

A note on two-pebble automata over infinite alphabets

Michael Kaminski

Department of Computer Science
Technion – Israel Institute of Technology

Tony Tan

Department of Computer Science
Technion – Israel Institute of Technology

Abstract. It is shown that the emptiness problem for two-pebble automata languages is undecidable and that two-pebble automata are weaker than three-pebble automata.

Keywords: pebble automata, infinite alphabet, undecidability, Hilbert's tenth problem

1. Introduction

Pebble automata (PA)¹ over infinite alphabets were introduced in [7] and later found applications in XML, see [6].

As it has been shown in [7], the notion of PA over infinite alphabets is very robust. In addition to equivalence of various models of PA, the set of languages they accept is closed under *all* boolean operations. However, the emptiness problem for these languages is undecidable.

Reducing Hilbert's tenth problem to the emptiness problem for 2-PA languages, we prove that the latter is undecidable.² In addition, we present an example of a 3-PA language that is not accepted by

Address for correspondence: Michael Kaminski, Department of Computer Science, Technion – Israel Institute of Technology, Haifa 32000, Israel. Email: kaminski@cs.technion.ac.il.

¹We shall also write k -PA for k -pebble automata. Note that this notation differs from that in [7], where, in particular, 2-PA denotes two-way pebble automata.

²It should be noted that reductions of Hilbert's tenth problem for proving undecidability are very rare in Computer Science.

2-PA.³ We also show that 2-PA can accept a *non-semi-linear* language.⁴

This note is organized as follows. In the next section we recall the definition of PA. Section 3 deals with 2-PA languages. In particular, we show that these languages are not necessarily semi-linear, their emptiness problem is undecidable, and present an example of a 3-PA language that is not accepted by 2-PA.

2. Pebble automata

First we recall the definition of pebble automata from [7]. We shall use the following notation: Σ is a fixed infinite alphabet not containing the left-end marker \triangleleft or the right-end marker \triangleright . The input word to the automaton is of the form $\triangleleft w \triangleright$, where $w \in \Sigma^*$. We also assume that Σ contains the symbol $\$$ that will be used as a delimiter.

Symbols of Σ are denoted by lower case letters a, b, c , etc., possibly indexed, and words over Σ are denoted by lower case letters u, v, w , etc., possibly indexed.

Definition 2.1. A *non-deterministic two-way k -pebble automata*, in short, *k -PA*, over Σ is a tuple $\mathcal{A} = \langle Q, q_0, F, \mu \rangle$ whose components are defined as follows.

- $Q, q_0 \in Q$ and $F \subseteq Q$ are a finite set of *states*, the *initial state*, and the set of *final states*, respectively; and
- μ is a finite set of transitions of the form $\alpha \rightarrow \beta$ such that
 - α is of the form (i, a, P, V, q) or (i, P, V, q) , where $i \in \{1, \dots, k\}$, $a \in \Sigma \cup \{\triangleleft, \triangleright\}$, $P, V \subseteq \{1, \dots, i-1\}$, and
 - β is of the form (q, action) , where $q \in Q$ and

$$\text{action} \in \{\text{left}, \text{right}, \text{place-pebble}, \text{lift-pebble}\}.$$

Transitions of the form $(i, a, P, V, q) \rightarrow \beta$ are called (i, a, P, V) -transitions and transitions of the form $(i, P, V, q) \rightarrow \beta$ are called (i, P, V) -transitions.

Given a word $w = a_1 \cdots a_n \in \Sigma^*$, a *configuration of \mathcal{A} on $\triangleleft w \triangleright$* is a triple $[i, q, \theta]$, where $i \in \{1, \dots, k\}$, $q \in Q$ and $\theta : \{1, \dots, i\} \rightarrow \{0, 1, \dots, n, n+1\}$.⁵ The function θ defines the position of the pebbles and is called *pebble assignment*. A configuration $[i, q, \theta]$ with $q \in F$ is called an *accepting configuration*.

A transition $(i, a, P, V, p) \rightarrow \beta$ *applies to a configuration* $[j, q, \theta]$, if

- (1) $i = j$ and $p = q$,
- (2) $V = \{l < i : a_{\theta(l)} = a_{\theta(i)}\}$,

³When this paper was written, this is the first separation example concerning PA. After that a general separation example was presented by the second author in [9]. However, the example in this paper deals with a different language that is of importance for our undecidability result.

⁴A language L is called *semi-linear* if $\{|w| : w \in L\}$ is a finite union of sets of integers of the form $\{l + im : i = 0, 1, \dots\}$, $l, m \geq 0$. It is well-known that regular languages are semi-linear, e.g., see [3, Exercise 3.9, p. 72].

⁵The symbols at the 0 and $n+1$ positions are \triangleleft and \triangleright , respectively, and we assume that \mathcal{A} “knows” these symbols.

(3) $P = \{l < i : \theta(l) = \theta(i)\}$, and

(4) $a_{\theta(i)} = a$.

A transition $(i, P, V, q) \rightarrow \beta$ applies to a configuration $[j, q, \theta]$, if conditions (1)–(3) above hold and no transition of μ of the form $(i, a, P, V, q) \rightarrow \beta$ applies to $[j, q, \theta]$.

We define the transition relation \vdash as follows: $[i, q, \theta] \vdash [i', q', \theta']$, if there is a transition $\alpha \rightarrow (p, \text{action})$ that applies to $[i, q, \theta]$ such that $q' = p$, $\theta'(j) = \theta(j)$ for all $j < i$, and if

- $\text{action} = \text{left}$, then $i' = i$ and $\theta'(i) = \theta(i) - 1$,
- $\text{action} = \text{right}$, then $i' = i$ and $\theta'(i) = \theta(i) + 1$,
- $\text{action} = \text{lift-pebble}$, then $i' = i - 1$,
- $\text{action} = \text{place-pebble}$, then $i' = i + 1$, $\theta'(i) = \theta(i)$, and $\theta'(i + 1) = 0$.

Remark 2.1. In [7], the above model of computation is called *strong* PA. A strictly weaker model in which the new pebble is placed at the position of the head pebble⁶ is referred to as *weak* PA.

As usual, we denote the reflexive, transitive closure of \vdash by \vdash^* . A word $w \in \Sigma^*$ is accepted by \mathcal{A} , if $[1, q_0, \theta_0(1) = 0] \vdash^* [i, q, \theta]$, for some accepting configuration $[i, q, \theta]$ of \mathcal{A} on $\langle w \rangle$, i.e., $q \in F$. The language $L(\mathcal{A})$ consists of all words accepted by \mathcal{A} .

The automaton \mathcal{A} is *deterministic*, if in each configuration at most one transition applies. If $\text{action} \in \{\text{right}, \text{lift-pebble}, \text{place-pebble}\}$ for all transitions, then the automaton is *one-way*.

Theorem 2.1. ([7, Theorem 4.6]) For each $k \geq 1$, non-deterministic two-way k -PA, deterministic two-way k -PA, non-deterministic one-way k -PA, and deterministic one-way k -PA all have the same recognition power.

In view of Theorems 2.1, we will always assume that pebble automata under consideration are deterministic and one way. We can define the hierarchy of languages accepted by PA. For $k \geq 1$, PA_k is the set of all languages accepted by k -PA, and PA is the set of all PA languages. That is,

$$\text{PA} = \bigcup_{k \geq 1} \text{PA}_k.$$

Remark 2.2. It follows from Theorem 2.1 that for each $k \geq 1$, the set of languages PA_k is closed under union, intersection, and complementation.

Theorem 2.2. ([7, Theorem 5.4]) The emptiness problem for 3-PA languages is undecidable.⁷

The proof of [7, Theorem 5.4] is based on the reduction of the *Post correspondence problem* ([8], see also [3, pp. 193–201]) to the emptiness problem for PA languages. Roughly speaking, one of the major technical steps in the proof relies on the fact that the language

$$L_{\text{ord}} = \{a_1 a_2 \cdots a_n \$ a_1 a_2 \cdots a_n : n \geq 1\},$$

⁶That is, in the case of $\text{action} = \text{place-pebble}$, $\theta'(i + 1) = \theta(i)$.

⁷In fact, it was shown in [7] that the emptiness problem is undecidable for 5-PA. However, a closer examination of the proof shows that it applies to 3-PA as well.

$$a_i \neq \$, \text{ for each } i = 1, 2, \dots, n, \text{ and } a_i \neq a_j, \text{ whenever } i \neq j\}$$

is accepted by 3-PA. In contrast, this language is not accepted by 2-PA, see Proposition 3.1 in the next section. This, together with a relative simplicity of 2-PA, might lead to the conjecture that the emptiness problem for 2-PA languages is decidable. However, Theorem 3.1 in the next section shows that (surprisingly?) the emptiness problem for 2-PA languages is undecidable either.

3. Two-pebble automata

This section deals with 2-PA. Its major result is Theorem 3.1 below.

Theorem 3.1. The emptiness problem for 2-PA languages is undecidable.

For the proof of Theorem 3.1 we reduce Hilbert’s tenth problem (existence of solutions of Diophantine equations) to the emptiness problem for 2-PA languages. Namely, we show that the set of solutions of a Diophantine equation is accepted by a 2-PA. Since the former is undecidable ([4], see also [2] or [5]), the latter is undecidable as well.

We precede the proof of Theorem 3.1 with a number of examples exhibiting an unexpectedly strong recognition power of 2-PA. These examples are rather simple, but despite their simplicity, they are the backbones of our subsequent results concerning 2-PA.

The first example deals with the language L_{diff} consisting of all words in which every symbol from Σ occurs at most one time:

$$L_{\text{diff}} = \{a_1 \cdots a_n : n \geq 1, a_i \neq \$, \text{ for each } i = 1, \dots, n, \text{ and } a_i \neq a_j, \text{ whenever } i \neq j\}.$$

The words of L_{diff} will be used for representation of positive integers in “unary” notation: a word $w \in L_{\text{diff}}$ represents the integer $|w|$.⁸ Of course in such way a positive integer has infinitely many equivalent representations, but as we shall see in the sequel, the integer equality can be tested by 2-PA.

Example 3.1. The language L_{diff} is accepted by a 2-PA that works as follows. Pebble 1 advances through the input from left to right. At each step it verifies that the symbol under it is not \$, and then pebble 2 scans the input and verifies that the input symbol under pebble 1 differs from all the others, see also [7, the example in Section 2.4 and Theorem 4.1].

Example 3.2 below employs the following notation. For two words $u, v \in \Sigma^*$ we write $u \sim v$, if one is a permutation of the other.

Example 3.2. Let

$$L_{\text{perm}} = \{u\$v : u, v \in L_{\text{diff}} \text{ and } u \sim v\}.$$

This language is accepted by a 2-PA that works as follows. Pebble 1 advances through the input from left to right. In each step pebble 2 scans the input and finds the symbol under pebble 1 on the other “half” of the input.⁹ Verifying that both u and v are in L_{diff} can be done like in Example 3.1.

⁸Cf. [1, Section 7], where a similar representation was used for the proof of undecidability of the emptiness problem for languages accepted by a kind of a *register* automata called po-2-DFA₁.

⁹Naturally, the first half of the input consists of the symbols occurring before “\$” and the second half of the input consists of the symbols occurring after it.

Our next example shows that positive integers represented by elements of L_{diff} can be tested for equality by 2-PA.

Example 3.3. Let L_{eq} consist of all words of the form

$$u\$a_1b_1 \cdots a_nb_n\$v,$$

where

- $u, v \in L_{\text{diff}}$,
- $a_1 \cdots a_n \sim u$, and
- $b_1 \cdots b_n \sim v$.

This language is accepted by 2-PA that, like in Example 3.1, verifies that both u and v are in L_{diff} , and then, like in Example 3.2, verifies that $a_1 \cdots a_n \sim u$ and $b_1 \cdots b_n \sim v$.

The following example shows that 2-PA can accept non-semi-linear languages.

Example 3.4. Let L_{sq} consist of all words of the form

$$u\$a_1v_1\$a_2v_2\$ \cdots \$a_nv_n,$$

where

- $u \in L_{\text{diff}}$,
- $u \sim a_1 \cdots a_n$,
- $u \sim v_1$
- $v_i \sim v_{i+1}$, for each $i = 1, \dots, n - 1$.

It follows that

$$|u| = |a_1 \cdots a_n| = |v_1| = \cdots = |v_n|,$$

implying

$$\{|w| : w \in L_{\text{sq}}\} = \{n^2 + 3n : n = 1, 2, \dots\}.$$

Thus, L_{sq} is not semi-linear. However, L_{sq} is a 2-PA language, because the membership test involves only verifying permutations of words. Namely, L_{sq} is accepted by a (two-way) 2-PA that first, like in Example 3.3, verifies the first three clauses of the definition and then verifies the last clause by “iterating” the automaton that accepts the language L_{perm} .

The reduction of Hilbert’s tenth problem is based on Examples 3.5 and 3.6 below which illustrate the core idea behind the proof. Namely, these examples show that 2-PA can test atomic integer equalities.

Example 3.5. Let L_{add} be the language consisting of all words of the form

$$u\$v\$a_1c_1 \cdots a_mc_m\$b_1c_{m+1} \cdots b_nc_{m+n}\$w,$$

where

A1. $u, v, w \in L_{\text{diff}}$,

A2. $a_1 \cdots a_m \sim u$,

A3. $b_1 \cdots b_n \sim v$, and

A4. $c_1 \cdots c_{m+n} \sim w$.

Obviously,

$$|u| + |v| = |w|.$$

The language L_{add} is accepted by a (two-way) 2-PA that first, like in Example 3.1, verifies clause A1 and then, like in Example 3.2, verifies clauses A2–A4 by visiting the appropriate positions in the input word.

Example 3.6. Let L_{mult} be the language consisting of all words of the form

$$u\$v\$a_1c_{1,1}b_{1,1} \cdots c_{1,n}b_{1,n}\$ \cdots \$a_m c_{m,1}b_{m,1} \cdots c_{m,n}b_{m,n}\$w,$$

where

M1. u, v and w are in L_{diff} ,

M2. $u \sim a_1 \cdots a_m$,

M3. $v \sim b_{1,1} \cdots b_{1,n}$,

M4. $b_{i,1} \cdots b_{i,n} \sim b_{i+1,1} \cdots b_{i+1,n}$, for each $i = 1, \dots, m - 1$, and

M5. $w \sim c_{1,1} \cdots c_{1,n} \cdots c_{m,1} \cdots c_{m,n}$.

Obviously,

$$|u| \times |v| = |w|.$$

The language L_{mult} is accepted by a (two-way) 2-PA that first, like in Example 3.1, verifies clause M1 and then, “iterating” the automaton that accepts the language L_{perm} , verifies clauses M2–M5 by visiting the appropriate positions in the input word.

Example 3.7 below is a straightforward extension of Example 3.5.

Example 3.7. Let k be a positive integer and let $L_{\text{add},k}$ be the language consisting of all words of the form

$$v_1\$ \cdots \$v_k\$a_{1,1}b_1 \cdots a_{1,|v_1|}b_{|v_1|}\$ \cdots \$a_{k,1}b_{|v_1|+\cdots+|v_{k-1}|+1} \cdots a_{k,|v_k|}b_{|v_1|+\cdots+|v_k|}\$v,$$

where

A1_k. $v_1, \dots, v_k, v \in L_{\text{diff}}$,

A2_k. $a_{i,1} \cdots a_{i,|v_i|} \sim v_i$, $i = 1, \dots, k$, and

A3_k. $b_1 \cdots b_{|v_1|+\cdots+|v_k|} \sim v$.

Obviously,

$$|v_1| + \dots + |v_k| = |v|.$$

The language $L_{\text{add},k}$ is accepted by a 2-PA similar to that described in Example 3.5.

At last, we have arrived at the proof of Theorem 3.1. The intuition behind the proof is as follows. Examples 3.3, 3.5, and 3.6 show how, by verifying only permutations of words, 2-PA can simulate the equality test and can verify the results of the arithmetic operations: addition and multiplication.

For the proof of Theorem 3.1 we (quite naturally) extend these examples to testing results of polynomial evaluation, or, more precisely, to languages corresponding to polynomial evaluation. Namely, we extend the evaluation of $|u| + |v|$ and $|u| \times |v|$ to all polynomials with positive integer coefficients.

Loosely speaking, for a non-constant polynomial $f(x_1, \dots, x_m)$ with positive integer coefficients, the language L_f consists of all words of the form

$$v_1 \$ \dots \$ v_m \$ \text{“an encoding of an evaluation of } f(|v_1|, \dots, |v_m|)\text{”} \$ w,$$

where $v_1, \dots, v_m, w \in L_{\text{diff}}$ and $|w| = f(|v_1|, \dots, |v_m|)$. The meaning of “an encoding of an evaluation of $f(|v_1|, \dots, |v_m|)$ ” will become clear in the course of the proof of Theorem 3.1. We shall see that only permutations of words are needed to check the encoding. Thus, the language L_f is accepted by 2-PA, see the proof of Theorem 3.1 below.

Proof:

of Theorem 3.1 We start with recalling Hilbert’s tenth problem that can equivalently be restated as follows. *Given two polynomials $f'(x_1, \dots, x_m)$ and $f''(x_1, \dots, x_m)$ with positive integer coefficients, do there exist positive integers n_1, \dots, n_m such that*

$$f'(n_1, \dots, n_m) = f''(n_1, \dots, n_m)? \tag{1}$$

It was shown in [4] (see also [2] or [5]) that Hilbert’s tenth problem is undecidable.

First, using Example 3.6, we show that 2-PA can recognize the values of monomials over positive integers. For a monomial $M(x_1, \dots, x_m)$ over positive integers, the language L_M is defined by the following recursion.

- If $M(x_1, \dots, x_m)$ is a constant n , then L_M consists of all words of the form

$$v_1 \$ \dots \$ v_m \$ \$ v,$$

where $v_1, \dots, v_m, v \in L_{\text{diff}}$ and $|v| = n$. That is,

$$M(|v_1|, \dots, |v_m|) = |v| (= n).$$

- If $M(x_1, \dots, x_m)$ is of the form $x_j \widetilde{M}(x_1, \dots, x_m)$, then L_M consists of all words of the form
- If $M(x_1, \dots, x_m)$ is of the form $x_j \widetilde{M}(x_1, \dots, x_m)$, then L_M consists of all words of the form

$$v_1 \$ \dots \$ v_m \$ w \$ v' \$ a_1 c_{1,1} b_{1,1} \dots c_{1,n} b_{1,n} \$ \dots \$ a_\ell c_{\ell,1} b_{\ell,1} \dots c_{\ell,n} b_{\ell,n} \$ v,$$

where

- $v_1 \$ \cdots \$ v_m \$ w \$ v' \in L_{\widetilde{M}}$, i.e., $|v'| = \widetilde{M}(|v_1|, \dots, |v_m|)$;
- $a_1 \cdots a_\ell \sim v_j$;
- $b_{i,1} \cdots b_{i,n} \sim v'$, for each $i = 1, \dots, \ell$; and
- $c_{1,1} \cdots c_{1,n} \cdots c_{\ell,1} \cdots c_{\ell,n} \sim v$.

That is, $|v_j| = \ell$, $|v'| = n$, and

$$|v| = |v_j| |v'| = |v_j| \widetilde{M}(|v_1|, \dots, |v_m|) = M(|v_1|, \dots, |v_m|).$$

Next, with monomials $M_1(x_1, \dots, x_m), \dots, M_k(x_1, \dots, x_m)$ over positive integers, we associate the language L_{M_1, \dots, M_k} that consists of words of the form

$$v_1 \$ \cdots \$ v_m \$ w_1 \$ u_1 \$ \cdots \$ w_k \$ u_k \$ w' \$ w,$$

where

- $v_1 \$ \cdots \$ v_m \$ w_i \$ u_i \in L_{M_i}$, i.e., $M_i(|v_1|, \dots, |v_m|) = |u_i|$, $i = 1, \dots, k$, and
- $u_1 \$ \cdots \$ u_k \$ w' \$ w \in L_{\text{add}, k}$.

That is,

$$M_1(|v_1|, \dots, |v_m|) + \cdots + M_k(|v_1|, \dots, |v_m|) = |w|.$$

Now, let

$$f'(x_1, \dots, x_m) = M'_1(x_1, \dots, x_m) + \cdots + M'_{k'}(x_1, \dots, x_m)$$

and let

$$f''(x_1, \dots, x_m) = M''_1(x_1, \dots, x_m) + \cdots + M''_{k''}(x_1, \dots, x_m),$$

where $M'_{i'}(x_1, \dots, x_m)$, $i' = 1, \dots, k'$, and $M''_{i''}(x_1, \dots, x_m)$, $i'' = 1, \dots, k''$, are monomials. Then, like in Example 3.7, we can “extend” $L_{M'_1, \dots, M'_{k'}, M''_1, \dots, M''_{k''}}$ to the language $L_{f', f''}$ that for each m -tuple $v_1, \dots, v_m \in L_{\text{diff}}$ contains a word of the form

$$v_1 \$ \cdots \$ v_m \$ u' \$ u'' \$ w' \$ w'',$$

where $v_1 \$ \cdots \$ v_m \$ u' \$ w' \in L_{M'_1, \dots, M'_{k'}}$ and $v_1 \$ \cdots \$ v_m \$ u'' \$ w'' \in L_{M''_1, \dots, M''_{k''}}$. That is,

$$|w'| = f'(|v_1|, \dots, |v_m|)$$

and

$$|w''| = f''(|v_1|, \dots, |v_m|).^{10}$$

Finally, let $L_{f'=f''}$ be the languages consisting of all words of the form

$$v_1 \$ \cdots \$ v_m \$ u' \$ u'' \$ w' \$ w'' \$ a'_1 a''_1 \cdots a'_n a''_n,$$

where

¹⁰Note that delimited patterns of $L_{M'_1, \dots, M'_{k'}, M''_1, \dots, M''_{k''}}$ can be detected by using just the finite memory (states) of an appropriate 2-PA.

- $v_1\$ \cdots \$v_m\$u'\$u''\$w'\$w'' \in L_{f',f''}$,
- $a'_1 \cdots a'_n \sim w'$, and
- $a''_1 \cdots a''_n \sim w''$.

Then, like in Example 3.3, one can show that $L_{f'=f''}$ is accepted by 2-PA.

Since, obviously, (1) has a solution if and only if $L_{f'=f''}$ is non-empty, our reduction (and, therefore, the proof of Theorem 3.1) is complete. \square

We conclude this paper with a negative result related to the language L_{ord} defined in the end of Section 2.

Proposition 3.1. The language L_{ord} is not accepted by 2-PA.

Proof:

Assume to the contrary that L_{ord} is accepted by an s -state 2-PA \mathcal{A} . By Theorem 2.1, we may assume that \mathcal{A} is one-way and deterministic. We may also assume that \mathcal{A} is normalized as follows:

- after each move of pebble 1, \mathcal{A} places pebble 2;
- pebble 2 is lifted only when it reaches the end of the input; and
- immediately after pebble 2 is lifted, pebble 1 moves right.

Consider the word

$$u = a_1 \cdots a_n \$ a'_1 \cdots a'_n \in L_{\text{ord}},^{11}$$

where $n > s^5$. Since $u \in L_{\text{ord}} = L(\mathcal{A})$, there is an accepting run of \mathcal{A} on u . In that run, with each integer $i = 1, \dots, n$, we associate a quintuple of states $\langle q_1^i, q_2^i, q_3^i, q_4^i, q_5^i \rangle$ that is defined as follows.

- q_1^i is the state in which pebble 1 arrives at a_i .
- q_2^i is the state in which pebble 1 leaves a_i .
- q_3^i is the state in which pebble 1 arrives at a'_i .
- q_4^i is the state in which pebble 1 is above a_i and pebble 2 arrives at $\$$.
- q_5^i is the state in which pebble 1 is above a'_i and pebble 2 arrives at $\$$.

Since $n > s^5$, there exist states q_k , $k = 1, \dots, 5$, and two indexes i and j , $i < j$, such that

$$\langle q_1^i, q_2^i, q_3^i, q_4^i, q_5^i \rangle = \langle q_1^j, q_2^j, q_3^j, q_4^j, q_5^j \rangle = \langle q_1, q_2, q_3, q_4, q_5 \rangle. \quad (2)$$

We contend that the word

$$v = a_1 \cdots a_i \cdots a_j \cdots a_n \$ a'_1 \cdots a'_j \cdots a'_i \cdots a'_n,$$

obtained from u by switching a'_i and a'_j , is also accepted by \mathcal{A} , in contradiction with $L(\mathcal{A}) = L_{\text{ord}}$.

To proceed we need the following notation. Let w' be a non-empty prefix of w . We denote by $R(w', w)$ the state in the run of \mathcal{A} on w in which pebble 1 leaves the rightmost symbol of w' .

For example, in this notation,

¹¹We use primes to differentiate between occurrences of a symbol before and after $\$$.

- $R(a_1 \cdots a_{i-1}, u) = q_1$,
- $R(a_1 \cdots a_i, u) = q_2$,
- $R(a_1 \cdots a_n \$ a'_1 \cdots a'_{i-1}, u) = q_3$, and
- $R(u, u)$ is a final state of \mathcal{A} .

We shall prove that

$$R(v, v) = R(u, u), \tag{3}$$

which would imply $v \in L(\mathcal{A})$, in contradiction with $v \notin L_{\text{ord}}$.

We start with the proof of the relationship between the runs of \mathcal{A} on u and v given by equations (4)–(12) below, see also Figure 1 on the next page for a graphical representation of these equations.

$$R(a_1 \cdots a_{i-1}, v) = R(a_1 \cdots a_{i-1}, u) = q_1 \tag{4}$$

$$R(a_1 \cdots a_i, v) = R(a_1 \cdots a_j, u) = q_2 \tag{5}$$

$$R(a_1 \cdots a_{j-1}, v) = R(a_1 \cdots a_{j-1}, u) = q_1 \tag{6}$$

$$R(a_1 \cdots a_j, v) = R(a_1 \cdots a_i, u) = q_2 \tag{7}$$

$$R(a_1 \cdots a_j, v) = R(a_1 \cdots a_j, u) = q_2 \tag{8}$$

$$R(a_1 \cdots a_n \$ a'_1 \cdots a'_{i-1}, v) = q_3 = R(a_1 \cdots a_n \$ a'_1 \cdots a'_{i-1}, u) = q_3 \tag{9}$$

$$R(a_1 \cdots a_n \$ a'_1 \cdots a'_{i-1} a'_j, v) = R(a_1 \cdots a_n \$ a'_1 \cdots a'_{i-1} a'_i, u) \tag{10}$$

$$R(a_1 \cdots a_n \$ a'_1 \cdots a'_{j-1}, v) = R(a_1 \cdots a_n \$ a'_1 \cdots a'_{j-1}, u) = q_3 \tag{11}$$

$$R(a_1 \cdots a_n \$ a'_1 \cdots a'_j \cdots a'_i, v) = R(a_1 \cdots a_n \$ a'_1 \cdots a'_i \cdots a'_j, u) \tag{12}$$

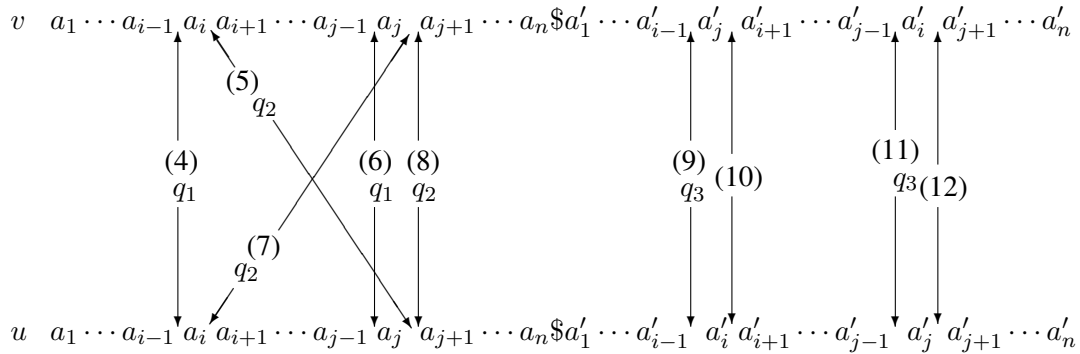


Figure 1. Equations (4)–(12) are illustrated by the corresponding arrows. The states adjacent to each arrow indicates the value of the function R .

Proof of (4): With pebble 1 above $a_1 \cdots a_{i-1}$, pebble 2 alone cannot detect that a'_i and a'_j have switched places, because they differ from all a_1, \dots, a_{i-1} . Thus, pebble 1 leaves the subword $a_1 \cdots a_{i-1}$ and arrives at a_i in the same state q_1 on both u and v .

Proof of (5): Pebble 1 arrives at a_j on u in the state q_1 , which, by (4) and (2), is the state in which pebble 1 arrives at a_i on v .

In addition, since $a_1 \cdots a_n \in L_{\text{diff}}$, by (2), pebble 2 arrives at $\$$ on u (where pebble 1 reads a_j) in the state q_4 – the same state in which pebble 2 arrives at $\$$ on v (where pebble 1 reads a_i). Therefore, since in v a'_i and a'_j have switched places, after leaving $\$$, the behavior of pebble 2 on the pattern $a'_1 \cdots a'_j \cdots a'_i \cdots a'_n$ of v (where pebble 1 reads a_i) is the same as that of pebble 2 on the pattern $a'_1 \cdots a'_i \cdots a'_j \cdots a'_n$ of u (where pebble 1 reads a_j). Thus, pebble 1 leaves a_j on u in the same state in which pebble 1 leaves a_i on v , namely, in the state q_2 , see (2).

Proof of (6): The proof is similar to that of (4). By (5) and (2),

$$R(a_1 \cdots a_i, v) = R(a_1 \cdots a_j, u) = q_2 = R(a_1 \cdots a_i, u).$$

With pebble 1 above $a_{i+1} \cdots a_{j-1}$, pebble 2 alone cannot detect that a'_i and a'_j have switched places, because they differ from all a_{i+1}, \dots, a_{j-1} . Thus, pebble 1 arrives at a_j in the same state q_1 on both u and v .

Proof of (7): The proof is similar to that of (5). Pebble 1 arrives at a_i on u in the state q_1 , in which, by (2) and (6), pebble 1 arrives at a_j on v .

In addition, since $a_1 \cdots a_n \in L_{\text{diff}}$, by (2), pebble 2 arrives at $\$$ on u (where pebble 1 reads a_i) in the state q_4 – the same state in which pebble 2 arrives at $\$$ on v (where pebble 1 reads a_j). Therefore, since in v a'_i and a'_j have switched places, after leaving $\$$, the behavior of pebble 2 on the pattern $a'_1 \cdots a'_j \cdots a'_i \cdots a'_n$ of v (where pebble 1 reads a_i) is the same as that of pebble 2 on the pattern $a'_1 \cdots a'_i \cdots a'_j \cdots a'_n$ of u (where pebble 1 reads a_j). Thus, pebble 1 leaves a_i on u in the same state in which pebble 1 leaves a_j on v , namely, in the state q_2 .

Proof of (8): As we have seen in the proofs of (5) and (7), pebble 1 leaves a_j on both u and v in the state q_2 .

Proof of (9): The proof is similar to that of (4). By (8), pebble 1 leaves a_j on both u and v in the same state. With pebble 1 above $a_{j+1} \cdots a_n \$ a'_1 \cdots a'_{i-1}$, pebble 2 alone cannot detect that a'_i and a'_j have switched places, because they differ from all $a_{j+1}, \dots, a_n, \$, a'_1, \dots, a'_{i-1}$. Thus, pebble 1 arrives at a'_i on u in the state q_3 – the same state in which pebble 1 arrives at a'_j on v .

Proof of (10): By (9), pebble 1 arrives at a'_i on u in the same state in which pebble 1 arrives at a'_j on v , namely, in the state q_3 , see (2).

Thus, by (2), pebble 2 arrives at $\$$ on u (where pebble 1 reads a'_i) in the state q_5 – the same state in which pebble 2 arrives at $\$$ on v (where pebble 1 reads a'_j). Since pebble 1 is in the same position on both u and v and a'_1, \dots, a'_n are pairwise different, pebble 2 leaves a'_n on u in the same state in which pebble 2 leaves a'_n on v .

Proof of (11): By (10), pebble 1 leaves a'_i on u in the same state in which pebble 1 leaves a'_j on v .

With pebble 1 above $a'_{i+1} \cdots a'_{j-1}$, pebble 2 alone cannot detect that a'_i and a'_j have switched places, because they differ from all $a'_{i+1}, \dots, a'_{j-1}$. Thus, pebble 1 leaves a'_i and arrives at a'_j on u in the same state in which pebble 1 leaves a'_j and arrives at a'_i on v , namely, in the state q_3 .

Proof of (12): The proof is similar to that of (10). By (11), pebble 1 arrives at a'_j on u in the same state in which pebble 1 arrives at a'_i on v , namely, in the state q_3 , see (2).

Thus, by (2), pebble 2 arrives at $\$$ on u (where pebble 1 reads a'_i) in the state q_5 – the same state in which pebble 2 arrives at $\$$ on v (where pebble 1 reads a'_j). Since pebble 1 is in the same position on both u and v and a'_1, \dots, a'_n are pairwise different, pebble 2 leaves a'_n on u in the same state in which pebble 2 leaves a'_n on v .

Now we are ready for the proof of (3). By (12), pebble 1 leaves a'_j on u in the same state in which pebble 1 leaves a'_i on v . With pebble 1 above $a'_{j+1} \cdots a'_n$, pebble 2 alone cannot detect that a'_i and a'_j have switched places, because they differ from all a'_{j+1}, \dots, a'_n . Thus, \mathcal{A} finishes the computation on both u and v in the same state. \square

References

- [1] C. David. Mots et données infinies. Master's thesis, Université Paris 7, LIAFA, 2004.
- [2] M. Davis. Hilbert's tenth problem is unsolvable. *The American Mathematical Monthly*, 80:233–269, 1973.
- [3] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Reading, MA, 1979.
- [4] Ju.V. Matijasevič. Enumerable sets are diophantine. *Soviet Mathematics. Doklady*, 11:354–358, 1970.
- [5] Y. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, Cambridge, MA, 1993.
- [6] F. Neven. Automata, logic, and XML. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26, Berlin, 2002. Springer.
- [7] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5:403–435, 2004.
- [8] E.L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.
- [9] T. Tan. Graph reachability and pebble automata over infinite alphabets. In *Proceedings of the 24th IEEE Symposium on Logic in Computer Science (LICS 2009)*, Los Alamitos, CA, 2009. IEEE Computer Society. To appear.