A note on two-pebble automata over infinite alphabets

Michael Kaminski  Tony Tan

Department of Computer Science
Technion – Israel Institute of Technology
Haifa 32000
Israel

Abstract

It is shown that the emptiness problem for 2-pebble automata languages is undecidable and that 2-pebble automata are weaker than 3-pebble automata.

1 Introduction

Pebble automata (PA) 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.\footnote{It should be noted that reductions of Hilbert’s tenth problem for proving undecidability are very rare in Computer Science.} We also present an example of a 3-PA language that is not accepted by 2-PA. To the best of our knowledge, this is the first separation example concerning PA.

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: $\Sigma$ is a fixed infinite alphabet not containing the left-end
marker $<$ or the right-end marker $>$. The input word to the automaton is of
the form $\langle w \rangle$, where $w \in \Sigma^*$. We also assume that $\Sigma$ contains the symbol $\$$
that will be used as a delimiter.

Symbols of $\Sigma$ are denoted by low case letters $a$, $b$, $c$, etc., possibly indexed,
and words over $\Sigma$ are denoted by low case letters $u$, $v$, $w$, etc., possibly
indexed.

Definition 1 A non-deterministic two-way $k$-PA over $\Sigma$ is a tuple $A =
\langle Q, q_0, F, \mu \rangle$ 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

- $\mu$ is a finite set of transitions of the form $\alpha \rightarrow \beta$ such that
  - $\alpha$ is of the form $(i, a, P, V, q)$ or $(i, P, V, q)$, where $i \in \{1, \ldots, k\}$,
    $a \in \Sigma \cup \{\langle, \rangle\}$, $P, V \subseteq \{1, \ldots, i-1\}$, and
  - $\beta$ is of the form $(q, \text{action})$, where $q \in Q$ and

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

Given a word $w = a_1 \cdots a_n \in \Sigma^*$, a configuration of $A$ on $\langle w \rangle$ is a triple
$[i, q, \theta]$, where $i \in \{1, \ldots, k\}$, $q \in Q$ and $\theta : \{1, \ldots, i\} \rightarrow \{0, 1, \ldots, n, n+1\}$.

The function $\theta$ defines the position of the pebbles and is called pebble assign-
ment. A configuration $[i, q, \theta]$ with $q \in F$ is called an accepting configuration.

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

$$ (1) \quad i = j \text{ and } p = q, $$

\footnote{The symbols at the 0 and $n+1$ positions are $<$ and $>$, respectively, and we assume that $A$ “knows” these symbols.}
\[(2) \ V = \{l < i : a_{\theta(l)} = a_{\theta(i)}\},\]
\[(3) \ P = \{l < i : \theta(l) = \theta(i)\}, \text{ and}\]
\[(4) \ a_{\theta(i)} = a.\]

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 \(\mu\) 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 \(\theta'(i+1) = 0.\)

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

The automaton \(A\) 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** ([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.

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

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

\(^3\)In 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.
is accepted by 3-PA. In contrast, this language is not accepted by 2-PA, see Proposition 12 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 4 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 4 below.

**Theorem 4** The emptiness problem for 2-PA languages is undecidable.

For the proof of Theorem 4 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 4 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 $L_{\text{ord}}$ consisting of all words in which every symbol from $\Sigma$ occurs at most one time:

$$L_{\text{ord}} = \{a_1a_2 \cdots a_n$\$a_1a_2 \cdots a_n : n \geq 1, \quad a_i \neq $, for each $i = 1, 2, \ldots, n, \text{ and } a_i \neq a_j, \text{ whenever } i \neq j\}$$

The words of $L_{\text{ord}}$ will be used for representation of positive integers in “unary” notation: a word $w \in L_{\text{ord}}$ represents the integer $|w|$.\footnote{Cf. [1, Section 7], where a similar representation was used for the proof of undecidability of the emptiness problem languages accepted by a kind of a register automata called po-2-DFA$_1$.} 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 5 The language $L_{\text{diff}}$ 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 6 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 6 Let

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

This language is accepted by 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.\(^5\) Verifying that both $u$ and $v$ are in $L_{\text{diff}}$ can be done in two swaps, see Example 5.\(^6\)

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

Example 7 Let $L_{\text{eq}}$ 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 5, verifies that both $u$ and $v$ are in $L_{\text{diff}}$, and then, like in Example 6, 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.

\(^5\)Naturally, 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.

\(^6\)Recall that we are dealing with two-way automata.
**Example 8** Let $L_{sq}$ consist of all words of the form

$$u{s_1}a_1v_1{s_2}a_2v_2\cdots{s_n}a_nv_n,$$

where

- $u \in L_{diff}$, and
- $u \sim a_1 \cdots a_n \sim v_1 \sim \cdots \sim v_n$.

By Example 6, $L_{sq}$ is accepted by 2-PA, because the membership test involves only verifying permutations of words. Obviously,

$$\{|w| : w \in L_{sq}\} = \{n^2 + 3n - 1 : n = 1, 2, \ldots\}$$

is not semi-linear.

The reduction of Hilbert’s tenth problem is based on Examples 9 and 10 below which illustrate the core idea lying behind the proof. Namely, these examples show that 2-PA can test atomic integer equalities.

**Example 9** Let $L_{add}$ be the language consisting of all words of the form

$$u{s_1}v{a_1}{c_1}{a_2}{c_2}\cdots{a_m}{c_m}{b_1}{c_{m+1}}\cdots{b_n}{c_{m+n}}w,$$

where

1. $u, v, w \in L_{diff}$,
2. $a_1 \cdots a_m \sim u$,
3. $b_1 \cdots b_n \sim v$, and
4. $c_1 \cdots c_{m+n} \sim w$.

Obviously,

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

The language $L_{add}$ is accepted by a 2-PA similar to that described in Example 6.

**Example 10** Let $L_{mult}$ be the language consisting of all words of the form

$$u{s_1}v{s_1}a_1c_{1,1}b_{1,1}c_{1,n}b_{1,n}s_2a_{m,c_{1,1}}b_{m,1}c_{m,n}b_{m,n}s_3w,$$

where

6
M1. \( u, v \) and \( w \) are in \( L_{\text{diff}} \),

M2. \( a_1 \cdots a_m \sim u \),

M3. \( b_{i,1} \cdots b_{i,n} \sim v \), for each \( i = 1, \ldots, m \), and

M4. \( c_{1,1} \cdots c_{1,n} \cdots c_{m,1} \cdots c_{m,n} \sim w \).

Obviously,

\[ |u| \times |v| = |w|. \]

Like in Example 5 it can be shown that two pebbles are sufficient to verify condition M1; and like in Example 6 it can be shown that two pebbles are sufficient to verify conditions M2–M4. Thus, \( L_{\text{mult}} \) is accepted by 2-PA.

Example 11 below is a straightforward extension of Example 9.

Example 11 Let \( m \) be a positive integer and let \( L_{\text{add},m} \) be the language consisting of all words of the form

\[ v_1 a_1 b_1 \cdots a_{m-1} b_{m-1} a_m b_m \] \( v_1 a_1 b_1 \cdots a_{m-1} b_{m-1} a_m b_m v_1 \]

where

A1. \( v_1, \ldots, v_m, v \in L_{\text{diff}} \),

A2. \( a_{i,1} \cdots a_{i,|v_i|} \sim v_i, i = 1, \ldots, m \), and

A3. \( b_1 \cdots b_{|v_1|+\cdots+|v_m|} \sim v \).

Obviously,

\[ |v_1| + \cdots + |v_m| = |v|. \]

The language \( L_{\text{add},m} \) is accepted by a 2-PA similar to that described in Example 6.

At last, we have arrived at the proof of Theorem 4. The intuition lying behind the proof is as follows. Examples 7, 9, and 10 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, respectively. For the proof of Theorem 4 we (quite naturally) extend these examples to testing results of polynomial evaluation, or, more precisely, to languages corresponding to polynomial evaluation. Loosely speaking, the
language $L_f$ corresponding to a polynomial $f(x_1, \ldots, x_m)$ with positive integer coefficients consist of all words of the form

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

where $v_1, \ldots, v_m, w \in L_{\text{diff}}$ and $|w| = f(|v_1|, \ldots, |v_m|)$. Since only permutations of words are needed to simulate the evaluation of $f(|v_1|, \ldots, |v_m|)$, $L_f$ is accepted by 2-PA, see the proof of Theorem 4 below.

**Proof of Theorem 4** We start with recalling Hilbert’s tenth problem that can equivalently be restated as follows. *Given two polynomials $f'(x_1, \ldots, x_m)$ and $f''(x_1, \ldots, x_m)$ with positive integer coefficients, do there exist positive integers $n_1, \ldots, n_m$ such that

$$f'(n_1, \ldots, n_m) = f''(n_1, \ldots, n_m)? \quad (1)$$

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

First, using Example 10, we show that 2-PA can recognize the values of monomials over positive integers. For this, with each sequence of monomials $M_1(x_1, \ldots, x_m), \ldots, M_k(x_1, \ldots, x_m)$ over positive integers we associate the language $L_{M_1, \ldots, M_k}$ defined by the following recursion.

- If $M_1(x_1, \ldots, x_m)$ is a constant $n$, then $L_{M_1}$ consists of all words of the form

  $$v_1 \cdot \cdot \cdot v_m \cdot \cdot \cdot v,$$

  where $v_1, \ldots, v_m, v \in L_{\text{diff}}$ and $|v| = n$, i.e., for all $v_1, \ldots, v_m \in L_{\text{diff}}, M_1(|v_1|, \ldots, |v_m|) = |v| (= n)$.

- If $M_1(x_1, \ldots, x_m)$ is of the form $x_j M(x_1, \ldots, x_m)$, then $L_{M_1}$ consists of all words of the form

  $$v_1 \cdot \cdot \cdot v_m \cdot \cdot \cdot v' a_1 c_{1,1} b_{1,1} \cdot \cdot \cdot c_{1,n} b_{1,n} \cdot \cdot \cdot a_m c_{m,1} b_{m,1} \cdot \cdot \cdot c_{m,n} b_{m,n} \cdot \cdot \cdot v,$$

  where

  - $v_1 \cdot \cdot \cdot v_m \cdot \cdot \cdot v' \in L_M$, i.e., $|v'| = M(|v_1|, \ldots, |v_m|)$;
  - $a_1 \cdots a_m \sim v_j$;
  - $b_{i,1} \cdots b_{i,n} \sim v'$, for each $i = 1, \ldots, m$; and
\[ c_{1,1} \cdots c_{1,n} \cdots c_{m,1} \cdots c_{m,n} \sim v. \]

That is,
\[ |v| = |v_j||v'| = |v_j|M(|v_1|, \ldots, |v_m|) = M_1(|v_1|, \ldots, |v_m|). \]

Assume that the language \( L_{M_1, \ldots, M_k} \) has been defined, and let \( M_{k+1} \) be a constant \( n \). Then \( L_{M_1, \ldots, M_{k+1}} \) consists of all words of the form \( w \$ v \), where \( w \in L_{M_1, \ldots, M_k} \), \( v \in L_{\text{diff}} \) and \( |v| = n \).

If \( M_{k+1}(x_1, \ldots, x_m) \) is of the form \( x_j M(x_1, \ldots, x_m) \), then \( L_{M_1, \ldots, M_{k+1}} \) is defined similarly to the second clause of the definition of \( L_{M_1} \).

Now, let \( f'(x_1, \ldots, x_m) = M'_1(x_1, \ldots, x_m) + \cdots + M'_{k'}(x_1, \ldots, x_m) \)
and let
\[ f''(x_1, \ldots, x_m) = M''_{1}(x_1, \ldots, x_m) + \cdots + M''_{k''}(x_1, \ldots, x_m), \]
where \( M'_i(x_1, \ldots, x_m), i' = 1, \ldots, k' \), and \( M''_i(x_1, \ldots, x_m), i'' = 1, \ldots, k'' \), are monomials. Then, like in Example 11, we can “extend” \( L_{M'_1, \ldots, M'_{k'}, M''_1, \ldots, M''_{k''}} \) to the language \( L_{f', f''} \) that for each \( m \)-tuple \( v_1, \ldots, v_m \in L_{\text{diff}} \) contains a word of the form
\[ v_1 \$ \cdots \$ v_m \$ \cdots \$ w' \$ w'', \]
where \( |w'| = f'(|v_1|, \ldots, |v_m|) \) and \( |w''| = f''(|v_1|, \ldots, |v_m|) \).\(^7\)

Finally, let \( L_{f' = f''} \) be the languages consisting of all words of the form
\[ v_1 \$ \cdots \$ v_m \$ \cdots \$ w' \$ w'' \$ a'_{1} \cdots a'_{n} \$ a''_{1} \cdots a''_{n}, \]
where
- \( v_1 \$ \cdots \$ v_m \$ \cdots \$ w' \$ w'' \in L_{f', f''} \),
- \( a'_{1} \cdots a'_{n} \sim w' \), and
- \( a''_{1} \cdots a''_{n} \sim w'' \).

\(^7\)Note that delimited patterns of \( L_{M'_1, \ldots, M'_{k'}, M''_1, \ldots, M''_{k''}} \) can be detected by using just the finite memory (states) of an appropriate 2-PA.
Then, like in Example 7, 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 4) is complete. \hfill \Box

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

**Proposition 12** The language $L_{\text{ord}}$ is not accepted by 2-PA.

**Proof** Assume to the contrary that $L_{\text{ord}}$ is accepted by an $s$-state 2-PA $\mathcal{A}$. By Theorem 2, 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

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

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 $i = 1, \ldots, n$, we associate a quintuple of states $\langle q^i_1, q^i_2, q^i_3, q^i_4, q^i_5 \rangle$ that is defined as follows.

- $q^i_1$ is the state in which pebble 1 arrives at $a_i$.
- $q^i_2$ is the state in which pebble 1 leaves $a'_i$.
- $q^i_3$ is the state in which pebble 1 arrives at $a'_i$.
- $q^i_4$ is the state in which pebble 1 is above $a_i$ and pebble 2 arrives at $\$. 
- $q^i_5$ is the state in which pebble 1 is above $a'_i$ and pebble 2 arrives at $\$. 

\textsuperscript{8}We use primes to differentiate between occurrences of a symbol before and after $\$. 

\hfill 10
Since \( n > s^5 \), there exist states \( q_k, k = 1, \ldots, 5 \), and two indexes \( i \) and \( j \), \( i < j \), such that

\[
\langle q_1^i, q_2^i, q_3^i, q_4^i, q_5^i \rangle = \langle q_1^j, q_2^j, q_3^j, q_4^j, q_5^j \rangle = \langle q_1, q_2, q_3, q_4, q_5 \rangle.
\]  

(2)

We contend that the word

\[ v = a_1 \cdots a_i \cdots a_j \cdots a_n \$ a_1' \cdots a_j' \cdots a_n', \]

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,

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

We shall prove that

\[ R(v, v) = R(u, u), \]  

(3)

which would imply \( v \in L(\mathcal{A}) \), in contradiction with \( v \not\in L_{\text{ord}} \).

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.

\[
\begin{align*}
R(a_1 \cdots a_{i-1}, v) &= R(a_1 \cdots a_{i-1}, u) = q_1 \quad (4) \\
R(a_1 \cdots a_i, v) &= R(a_1 \cdots a_j, u) = q_2 \quad (5) \\
R(a_1 \cdots a_{j-1}, v) &= R(a_1 \cdots a_{j-1}, u) = q_1 \quad (6) \\
R(a_1 \cdots a_j, v) &= R(a_1 \cdots a_i, u) = q_2 \quad (7) \\
R(a_1 \cdots a_j, v) &= R(a_1 \cdots a_j, u) = q_2 \quad (8) \\
R(a_1 \cdots a_n \$ a_1' \cdots a_{i-1}', v) &= q_3 = R(a_1 \cdots a_n \$ a_1' \cdots a_{i-1}' u) = q_3 \quad (9) \\
R(a_1 \cdots a_n \$ a_1' \cdots a_{i-1}' a_j', v) &= R(a_1 \cdots a_n \$ a_1' \cdots a_{i-1}' a_j', u) \quad (10) \\
R(a_1 \cdots a_n \$ a_1' \cdots a_{j-1}' v) &= R(a_1 \cdots a_n \$ a_1' \cdots a_{j-1}' u) = q_3 \quad (11) \\
R(a_1 \cdots a_n \$ a_1' \cdots a_j' \cdots a_i', v) &= R(a_1 \cdots a_n \$ a_1' \cdots a_i' \cdots a_j', u) \quad (12)
\end{align*}
\]
Figure 1: Equations (4)–(12) are illustrated by the corresponding arrows. The states adjacent to each arrow indicate 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 $a_1, \ldots, a_{i-1}$. 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 $a'_i \cdots a'_j \cdots a'_n$ of $v$ (where pebble 1 reads $a_i$) is the same as that of pebble 2 on the pattern $a'_i \cdots a'_j \cdots a'_n$ 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 $a_{i+1}, \ldots, a_{j-1}$. 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 $ (where pebble 1 reads $a_j$). Therefore, since in $a_i'$ and $a_j'$ have switched places, after leaving $\$, the behavior of pebble 2 on the pattern $a'_1 \cdots a'_j \cdots a'_{i-1} \cdots a'_n$ of $v$ (where pebble 1 reads $a_i$) is the same as that of pebble 2 on the pattern $a'_1 \cdots a'_j \cdots a'_{j-1} \cdots a'_n$ 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 $a'_j$ and $a'_j$ have switched places, because they differ from all $a_{j+1}, \ldots, a_n$, $a'_1, \ldots, a'_{i-1}$. Thus, pebble 1 arrives at $a'_i$ on $u$ in the state $q_3$ in 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 $q_5$ in 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 $a'_1, \ldots, a'_n$ 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 $a'_{i+1}, \ldots, a'_{j-1}$. 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 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 $a'_1, \ldots, a'_n$ are pairwise different, pebble 2 leaves $a'_n$ on $u$ in the same state in which pebble 2 leaves $a'_n$ on $v$. 


13
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 $a'_j$ and $a'_i$ have switched places, because they differ from all $a'_{j+1}, \ldots, a'_n$. Thus, $\mathcal{A}$ finishes the computation on both $u$ and $v$ in the same state. \hfill\Box

Acknowledgment

This research was supported by the S. and N. Grand research fund.

References


