & \cdots & 0 \end{pmatrix}^{2^h/2^n}.$$

$V$ can be constructed as follows. Let

$$U(2^j) = (1, 2, \ldots, 2^j, 0, \ldots, 0),$$

$$W(2^j) = (1, 0, 0, \ldots, 0).$$

By applying the Booleanization operation to $U(2^j)$ and then the inverse operation to the result, we obtain $((0, 1, 1, \ldots, 1))$ of length $2^{2^j}$. By repeating it and then applying the element-wise doubling operation to the result, we obtain $V$. ($U(2^j)$ and $W(2^j)$ can be constructed by induction on $j$ as follows: We can construct $U(2^{2^j})$ by adding 1 to each element of $\Omega(2^n) = (0, 1, 2, \ldots, 2^n - 1)$. Next we can construct $W(2^{2^j})$ by applying the Booleanization operation to $\Omega(2^n)$ and then the inverse operation to the result. $U(2^{2^j})$ can be obtained by masking $\downarrow U(2^j)$ with $\rightarrow W(2^j)$, and $W(2^{2^j})$ by $\downarrow W(2^j) \times \rightarrow W(2^{2^j})$.)

Now we construct $T_2(0)$ using $V$ as follows. By repeating $(0, 1)$ $n$ times, we obtain $(01)^{2^n/2^n}$. Stretching it gives $((0)2^n, (1)2^n)_{2^n/2^n}$. By subtracting $V$ from $((0)2^n, (1)2^n)_{2^n/2^n}$ and then adding $V$ to the result, we obtain the following $T_2(0)$. 

$$T_2(h) = (s_h(0), s_h(1), \ldots, s_h(2^{2h} - 1))^{2^n/2^n}.$$
We may omit how to construct $T_{2^h}(h+1)$ from $T_{2^h}(h)$, since it is similar to the construction of $T_n(h+1)$ from $T_n(h)$ in Section 4.2. For example, we use the following $A_{2^h}(h)$ instead of $A_n(h)$.

$$A_{2^h}(h) = (00 \ldots 0 \ 11 \ldots 1 \ 22 \ldots 2 \ (2^h-1) \ldots (2^h-1))^{2^h/2^h}.$$ 

For more details including the general case for $d(m) \geq m$ (discussed briefly below), see Iwama and Iwamoto (1996).

4.4. Construction of All the Different $2^n$-Bit Patterns ($d(m) \geq m$)

The construction is much more complicated in details but the basic strategy is the same as before. We only present the fundamental vectors. Let $d_h$ be the function defined in Theorem 4. Then, $T_{2^h}(h)$ has the following structure. (It should be noted that $T_{2^h}(h)$ when $d(h) = m$ (i.e., $d_h = 2^h$) is a special case of the following general $T_{2^h}(h)$.)

$$T_{2^h}(h) = (s_{d_h}(0), s_{d_h}(1), \ldots, s_{d_h}(2^h-1), \ldots)^{(d_h+1)/d_h} \cdot d_h,$$

where each $s_{d_h}(i)$ is a section written as

$$s_{d_h}(i) = (B(i)^{2^{i-k}}, 2, 2, \ldots, 2)^{d_h+1/d_h}$$

and $B(i)$ denotes the binary representation of the integer $i$ using $j$ bits (e.g., $B_2(6) = 0110$). The length of $T_{2^h}(h)$ is $d_{h+1}$, which is independent of $h$. This also explains why we need the condition given in Theorem 4: we want $d_{h+1}/(d_h)^2$ to be an integer (note that $d_{h+1}/(d_h)^2 = (d_h \cdot d_h)/(d_h)^2 = d_h/(d_h)$). Since $d_{h+1}/(d_h)^2$ is an integer, $d_h$ is a divisor of $d_{h+1}$. This implies that $d_i$ is always a divisor of $d_j$ for $i < j$, and therefore the exponent $(d_h/d_{h+1}) \cdot d_e$ in $T_{2^h}(h)$ is an integer.

4.5. Construction of Vector $\Omega(l)$

We first construct $\Omega(2^n)$ for $d(m) = m$ without using the special vector $\Omega$. Suppose that $n$ is a power of two (and thus $\Omega(2^n)$ can be written as $\Omega(2^{2^n})$). We construct $\Omega(2^n)$ by induction on $i$, where $i \leq \log n$. When $i = 0$, $\Omega(2^n) = (0, 1)$. We first apply the element-wise doubling operation to $\rightarrow \Omega(2^n)$ $2^i$ times (note that $2^i \leq n$). By adding $\Omega(2^n)$ to the result, we can obtain $\Omega(2^{2^n})$. Therefore, we can obtain
\(O(2^n) = \Omega(2^{2n})\) in \(O(n \log n)\) time when \(d(m) = m\). Again recall that we can simulate a single step of the \textit{repeat} operation for \(d(m) = m\) by polynomial steps of the \textit{repeat} operations for \(d(m) = 2\) if the length of the vector is exponential. Hence, we can construct \(\Omega(2^n)\) in polynomial time when \(d(m) = 2\).

If we use the special vector constant \(\Omega\) and if \(d(m) = m\), then we can construct \(\Omega(2^n)\) in \(O(n)\) time as follows. By \textit{stretching} \((0, 0)\) (of length \(2^2\)) \(n\) times, we obtain \((0)2^n\). By adding it to \(\Omega\), we \textit{“cut”} \(\Omega\) and obtain \(\Omega(2^n)\). If the element-wise multiplication, say \(*\), is allowed on vectors, we can construct \(\Omega(2^n)\) in \(O(n)\) time when \(d(m) = m\) without using the special vector \(\Omega\). Let \(Y(2^n) = (2^n)2^n\). Then \(\Omega(2^n) = (\Omega(2^n)) + ((\rightarrow Y(2^n)) + Y(2^n+1))\). If \(2^n+1\) \(= 2^n(\rightarrow Y(2^n)) \rightarrow Y(2^n))\). Note that Theorems 2 and 5 also hold for RS-VMs in which \(\Omega\) is replaced by the \(*\) operation.

5. \(\text{RSTIME}(2, \text{poly}) = \text{ATIMALT}(\text{poly}, \text{poly})\)

5.1. \textit{Overview}

In this section, we prove Lemma 1 (i.e., \(\text{RSTIME}(2, \text{poly}) \supseteq \text{ATIMALT}(\text{poly}, \text{poly})\)). The proof of Lemma 2 (i.e., \(\text{RSTIME}(2, \text{poly}) \subseteq \text{ATIMALT}(\text{poly}, \text{poly})\)) is omitted, since it is similar to the proof of Lemma 4 which will be given in Section 6. At first, \(\text{RSTIME}(2, \text{poly}) \supseteq \text{ATIMALT}(\text{poly}, 0) = \text{DTIME}(\text{poly})\) is proved, and then it is extended to the alternating case.

Let \(t(n)\) be a time-constructible function, and let \(M\) be a \(t(n)\)-time bounded, deterministic, single read-write tape TM with only 0 and 1 as its tape symbols. We construct a vector machine \(V\) of \(O(t(n)^c)\) steps (constant \(c\) is not large) which simulates \(M\). Since \(M\) finishes its operation within \(t(n)\) steps, it uses at most \(t(n)\) tape cells. We determine whether \(M\) accepts its input by constructing \(t(n)\)-step configuration sequences of \(M\).

Roughly speaking, the vector machine \(V\) operates as follows: (i) \(V\) computes the time \(t(n)\) and determines the least integer \(N\) that is greater than or equal to \(t(n)\) and is a power of two. (ii) \(V\) generates all the different \(N\)-step configuration sequences, \(c_1, c_2, \ldots, c_N\). (iii) \(V\) then generates all the different \(N\)-step “shifted” configuration sequences, \(c_Nc_1, \ldots, c_{N-1}\). (iv) According to \(M\)’s transition function, we change each \(c_i\) into its “one-move-after” configuration \(c_i’\). By comparing every sequence \(c_1c_2\cdots c_N\) with the corresponding shifted sequence \(c_Nc_1\cdots c_{N-1}\), \(V\) determines whether there exists a proper sequence \(c_1c_2\cdots c_N\) which simulates the computation of \(M\).

Before giving the details of each step, we explain the main ideas of constructing necessary vectors. In Section 4.2, we constructed a vector containing all the different \(2^h\)-bit patterns. By concatenating every pair of \(2^h\)-bit patterns, we then constructed all the different \(2^{h+1}\)-bit patterns. In this section, we first construct a vector containing all the different “configurations” of the \(t(n)\)-time TM \(M\). Then, by concatenating every pair of configurations, we construct a vector containing all two-step-sequences of configurations. By repeating this procedure, we obtain four-step-sequences, eight-step-sequences, and so on, up to \(t(n)\)-step-sequences of configurations. Thus, the main ideas of constructing vectors are similar. In the
following, we mainly explain how the constructions of the necessary vectors in this section differ from those in Section 4.

(i) Computing $t(n)$ and $N$ is not hard and is omitted.

(ii) Since $M$ can use up to $N$ space, the maximum numbers of all the different possible tapes and head-positions are $2^N$ and $N$, respectively. Let $L$ be the number of $M$’s configurations; i.e., $L = 2^N N k$, where $k$ is the number of $M$’s states. We first construct the following vector $C$, which contains all the different $N$-step configuration sequences of $M$. (See Section 5.2 for details.)

$C = (c(1) \cdots c(1) c(1) \cdots c(1) c(2) \cdots c(1) c(3))$

In $C$, each $c(j)$ represents configuration $j$. (We also use $c_i$ which denotes $M$’s configuration at step $i$. For example, $c(1) c(4) \cdots c(9) c(6)$ means that $c_1 = c(1)$, $c_2 = c(4)$, ...)

Actually, we use two vectors, tape vector $T$ and head-state-position vector $H$, instead of $C$. A single integer $(j - 1)N + i$ represents state $s_j$ and head-position $i$. For example, suppose that $M$’s tape is $10111001$ (when $N = 8$), the head is placed on the third cell, and the state is $s_5$. Then we represent such a configuration by two vectors $(1, 0, 1, 1, 1, 0, 0, 1)$ and $(35, 35, 35, 35, 35, 35, 35, 35)$, where $35 = (5 - 1) \cdot N + 3$. We sometimes regard this pair of vectors $T$ and $H$ as a single vector $C$. Each section of $T$ and $H$ looks like (the following example is when $N = 8$):

$T : \cdots \ 10010111 \ 10011111 \ 10011011 \ 10011001 \ \cdots \ \cdots$

$H : \cdots \ 29 \ 29 \ 29 \ 22 \ 22 \ 22 \ 22 \ \cdots \ \cdots$

$TS : \cdots \ \cdots \ 10010111 \ 10011111 \ 10011011 \ 10011001 \ \cdots \ \cdots$

$HS : \cdots \ \cdots \ 29 \ 29 \ 29 \ \cdots \ 22 \ 22 \ 22 \ \cdots \ 39 \ 39 \ 39 \ \cdots \ 14 \ 14 \ 14 \ \cdots \ \cdots$

($TS$ and $HS$ are the shifted vectors defined later.)

More natural way of expressing the head position is, as in (Bertoni et al., 1974), to use a 0/1 vector which holds exactly one 1 at the position where the head is placed (or to change $T$ above so as to hold 2 or 3 (when its tape symbol is 0 or 1, respectively) only at the head position). Our current approach is quite different;
we explicitly write the head position as an integer, which is repeated \( N \) times to be adjusted to tape vectors. The major reason is that our vector machines lack the ability of the “shifting operation.” One can feel that if we use the conventional approach then we must “shift” the position of 1 to the left or to the right according to the motion of \( M \)'s head. However, it is quite unlikely for our RS-VMs to be able to “shift” something, even by one position.

(iii) We then construct another vector, say \( CS \), which has the same structure as \( C \). (Actually, \( CS \) is also composed of two vectors, say \( TS \) and \( HS \).) The only difference between \( C \) and \( CS \) is as follows. Suppose for example that some section of \( C \) contains sequence \( c(1) c(4) \cdots c(9) c(6) \). Then the corresponding section of \( CS \) contains the “shifted” sequence \( c(6) c(1) c(4) \cdots c(9) \). (See Section 5.3 for details.)

(iv) We then modify \( TS \) and \( HS \) into \( NTS \) and \( NHS \), where a pair of \( NTS \) and \( NHS \) contains “one-move-after” configurations according to \( M \)'s transition function. (\( NTS \) and \( NHS \) stand for “next” \( TS \) and \( HS \), respectively.) By comparing \( T \) (resp. \( H \)) with \( TS \) (resp. \( NHS \)), we can determine the \((i + 1)\)st configuration \( c_{i+1} \) is the proper successor to \( c_i \) in terms of \( M \)'s transition function. A single section is said to be proper if all the successions in that section are proper.

We regard \( C \) as matrices such that each row contains a configuration sequence. Then we construct a column vector, say \( ACC \), such that the \( j \)th element of \( ACC \) is 2 iff the configuration sequence in the \( j \)th section (\( j \)th row) of \( C \) (i.e., \( T \) and \( H \)) is not proper and is 1 (resp. 0) iff the \( j \)th section of \( C \) is proper and ends with an accepting configuration (resp. rejecting configuration). Now one can see that \( V \) accepts the input if and only if \( ACC \) contains at least one 1, which can be checked easily by using the \( \_ \)-contract operation. See Section 5.4 for details.

5.2. Construction of \( C \)

We first construct the following vector \( T_1 \) of length \( N^{2\cdot 2^{N^2}} \), which can be constructed in the similar way to \( T_{(n)} \) (defined in Section 4.2). (The reason \( T_1 \) has length \( N^{2\cdot 2^{N^2}} \) is given later.) Note that the length of the division is \( N \cdot 2^N \) and the length of \( 2^{N(N-1)} \cdot 2^N \) divisions is \( N^{2\cdot 2^{N^2}} \).

\[
T_1 = \left( S_N(0), S_N(1), \ldots, S_N(2^{N^2}-1) \right)^{2^{N(N-1)} \cdot 2^N}
\]

\[
N \times N \quad N \times N \quad N \times N
\]

\[
\begin{array}{cccc}
0 & \cdots & 0 & 0 \\
0 & \cdots & 0 & 1 \\
0 & \cdots & 1 & 1 \\
\end{array}
\]

\[
T_1 \text{ contains all the different tapes of length } N. \text{ We then construct the following vector } H_1 \text{ of the same length as } T_1. \text{ } H_1 \text{ contains all the different state-head-positions.}
\]
(Recall that value \((j - 1)N + i\) represents state \(s_j\) and head-position \(i\).) We omit explaining how to construct \(H_1\).

\[
H_1 = ((H_N(1))^{2^N}, (H_N(2))^{2^N}, (H_N(3))^{2^N}, \ldots, (H_N(kN))^{2^N}, \ldots, (H_N(2^N))^{2^N})^{2^{N(N-1)}}
\]

The maximum value for the state-head-positions is \(kN\). To adjust the length, however, \(H_1\) holds “dummy” subvectors holding larger values (from \(kN + 1\) through \(2^N\)). In the following, for simplicity, we regard the number of all the different state-head-positions as \(2^N\) and hence the number of all the different configurations is \(L = 2^{2^N}\).

One can find all the configurations in the pair of \(C_1\) and \(H_1\) as follows.

\[
T_1 = S_N(0) S_N(1) \ldots S_N(2^N - 1) \quad H_1 = H_N(1) H_N(2) \ldots H_N(1) \quad H_N(2) H_N(2) \ldots H_N(2) \quad \ldots
\]

In each section of the above pair, there are \(N\) pairs of subvectors all of which are the same. We sometimes regard \(T_1\) and \(H_1\) as the following single vector \(C_1\).

\[
C_1 = (c(1) c(1) \ldots c(1)) c(2) c(2) \ldots c(2) (c(3) c(3) \ldots c(3)) \ldots (c(L) c(L) \ldots c(L))^{2^{N(N-1)}}
\]

The difference between \(C\) and \(C_1\) should be observed carefully. What we really want is not all the configurations \((= C_1)\) but all the \(N\)-step sequences of configurations \((= C)\). In each section of \(C_1\), \(c(i)\) is repeated \(N\) times, which is similar to \(T_n(h)\) in Section 4.2 In each section of \(T_n(h)\), there were \(n/2^h\) subvectors of length \(2^h\), all of
which were the same. As in Section 4.2, by concatenating every pair of configurations, we obtain a vector, say $C_2$, which contains all the different 2-step configuration sequences. Then by concatenating every pair of 2-step sequences, we obtain a vector, say $C_4$. Continuing this procedure, we finally obtain a vector, say $C_N$ ($= C$), which contains all the different $N$-step configuration sequences.

Recall that $C_1$ consists of $2^{2(N-1)}$ subvectors each of which contains $2^{2N}$ different 1-step configuration sequences. After the single round of the above procedure, we can obtain all the different 2-step sequences, the number of which is $(2^{2N})^2$. That means the number of repeated subvectors decreases from $2^{2(N-1)} = 2^{2N}/(2^{2N})^2$ to $2^{2N}/(2^{2N})^2$. After log $N$ rounds, we obtain all the $N$-step sequences, the number of which is $(2^{2N}/(2^{2N})^2)^{2^{logN}}$. Note that $2^{2N}/(2^{2N})^2 = 1$, which means the number of the repeated subvectors finally becomes one.

5.3. Construction of $CS$

Recall that the procedure in Section 5.2 produced, say, 8-step configuration sequences

$$c(1) c(16) c(4) c(12) c(8) c(17) c(2) c(14).$$

Intuitively, this sequence is obtained from the following two sequences

$$c(1) c(16) c(4) c(12) c(1) c(16) c(4) c(12)$$

$$c(8) c(17) c(2) c(14) c(8) c(17) c(2) c(14)$$

by combining the underscored portions. Now one can see the following shifted 8-step sequence of (9)

$$c(14) c(1) c(16) c(4) c(12) c(8) c(17) c(2)$$

can be obtained from

$$c(12) c(1) c(16) c(4) c(12) c(1) c(16) c(4)$$

$$c(14) c(8) c(17) c(2) c(14) c(8) c(17) c(2),$$

where (12) and (13) are shifted sequences of (10) and (11), respectively.

That is the basic idea of obtaining the shifted $N$-step configuration sequences, $CS$. The procedure is exactly the same until constructing $C_1$. Note that we can regard this $C_1$ as the vector $CS_1$ containing all the different shifted 1-step configuration sequences. Starting with $CS_1$, we construct $CS_2$, which contains all the different shifted 2-step sequences, and we then construct $CS_4$, and so on, up to $CS_N$ ($= CS$). The only difference between constructing $C$ and $CS$ is as follows: To construct $C$, we needed $E_{N}(h)$ to extract a pair of two portions shown in (10) and (11). (Here, the definition $E_{N}(h)$ is the same as $E_{h}(h)$ in Section 4.2, except that the exterior exponent of $E_{N}(h)$ is now $2^{2N}$ (and not $2^{N^2}$) in order that $E_{N}(h)$ has the same length as $T_1$ and $H_1$.) Instead of $E_{N}(h)$, we use its “shifted” version $ES_{N}(h)$:
Constructing $ES_N(h)$ from $EN_N(h)$ is as follows. Consider the following vector $EN_N(h)$ and $FN_N(h)$.

$$\begin{align*}
EN_N(h) &= \left[ \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
\end{array} \right] \\
FN_N(h) &= \left[ \begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array} \right] \\
ES_N(h) &= \left[ \begin{array}{cccccccc}
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \right]
\end{align*}$$

We compute $G_1 := FN_N(h) + FN_N(h) \& EN_N(h)$ and $G_2 := FN_N(h) \& G_1$. Then $ES_N(h) := EN_N(h) + G_1 \& G_2$. The construction of $FN_N(h)$ may be left to the reader.

5.4. Choosing Proper Sequences

We have constructed $C$ (i.e., $T$ and $H$) and $CS$ (i.e., $TS$ and $HS$), which contain all the different $N$-step sequences $c_1 c_2 \cdots c_N$ and the shifted sequences $c_N c_1 c_2 \cdots c_{N-1}$, respectively. Any accepting sequence must satisfy that (i) its first configuration $c_1$ is the initial configuration of $M$ ($M$'s input, an initial state, and head-position 1), (ii) each $c_{i+1}$ is the proper successor to $c_i$, and (iii) the final configuration includes an accepting state. In order to check whether those (i), (ii), and (iii) are satisfied, we introduce (column) vectors $INIT$, $SUC$, and $FIN$ of length $2^{2N^2}$, respectively. We will obtain $ACC$, introduced in Section 5.1, from these three vectors. In the rest of the proof, we regard the vectors $C$ and $CS$ of length $N^2 2^{2N^2}$ as matrices of $N^2$ columns and $2^{2N^2}$ rows. Note that if we contract these matrices log $N^2$ times, then a single row, composed of $N^2$ elements, is compressed to a single element, and thus we obtain column vectors of length $2^{2N^2}$.

**INIT.** $INIT$ is a $0/2$-vector such that $INIT[i] = 2$ if and only if the first configuration of the sequence in the $i$th row of the matrix is not proper. We first construct the following two vectors.

$$T_{INIT} = (T_{INIT})^{N \times N} = (1 \cdots 1 0 0 \cdots 0 \cdots 0) 2^{2N^2}$$

$$T_{IN} = (T_{IN})^{N \times N} = (i_1 i_2 \cdots i_N 0 0 \cdots 0 \cdots 0) 2^{2N^2}$$

Recall that $i_1 i_2 \cdots i_N$ is the input string of $M$. $(T_{INIT})$ can be obtained by applying the coincide operation to $((\Omega(N))^5) 2^{2N^2}$ and $(\Omega(N^2))^2 2^{2N^2}$. To obtain $T_{IN}$, we first “cut” the input $IN$ by adding $(00 \cdots 0)$ of length $N^2$ to $IN$. Recall that the vector...
IN holds integers $\geq 2$ except the first $n$ elements. By changing all elements greater than 1 into 0’s and then repeating the result $2N^2$ times, we obtain $T_{IN}$. Using $T_{INIT}$ and $T_{IN}$, it is not hard to obtain $INIT_T$ of length $2^{2N^2}$ such that $INIT_T[i] = 2$ if and only if the first $n$ bits of the $i$th row of $T$ do not coincide with the input symbols $i_1i_2 \cdots i_n$. Similarly, we construct $INIT_H$ such that $INIT_H[i] = 2$ if and only if the sequence of $H$’s $i$th row does not start with the initial state of $M$ or head-position 1. $INIT$ can be obtained by $(INIT_T - INIT_H) + INIT_H$.

FIN. FIN is a 0/1-vector such that $FIN[i] = 1$ if and only if the last configuration of the $i$th row is an accepting configuration. How to construct FIN is omitted.

SUC. We change each shifted configuration sequence $c_{2N}c_2 \cdots c_{2_n}c_{-1}$ in $CS$ into $c_{2N}c_2'c_2' \cdots c_{2_n}c_{-1}$, where $c_i'$ is the one-move-after configuration of $c_i$.

The following procedure is applied to state $s_1$, then to state $s_2$, and so on. Suppose that we are now doing the job for state $s_j$. See Fig. 2. The subvectors of length $N$ in $T, H, TS,$ and $HS$ are called blocks. We first construct a 0/1-vector, say $ST(j)$, which has 1’s in the positions where vector $HS$ has values from $(j - 1)N + 1$ through $jN$ (i.e., corresponding to state $s_j$). (The example given in Fig. 2 is when $N = 8$ and $j = 4$). $ST(j)$ is obtained by using the mask operation and the Booleanization operation (details are omitted). By masking $HS$ by $ST(j)$ and then subtracting the same value $(j - 1)N$ from all the elements, we can obtain a vector, say $HS(j)$, which contains values from 1 through $N$, i.e., the head position, only in the blocks associated with state $s_j$ (other blocks are filled by 0’s). Similarly, $TS(j)$ is obtained by masking $TS$ with $ST(j)$. Also we need another vector, say $HP(j)$, which has 1 only in the $p$th position of every block, where $p$ is the head position of every block given by $HS(j)$. Applying the coincide operation to $(1, 2, ..., N)^{2N^2}$ and $HS(j)$ gives $HP(j)$.

Now we decompose $HP(j)$ into $HP_{0j}(j)$ and $HP_{1j}(j)$, such that (i) $HP(j) = HP_{0j}(j) + HP_{1j}(j)$ and (ii) $HP_{0j}(j)$ (resp. $HP_{1j}(j)$) holds 1’s in the positions where $HP_{0j}(j)$ holds 1’s and $TS(j)$ holds 1’s (resp. 0’s); i.e., the symbol under the head is 1 (0, respectively). $HP_{0j}(j)$ and $HP_{1j}(j)$ are obtained by $HP_{0j}(j) := HP(j) - TS(j)$ and $HP_{1j}(j) := HP(j) - HP_{0j}(j)$.

![FIGURE 2](image-url)
Now using $H_{P_d}(j)$ and $H_{P_s}(j)$, we can modify $T_S(j)$ into $N_T(j)$, which stands for the "next" $T_S(j)$. Intuitively, $N_T(j)$ is obtained by replacing every tape with a "one-move-after" tape of TM $M$. $N_T(j)$ is computed according to the transition function of $M$ as follows:

$$N_T(j) = \begin{cases} T_S(j) + H_{P_d}(j) - H_{P_s}(j) & \text{if $M$ on state $s_j$ changes symbol 0 to 1 and 1 to 0,} \\ T_S(j) + H_{P_d}(j) & \text{if $M$ on state $s_j$ changes symbol 0 to 1 and 1 to 1,} \\ T_S(j) - H_{P_s}(j) & \text{if $M$ on state $s_j$ changes symbol 0 to 0 and 1 to 0,} \\ T_S(j) & \text{if $M$ on state $s_j$ changes symbol 0 to 0 and 1 to 1.} \end{cases}$$

$N_{HS}(j)$, next $H_{S}(j)$, can be computed in a similar manner. $N_{HS}(j)$ has value $p'$ in every position where (i) $H_{P}(j)$ has 1 ($p'$ is the state-head-position value of one-move-after) and (ii) value 0 in every position where $H_{P}(j)$ has 0. Thus, unlike $H_{S}(j)$, each block (associated with $s_j$) of $N_{HS}(j)$ has at most one nonzero value, which represents the state-head-position of one-move-after (others are set to 0's).

After completing above procedure for all $k$ states, we sum up the vectors as follows:

$$N_T = N_T(1) + N_T(2) + \cdots + N_T(k),$$
$$N_{HS} = N_{HS}(1) + N_{HS}(2) + \cdots + N_{HS}(k).$$

In the pair of $N_T$ and $N_{HS}$, there are all the different shifted "one-move-after" configuration sequences $c_1^N c_2^N c_3^N \cdots c_{N-1}^N$. If $c_1^N = c_2^N = c_3^N = \ldots = c_{N-1}^N = c_N^N$, then the sequence $c_1^N c_2^N c_3^N \cdots c_N^N$ of configurations is proper. To see if that is true, we make a 0/2-vector, $SUC_T$, of length $N^2 2^{2N^2}$, which has 2's in the positions where $N_T$ and $T$ do not coincide except for the first 2$N$ bits of each row. (Recall that these positions have been checked by $INIT$.) Formally, $SUC_T[i] = 2$ if and only if (i) $N_T[i] \neq T[i]$ and (ii) $T_{INIT}[i] = 0$. Similarly, we also make a 0/2-vector, $SUC_H$, of length $N^2 2^{2N^2}$ such that $SUC_H[i] = 2$ if and only if (i) $N_{HS}[i] \neq H[i]$ and (ii) $T_{INIT}[i] = 0$ and (iii) $N_{HS}[i] = 0$ (we should pay attention only to the positions the head exists). (Although the details are omitted, $SUC_T$ and $SUC_H$ are obtained by using the coincide and mask operations.) Let $SUC = (SUC_T - SUC_H)$ + $SUC_H$. If $SUC$ holds at least one 2 in some row, then the row is not proper; i.e., the sequence of configurations does not follow the transition function. By contracting $SUC$ log $N^2$ times, we obtain 0/2-vector $SUC$ of length $2^{2N^2}$.

Now one can see that

$$ACC(2^{2N^2}) := (FIN - INIT - SUC) + INIT + SUC$$

satisfies the condition given at the beginning of this section.
5.5. \textit{RSTIME}(2, \textit{poly}) \supseteq \textit{ATIMALT}(\textit{poly}, \textit{poly})

Suppose that the ATM halts within \(N\) steps. Instead of all the \(N\)-step sequences of configurations in the previous sections, we now construct the \((N \times A)\)-step sequences, where \(A - 1\) is the number of alternations and is polynomial in \(n\). For simplicity, we assume that \(A\) is a power of two and that the number of nondeterministic choices are exactly two for any configuration. We construct the following vectors \(T_2\) and \(H_2\) instead of \(T_1\) and \(H_1\), respectively.

\[
T_2 = (0 \ldots 00 \cdots 00) \quad 0 \ldots 01 \cdots 0 \quad 1 \ldots 11 \cdots 11 \quad 2^{N(A - 1)} \cdot 2^N
\]

\[
H_2 = ((1)^N (1)^N \cdots (1)^N) \quad ((2)^N (2)^N \cdots (2)^N) \quad \cdots \quad ((2^N)^N (2^N)^N \cdots (2^N)^N) \quad 2^{N(A - 1)}
\]

We then construct four vectors \(T\), \(TS\), \(H\), and \(HS\) from \(T_2\) and \(H_2\) by the same technique as in Section 5.2. A pair of \(T\) and \(H\) (resp. \(TS\) and \(HS\)) now contains all the different \((N \times A)\)-step configuration sequences \(c_1 c_2 \cdots c_{N \times A}\) (resp. shifted sequences \(c_{N \times A} c_1 c_2 \cdots c_{N \times A - 1}\)). The length of all those vectors is \(N^2 A \cdot 2^{N \times A}\).

Without loss of generality, we can assume that the ATM \(M\) begins its operation in an \(\exists\)-state and terminates in a \(\forall\)-state. Ordinarily, an ATM can change its state from a \(\forall\)-state to an \(\exists\)-state (and vice versa) at arbitrary steps during its computation. However, we may assume without loss of generality that our ATM \(M\) is normalized in the sense that it changes its state from a \(\forall\)-state to an \(\exists\)-state (and vice versa) only at predetermined points, i.e., after every \(N\)th step. This causes no problem if we allow each configuration to appear repeatedly; i.e., \(c_i\) is one of \(c_{i - 1}\)'s proper successors if \(c_i = c_{i - 1}\). Thus, when \(N = 8\), the configuration sequence \(c_1 c_2 \cdots c_{N \times A}\) is partitioned like:

\[
\begin{align*}
&c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 \quad c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} \quad c_{17} c_{18} c_{19} c_{20} c_{21} c_{22} c_{23} c_{24} \\
&c_25 c_{26} c_{27} c_{28} c_{29} c_{30} c_{31} c_{32} \quad \cdots \\
&N \text{ steps} \quad N \text{ steps} \quad N \text{ steps} \\
\exists\text{states} \quad \forall\text{states} \quad \exists\text{states} \\
N\text{-steps} \quad \forall\text{states}
\end{align*}
\]
As in Section 5.4, we regard a pair of \( T \) and \( H \) (and \( TS \) and \( HS \)) as a matrix of \( N^2A \) columns and \( 2^{N^2A} \) rows, where each row corresponds to a section of length \( N^2A \) and contains an \((NA)\)-step configuration sequence. We again construct a column vector \( ACC(2^{N^2A}) \) of length \( 2^{N^2A} \), which contains 0, 1, and 2 as its elements.

It should be noted that the matrix contains all the different \((NA)\)-step configuration sequences and that they appear in lexicographical order; i.e., the matrix contains the sequences in the following order:

\[
\begin{align*}
\text{NA steps} \\
\text{c(1) \cdots c(1) c(1)} \\
\text{c(1) \cdots c(1) c(2)} \\
\text{c(1) \cdots c(1) c(3)} \\
\vdots \\
\text{c(L) \cdots c(L) c(L)}.
\end{align*}
\]

Now one can associate the matrix with the tree shown in Fig. 3. Note that this tree contains \( M \)'s alternating computation (binary) tree as a subgraph; for leaves not included in this computation \( ACC(2^{N^2A}) \) holds 2 at the corresponding place. Recall that \( L = 2^{2^N} \) and that the height of the tree is \( NA \). Thus, the tree has \( L^{NA} = 2^{2^{N^2A}} \) leaves. Now here is the key observation: Suppose that we have a vector \( W \) of length \( L^{NA} \) (\( = \) the number of the leaves). Then if we contract \( W \) log \( L \) times, the result is the vector of length \( L^{NA-1} \), which happens to be the same as the number of nodes at depth \( NA - 1 \). If we repeat the same operation, then the depth becomes \( NA - 2 \), and so on.

As above, we consider \( ACC(2^{N^2A}) \) (\( = ACC(L^{NA}) \)). Since the final \( N \) configurations are \( \forall \)-states, we start with applying the \( \forall \)-contract operation to \( ACC(L^{NA}) \) log \( L \) times, and we obtain a vector, say \( ACC(L^{NA-1}) \), of length \( L^{NA-1} \). Since the second last \( N \) configurations are \( \exists \)-states, we apply the \( \exists \)-operation to \( ACC(L^{NA-1}) \) log \( L \) times, and we obtain a vector, say \( ACC(L^{NA-2}) \), of length \( L^{NA-2} \). Continuing this procedure \( A - 1 \) times, we finally obtain a vector, say \( ACC(L^{L}) = ACC(2^{2^N}) \). After that, we can apply exactly the same procedure as in Section 5.4.

FIGURE 3
Obtaining $ACC(2^{2N^3_A})$ is the same as before. However, it should be noted that $M$ is now nondeterministic and therefore the construction of $SUC$ is slightly different from that in Section 5.4.

5.6. $RSTIME(m, poly) \supseteq ATIMALT(2^{m^d}, poly)$

Since the proof of Lemma 3 (i.e., $RSTIME(m, poly) \supseteq ATIMALT(2^{m^d}, poly)$) is similar to that of Lemma 1 (i.e., $RSTIME(2, poly) \supseteq ATIMALT(poly, poly)$) in Section 5.5, it is enough to point out the difference. Also see Iwama and Iwamoto (1996) for a general $d(m) \geq m$.

We first construct the following $C$, which contains all the different $2^{\cdot \cdot \cdot 2^{N^3} + A}$-step configuration sequences of TM $M$, where $A$ is the number of alternations.

\[ C = \begin{pmatrix}
2^N & 2^N & 2^N \\
\vdots & \vdots & \vdots \\
2^N & 2^N & 2^N
\end{pmatrix}
\]

Like $T_3(h)$ in Section 4.3, $C$ contains dummy portions, filled by 2's. The number of all the different tapes of length $2^N$ is $2^{2^N}$. As in Section 5.1, we regard the number of all the different state-head-positions is $2^{2^N}$. Thus the number $L$ of all the different configurations is $L = (2^{2^N})^2 = 2^{2^{2^N+1}}$, and the number of all the different $2^{2^N + A}$-step sequences is therefore $L^{2^{2^N + A+1}} = 2^{2^{2^{2^N+1}}}$. Since the length of each section of $C$ is $2^{2^{2^N+1}}$, the length of $C$ is $2^{2^{2^N+2}}$. The construction of $C$ is a simple combination of the techniques in Sections 4.3 and 5.2.

We also construct the vector $CS$, which contains all the different “shifted” $2^{2^N + A}$-step sequences. As in Section 5.4, we regard $C$ and $CS$ as matrices of $2^{2^{2^N+1}}$ columns and $2^{2^{2^N+1}}$ rows, where each row corresponds to a configuration sequence. Then we construct column vector $ACC(2^{2^{2^N+1}})$ of length $2^{2^{2^N+1}}$.

For a reason similar to before, the configuration sequence $c_1 c_2 \cdots c_{2^{2^N + A}}$ of the ATM $M$ (when $2^N = 4$) is partitioned like:

\[
\begin{align*}
&c_1 c_2 c_3 c_4 \quad c_5 c_6 c_7 c_8 \quad c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} \\
&2^N \text{ steps} \quad 2^N \text{ steps} \quad 2 \cdot 2^N \text{ steps} \\
&3\text{-states} \quad \forall\text{-states} \quad 3\text{-states} \\
&c_{17} c_{18} c_{19} c_{20} c_{21} c_{22} c_{23} c_{24} c_{25} c_{26} c_{27} c_{28} c_{29} c_{30} c_{31} c_{32} \cdots \\
&2^2 \cdot 2^N \text{-steps} \\
&\forall\text{-states}
\end{align*}
\]
Consider the $M$'s computation (binary) tree in Fig. 4, which has $2^{2N+4+1}$ leaves. (Note that the height of the tree is $2^{2N+4+1}$. ) Suppose that we have a vector $W$ of length $2^{2N+4+1}$ ($=$ the number of leaves). Then if we contract $W$, the result is the vector of length $2^{2N+4}$, which is the same number of nodes at depth $2^{2N+4}$. If we repeat the same operation, then the depth now becomes $2^{2N+4-1}$, and so on. This explains the increasing sizes of the blocks of instructions between two alternations.

Since the final configurations are $\exists$-states, we start with applying the $\forall$-contract operation to $\text{ACC}(2^{2N+4+1})$, and we obtain a vector, say $\text{ACC}(2^{2N+4})$, of length $2^{2N+4}$. By applying the $\exists$-contract operation to $\text{ACC}(2^{2N+4})$, we obtain a vector, say $\text{ACC}(2^{2N+4-1})$, of length $2^{2N+4-1}$. By continuing this procedure $A$ times, we finally obtain a vector, say $\text{ACC}(2^{2N})$. Since the first $N$ configurations are $\forall$-states, $M$ accepts the input string if and only if $\text{ACC}(2^{2N+1})$ contains at least one 1 (use $\exists$-contract).

6. RSTIME$(m, \text{poly}) \subseteq \text{ATIMALT}(2^{\text{poly}}, \text{poly})$

The basic strategy is the same as in Hartmanis and Simon (1974), where the simulation is carried out by space-bounded TMs instead of by alternation-bounded TMs in this paper. Recall that our TMs can spend exponential time but only polynomial alternations. Let $V$ be a program that runs on an RS-VM when $d(m)=m$ and halts within (again polynomial) $T(n)$ steps. We construct an ATM which simulates $V$ in $O(2^{T(n)})$ steps for a constant $c$ and $O(T(n))$ alternations. Repeating (or stretching) a vector constant of length $l$ yields a vector of length $l2^{T(n)}$. Since $l2^{T(n)} \leq 2^{2cT(n)}$ for some constant $c_1$, there is a constant $c$ such that the length of any vector cannot surpass $2^{2cT(n)}$. Therefore, even if we use a special vector $\Omega = (0, 1, 2, ...)$, the value of each element cannot surpass $2^{2cT(n)}$. 

FIG. 4. Computation tree.
What M does first is to guess a specific execution sequence of program V that ends with an accepting state; in other words, a sequence Q of RS-VM instructions whose length q is \( T(n) \) or less. It is clear that M can generate Q in polynomial time. After that M calculates the values of all scalar variables at each step along Q. If necessary, M guesses their values. For example, if M encounters "\( x := A[1] \)" instruction, then M must guess the value for \( x \) (\( \leq 2^{2^{T(n)}} \)). Since \( x \) is represented by at most \( 2^{2^{T(n)}} \) bits, M can guess its value in \( O(2^{2^{T(n)}}) \) steps. At the same time M calculates the size of all vector variables at each step along Q. All those data (Q, scalar values, and vector sizes) are written on M’s tape, which can be accessed in exponential steps whenever necessary. (In what follows, “exponential steps” means \( O(2^{c_2 T(n)}) \) steps for some constant \( c_2 \).) At this moment, M verifies that there is no inconsistency between the sequence Q and the values of the scalar variables. For example, if Q includes

\[
\begin{align*}
&\text{if } x > 0 \text{ then } l_1; \\
l_1: &\text{ \ldots \ldots \ldots \ldots ;} \\
&\text{\ldots}
\end{align*}
\]

then \( x \) must be positive at that step. If there is any inconsistency, M stops in a rejecting state.

Now M checks whether or not Q is actually executable by simulating Q backward (from the end at step q to the beginning). It is convenient to introduce recursive procedure calls of the following type "\( A[i] \text{ at } t \text{ is } a \)”, which means the \( i \)th element of vector \( A \) holds value \( a \) after the execution of the \( t \)th instruction of Q. We assume without loss of generality that none of \( \downarrow, \rightarrow, \uparrow, \leftarrow, +, \text{ or } - \) are applied to vector constants.

Let us look at a small example: Suppose that Q ends with

\[
\begin{align*}
&\text{\ldots} \\
&\text{“step q } - 3 \text{ } A := B + C \text{ ;"} \\
&\text{“step q } - 2 \text{ } x := A[1] \text{ ;“} \\
&\text{“step q } - 1 \text{ if } x > 0 \text{ then } l_1;\text{“} \\
&\text{“step q } l_1: \text{ accept \ldots ;“} \\
\end{align*}
\]

M first calls “\( A[1] \text{ at } q - 2 \text{ is } a \)”, where \( a \) is the value of \( x \) guessed earlier. In that procedure M again calls universally “\( B[1] \text{ at } q - 3 \text{ is } b \)” and “\( C[1] \text{ at } q - 3 \text{ is } a - b \)” where \( b \) (\( \leq 2^{2^{T(n)}} \)) is existentially guessed before the universal branch. Since \( b \) is double exponential, it can be written on the M’s tape in a (single) exponential number of bits and in roughly the same amount of time.

In general, suppose that “\( A[i] \text{ at } t \text{ is } a \)” is now called. Then M looks at the \( i \)th instruction of Q. Several cases exist as follows: One should observe that, in each
case, (i) $M$ needs to change its state from universal ones to existential ones (or vice versa) at most constant times and (ii) the number of steps required is at most exponentially large.

1. $x := A[1]$ (i.e., $i = 1$). If $x = a$, then $M$ simply calls $"A[1] at t - 1 is a"$; otherwise $M$ stops in a rejecting state (the value for variable $x$ guessed before is not correct).

2. The instruction does not change $A[i]$ and is not the same as case 1 (such as $B := C + A$ or those having nothing to do with vector $A$), then $M$ simply calls $"A[i] at t - 1 is a"$.

3. $A := B \cdot C$ ( i.e. + or - ). Then $M$ guesses two values $b$ and $c$ (an exponential number of bits) such that $a = b \cdot c$ and calls universally $"B[i] at t - 1 is b"$ and $"C[i] at t - 1 is c"$. It should also be noted that the lengths of $B$ and $C$ may be different. Suppose that those lengths were guessed such that $|B| > |C|$, then it might happen that some portion of $B$ will never be touched in the simulation. That does not cause, however, any problems in this simulation.

4. $A := IN$. If $a = IN[i]$, $M$ stops in an accepting state; otherwise rejects. Similarly for $A := K$ for a vector constant $K$.

5. $A := \rightarrow B$. $M$ calls $"B[j] at t - 1 is a"$, where $j = [i/d(l_B)]$ and $l_B$ is the length of $B$ at $t - 1$. Similarly for $\leftarrow$.

6. $A := \leftarrow B$. Suppose that $|A| = l$ currently. If $a = 0$ then $M$ calls universally

$$"B[(i - 1) \cdot d(l) + 1] at t - 1 is 0",$$
$$"B[(i - 1) \cdot d(l) + 2] at t - 1 is 0"$$
$$\vdots \quad \vdots$$
$$"B[(i - 1) \cdot d(l) + d(l)] at t - 1 is 0".$$ Note that $d(l)$ may be double-exponentially large. Therefore, we need to use a binary tree structure to realize this universal branching in single-exponential steps. (Recall that the length of vectors cannot surpass $2^{2^{2^c}}$.) If $a > 0$, $M$ first guesses value $k$ ($1 \leq k \leq d(l)$). Then $M$ universally calls

$$"B[(i - 1) \cdot d(l) + 1] at t - 1 is b_1",$$
$$\vdots$$
$$"B[(i - 1) \cdot d(l) + k - 1] at t - 1 is b_{k-1}"$$
$$"B[(i - 1) \cdot d(l) + k] at t - 1 is a"$$
$$"B[(i - 1) \cdot d(l) + k + 1] at t - 1 is b_{k+1}"$$
$$\vdots$$
$$"B[(i - 1) \cdot d(l) + d(l)] at t - 1 is b_{d(l)}".$$
where \( b_i \) is existentially guessed to be 0 or \( a \) after the universal branch. Similarly for \( \uparrow \).

Remark. Although the proof is omitted, the above proof is still true if \( V \) is an alternating RS-VM.

7. CONCLUSION

In this paper, we introduced RS-VMs as a canonical form of vector machines. It was shown that we can change the power of RS-VMs depending on the single expansion factor \( d(m) \) of the vector operations. Here are several questions and sources for future research.

1. In section 3, we conjectured that the computational power given in Theorem 4 is the best possible for RS-VMs of any expansion factor. Is this true?

2. If the above conjecture is correct, is there any (a different kind of) global operation set whose power increases unlimitedly as the expansion factor (or a similar parameter) grows?

3. Are there any expansion factors which characterize natural complexity classes between DSPACE(\( \text{poly} \)) and ATIMALT(\( 2^{\text{poly}} \), \( \text{poly} \))?

4. There can be some mechanism in parallel computation (such as, e.g., the degree of fan-in and fan-out of logic gates in circuit complexity) which can explain the expansion factor from a different angle. Is there any?

ACKNOWLEDGMENTS

The authors thank Peter van Emde Boas and two anonymous referees for their useful comments on the initial drafts of this paper.

Received May 7, 1993; final manuscript received September 22, 1997

REFERENCES


