Journal of Computer and System Sciences 76 (2010) 778–791

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

On pebble automata for data languages with decidable emptiness problem ☆,☆☆

Tony Tan

School of Informatics, University of Edinburgh, United Kingdom

ARTICLE INFO

Article history:
Received 6 November 2009
Received in revised form 16 February 2010
Available online 17 March 2010

Keywords:
Finite state automata
Infinite alphabet
Decidability

ABSTRACT

In this paper we study a subclass of pebble automata (PA) for data languages for which the emptiness problem is decidable. Namely, we show that the emptiness problem for weak 2-pebble automata is decidable, while the same problem for weak 3-pebble automata is undecidable. We also introduce the so-called *top view* weak PA. Roughly speaking, top view weak PA are weak PA where the equality test is performed only between the data values seen by the two most recently placed pebbles. The emptiness problem for this model is still decidable. It is also robust: alternating, non-deterministic and deterministic top view weak PA have the same recognition power; and are strong enough to accept all data languages expressible in Linear Temporal Logic with the future-time operators, augmented with one register freeze quantifier.

1. Introduction

Regular languages are clearly one of the most important concepts in computer science. They have applications in basically all branches of computer science. It can be argued that the following properties contributed to their success.

1. Expressiveness: In many settings regular languages are powerful enough to capture the kinds of patterns that have to be described.
2. Decidability: Unlike many general computational models, the mechanisms associated with regular languages allow one to perform automated semantic analysis.
3. Efficiency: The model checking problem, that is, testing whether a given string is accepted by a given automaton can be solved in polynomial time.
4. Closure properties: The class of regular languages possesses all important closure properties.
5. Robustness: The class of regular languages has many characterizations, which include finite state automata, regular expressions, monoids and monadic second-order logic.

Moreover, similar notions of regularity have been successfully generalized to other kinds of structures, including infinite strings and finite, as well as infinite, ranked or unranked, trees. Most recent applications of regular languages (on infinite strings and finite, unranked trees, respectively) are in model checking and XML processing.[1]

☆ This work was done while the author was in the Department of Computer Science in Technion – Israel Institute of Technology.

☆☆ Its extended abstract version has also been published in Tan (2009) [17].

E-mail address: ttan@inf.ed.ac.uk.

[1] Much of the materials in Section 1 are taken from [3].

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2010.03.004

- In model checking a system is a finite state one and properties are specified in a logic like LTL. Satisfiability of a formula in a system is checked on the structure that is the product of the system automaton and an automaton corresponding to the formula. The step from the "real" system to its finite state representation usually involves many abstraction, especially with respect to data values (variables, process numbers, etc.). Often their range is restricted to a finite domain. Even though this approach has been successful and found its way into large scale industrial applications, the finite abstraction has some inherent shortcomings. As an example, n identical processes with m states each give rise to an overall model of size m^n. If the number of processes is unbounded or unknown in advance, the finite state approach fails. Previous work has shown that even in such setting decidability can be obtained by restricting the problem in various ways [1,7].
- In XML document processing, regular concepts occur in various contexts. For example, the structural specification of XML documents are usually given as regular languages, like DTD or XML schema [12]. However, such specifications usually ignore the attributes and data values. From a database point of view, this is not completely satisfactory, because a *schema* should allow one to describe not only the structure of the data, but also to define restrictions on the data values via integrity constraints such as key or inclusion constraints. There exists a work addressing this problem [2], but like in the case of model checking, the methods rely heavily on a case-to-case analysis.

So, in the above settings, the finite state abstraction leads to interesting results, but does not address all problems arising in applications. In both cases, it would already be a big advance, if each position, in either a string or a tree, could carry a *data value*, in addition to its label.

This paper is part of a broader research program which aims at studying such extensions in a systematic way. As any kind of operations on the infinite domain quickly leads to undecidability of basic processing tasks (even a linear order on the domain is harmful), we concentrate on the setting, where data values can only be tested for equality. Furthermore, in this paper we only consider *finite data strings*, that is, finite strings, where each position carries a label from a finite alphabet and a data value from an infinite domain. A *data language* is a language consisting of finite data strings. Recently, there has been a significant amount of work in this direction. See, for example, [3,4,6,9,13,15].

Roughly speaking, there are two approaches to studying data languages: logic and automata. Below is a brief survey on both approaches. For a more comprehensive survey, we refer the reader to [15]. The study of data languages, which can also be viewed as languages over infinite alphabets, starts with the introduction of finite-memory automata (FMA) in [9], which are also known as *register automata* (RA). The study of FMA was continued and extended in [13], in which *pebble automata* (PA) were also introduced. Each of these models has its own advantages and disadvantages. Languages accepted by FMA are closed under standard language operations: intersection, union, concatenation, and Kleene star. In addition, from the computational point of view, FMA are a much easier model to handle. Their emptiness problem is decidable, whereas the same problem for PA is not. However, the PA languages possess a very nice logical property: closure under *all* boolean operations, whereas FMA languages are not closed under complementation.

Later in [4] first-order logic for data languages was considered, and, in particular, the so-called *data automata* was introduced. It was shown that data automata define the fragment of existential monadic second-order logic for data languages in which the first-order part is restricted to two variables only. An important feature of data automata is that their emptiness problem is decidable, even for the infinite words, but is at least as hard as reachability for Petri nets. The automata themselves always work nondeterministically and seemingly cannot be determinized, see [3]. It was also shown that the satisfiability problem for the three-variable first-order logic is undecidable.

Another logical approach is via the so-called *linear temporal logic with n register freeze quantifier*, denoted $\mathrm{LTL}_n^{\downarrow}(\mathtt{X},\mathtt{U})$, see [6]. It was shown that one-way alternating n register automata accept all $\mathrm{LTL}_n^{\downarrow}(\mathtt{X},\mathtt{U})$ languages and the emptiness problem for one-way alternating one register automata is decidable. Hence, the satisfiability problem for $\mathrm{LTL}_1^{\downarrow}(\mathtt{X},\mathtt{U})$ is decidable, as well. Adding one more register or past time operators to $\mathrm{LTL}_1^{\downarrow}(\mathtt{X},\mathtt{U})$ makes the satisfiability problem undecidable.

In this paper we continue the study of PA, which are finite state automata with a finite number of pebbles. The pebbles are placed on/lifted from the input word in the stack discipline – first in last out – and are intended to mark positions in the input word. One pebble can only mark one position and the most recently placed pebble serves as the head of the automaton. The automaton moves from one state to another depending on the current label and the equality tests among data values in the positions currently marked by the pebbles, as well as, the equality tests among the positions of the pebbles.

Furthermore, as defined in [13], there are two types of PA, according to the position of the new pebble placed. In the first type, the ordinary PA, also called *strong* PA, the new pebbles are placed at the beginning of the string. In the second type, called *weak* PA, the new pebbles are placed at the position of the most recent pebble. Obviously, two-way weak PA is just as expressive as two-way ordinary PA. However, it is known that one-way non-deterministic weak PA are weaker than one-way ordinary PA, see [13, Theorem 4.5].

We show that the emptiness problem for one-way weak 2-pebble automata is decidable, while the same problem for one-way weak 3-pebble automata is undecidable. We also introduce the so-called *top view* weak PA. Roughly speaking, top view weak PA are one-way weak PA where the equality test is performed only between the data values seen by the two most recently placed pebbles. Top view weak PA are quite robust: alternating, non-deterministic and deterministic top view weak PA have the same recognition power. To the best of our knowledge, this is the first model of computation for

data language with such robustness. It is also shown that top view weak PA can be simulated by one-way alternating one-register RA. Therefore, their emptiness problem is decidable. Another interesting feature is that top view weak PA can simulate all $\mathrm{LTL}_1^{\downarrow}(\mathtt{X}, \mathtt{U})$ languages, and the number of pebbles needed to simulate such LTL sentences corresponds linearly to the so-called *free quantifier rank* of the sentences, the depth of the nesting level of the freeze operators in the sentence.

This paper is organized as follows. In Section 2 we review the models of computations for data languages considered in this paper. We present the proof of the equivalence between alternating and deterministic weak 2-PA in Section 3. Section 4 and Section 5 deals with the decidability and the complexity issues of weak PA, respectively. In Section 6 we introduce top view weak PA. We also introduce a simple extension to top view weak PA, called unbounded top view weak PA, in which the number of pebbles is unbounded in Section 7. Finally, we end our paper with a brief remark in Section 8.

2. Models of computation

In this section we recall the definitions of weak PA from [13] and of register automata (RA) from [6,9]. We will use the following notation. We always denote by Σ a finite alphabet of *labels* and by $\mathfrak{D}$ an infinite set of *data values*. A *Σ-data word* $w = \binom{\sigma_1}{a_1}\binom{\sigma_2}{a_2}\cdots\binom{\sigma_n}{a_n}$ is a finite sequence over $\Sigma \times \mathfrak{D}$, where $\sigma_i \in \Sigma$ and $a_i \in \mathfrak{D}$. A *Σ-data language* is a set of Σ-data words.

We will also use the following notations.

$$\mathsf{Proj}_{\Sigma}(w) = \sigma_1 \cdots \sigma_n$$

$$\mathsf{Proj}_{\mathfrak{D}}(w) = a_1 \cdots a_n$$

$$\mathsf{Cont}_{\Sigma}(w) = \{\sigma_1, \ldots, \sigma_n\}$$

$$\mathsf{Cont}_{\mathfrak{D}}(w) = \{a_1, \ldots, a_n\}$$

We assume that neither of Σ and $\mathfrak{D}$ contain 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 $\triangleleft$ and $\triangleright$ mark the left-end and the right-end of the input word. Finally, the symbols $\nu, \vartheta, \sigma, \ldots$, possibly indexed, denote labels in Σ and the symbols $a, b, c, d, \ldots$, possibly indexed, denote data values in $\mathfrak{D}$.

2.1. Pebble automata

Definition 1. (See [13, Definition 2.3].) A *one-way alternating weak k-pebble automaton* (k-PA) over Σ is a system $\mathcal{A} = \langle \Sigma, Q, q_0, F, \mu, U \rangle$ 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;
- $U \subseteq Q - F$ is the set of *universal* states; and
- $\mu \subseteq \mathcal{C} \times \mathcal{D}$ is the set of transitions, where
  - $\mathcal{C}$ is a set whose elements are of the form (i, σ, V, q), where $1 \leqslant i \leqslant k$, $\sigma \in \Sigma$, $V \subseteq \{i+1, \ldots, k\}$ and $q \in Q$; and
  - $\mathcal{D}$ is a set whose elements are of the form $(q, \mathtt{act})$, where $q \in Q$ and $\mathtt{act}$ is either `stay`, `right`, `place-pebble` or `lift-pebble`.

  Elements of μ will be written as $(i, \sigma, V, q) \to (p, \mathtt{act})$.

Given a word $w = \binom{\sigma_1}{a_1}\cdots\binom{\sigma_n}{a_n} \in (\Sigma \times \mathfrak{D})^*$, a *configuration of $\mathcal{A}$ on $\triangleleft w \triangleright$* is a triple $[i, q, \theta]$, where $i \in \{1, \ldots, k\}$, $q \in Q$, and $\theta : \{i, i+1, \ldots, k\} \to \{0, 1, \ldots, n, n+1\}$, where 0 and $n+1$ are positions of the end markers $\triangleleft$ and $\triangleright$, respectively. The function θ defines the position of the pebbles and is called the *pebble assignment*. The *initial* configuration is $\gamma_0 = [k, q_0, \theta_0]$, where $\theta_0(k) = 0$ is the *initial* pebble assignment. A configuration $[i, q, \theta]$ with $q \in F$ is called an *accepting* configuration.

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

(1) $i = j$ and $p = q$,
(2) $V = \{l > i\colon\ a_{\theta(l)} = a_{\theta(i)}\}$, and
(3) $\sigma_{\theta(i)} = \sigma$.

Note that in a configuration $[i, q, \theta]$, pebble i is in control, serving as the head pebble.

Next, we define the transition relation $\vdash_{\mathcal{A}}$ as follows: $[i, q, \theta] \vdash_{\mathcal{A}} [i', q', \theta']$, if there is a transition $\alpha \to (p, \mathtt{act}) \in \mu$ that applies to $[i, q, \theta]$ such that $q' = p$, $\theta'(j) = \theta(j)$, for all $j > i$, and

- if $\mathtt{act} = \mathtt{stay}$, then $i' = i$ and $\theta'(i) = \theta(i)$;
- if $\mathtt{act} = \mathtt{right}$, then $i' = i$ and $\theta'(i) = \theta(i) + 1$;
- if $\mathtt{act} = \mathtt{lift\text{-}pebble}$, then $i' = i + 1$;
- if $\mathtt{act} = \mathtt{place\text{-}pebble}$, then $i' = i - 1$, $\theta'(i-1) = \theta'(i) = \theta(i)$.

As usual, we denote the reflexive transitive closure of $\vdash_{\mathcal{A}}$ by $\vdash^*_{\mathcal{A}}$. When the automaton $\mathcal{A}$ is clear from the context, we will omit the subscript $\mathcal{A}$.

Remark 2. Note the pebble numbering that differs from that in [13]. In the above definition we adopt the pebble numbering from [5] in which the pebbles placed on the input word are numbered from k to i and not from 1 to i as in [13]. The reason for this reverse numbering is that it allows us to view the computation between placing and lifting pebble i as a computation of an $(i-1)$-pebble automaton.

Furthermore, the automaton is no longer equipped with the ability to compare positional equality, in contrast with the ordinary PA introduced in [13]. Such ability no longer makes any difference because of the "weak" manner in which the new pebbles are placed.

The acceptance criteria are based on the notion of *leads to acceptance* below. For every configuration $\gamma = [i, q, \theta]$,

- if $q \in F$, then γ leads to acceptance;
- if $q \in U$, then γ leads to acceptance if and only if for all configurations γ' such that $\gamma \vdash \gamma'$, γ' leads to acceptance;
- if $q \notin F \cup U$, then γ leads to acceptance if and only if there is at least one configuration γ' such that $\gamma \vdash \gamma'$ and γ' leads to acceptance.

A Σ-data word $w \in (\Sigma \times \mathfrak{D})^*$ is accepted by $\mathcal{A}$, if γ_0 leads to acceptance. The language $L(\mathcal{A})$ consists of all data words accepted by $\mathcal{A}$.

The automaton $\mathcal{A}$ is *non-deterministic*, if the set $U = \emptyset$, and it is *deterministic*, if there is exactly one transition that applies for each configuration. It turns out that weak PA languages are quite robust.

Theorem 3. *For all $k \geqslant 1$, alternating, non-deterministic and deterministic weak k-PA have the same recognition power.*

The proof is quite standard. We will sketch the proof for the case of $k = 2$ in the next section. The extension to the general case is straightforward, thus, omitted. In view of Theorem 3, we will always assume that our weak k-PA is deterministic.

We end this subsection with an example of a language accepted by weak 2-PA. This example will be useful in the subsequent section.

Example 4. Consider a Σ-data language $L_\sim$ defined as follows. A Σ-data word $w = \binom{\sigma_1}{a_1} \cdots \binom{\sigma_n}{a_n} \in L_\sim$ if and only if for all $i, j = 1, \ldots, n$, if $a_i = a_j$, then $\sigma_i = \sigma_j$. That is, $w \in L_\sim$ if and only if whenever two positions in w carry the same data value, their labels are the same.

The language $L_\sim$ is accepted by weak 2-PA which works in the following manner. Pebble 2 iterates through all possible positions in w. At each iteration, the automaton stores the label seen by pebble 2 in the state and places pebble 1. Then, pebble 1 scans through all the positions to the right of pebble 2, checking whether there is a position with the same data value as pebble 2. If there is such a position, then the label seen by pebble 1 must be the same as the label seen by pebble 2, which has previously been stored in the state. After that, pebble 1 is lifted, and the control returns to pebble 2 and the iteration continues.

2.2. Register automata

We are only going to sketch roughly the definition of register automata. Readers interested in its more formal treatment can consult [6,9]. In essence, k register automaton, or, shortly k-RA, is a finite state automaton equipped with a header to scan the input and k registers, numbered from 1 to k. Each register can store exactly one data value from $\mathfrak{D}$. The automaton is *two-way* if the header can move to the left or to the right. It is *alternating* if it is allowed to branch into a finite number of parallel computations.

More formally, a two-way alternating k-RA over the label Σ is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, u_0, \mu, F \rangle$ where

- Q_0, $q_0 \in Q$ and $F \subseteq Q$ are the finite state of states, the initial state and the set of final states, respectively.
- $u_0 = a_1 \cdots a_k$ is the initial content of the registers.
- μ is a set of transitions of the following form.
  - (i) $(q, \sigma) \to q'$ where $\sigma \in \{\lhd, \rhd\}$ and $q, q' \in Q$.
    That is, if the automaton $\mathcal{A}$ is in state q and the header is currently reading either of the symbols $\lhd, \rhd$, then the automaton can enter the state q'.
  - (ii) $(q, \sigma, V) \to q'$ where $\sigma \in \Sigma$, $V \subseteq \{1, \ldots, k\}$ and $q, q' \in Q$.
    That is, if the automaton $\mathcal{A}$ is in state q and the header is currently reading a position labeled with σ and V is the set of all registers containing the current data value, then the automaton can enter the state q'.
  - (iii) $q \to (q', I)$ where $I \subseteq \{1, \ldots, k\}$ and $q, q' \in Q$.
    That is, if the automaton $\mathcal{A}$ is in state q, then the automaton can enter the state q' and store the current data value into the registers whose indices belong to I.

(iv) $q \to (q_1 \wedge \cdots \wedge q_i)$ and $q \to (q_1 \vee \cdots \vee q_i)$ where $i \geqslant 1$ and $q, q' \in Q$.
That is, if the automaton $\mathcal{A}$ is in state q, then it can decide to perform conjunctive or disjunctive branching into the states $q_1, \ldots, q_i$.

(v) $q \to (q', \mathtt{act})$ where $\mathtt{act} \in \{\mathtt{left}, \mathtt{right}\}$ and $q, q' \in Q$.
That is, if the automaton $\mathcal{A}$ is in state q, then it can enter the state q' and move to the next or the previous word position.

A register automaton is called *non-deterministic* if the branchings of state (in item (iv)) are all disjunctive. It is called *one-way* if the header is not allowed to move to the previous word position.

A *configuration* $\gamma = [j, q, b_1 \cdots b_k]$ of the automaton $\mathcal{A}$ consists of the current position of the header in the input word j, the state of the automaton q and the content of the registers $b_1 \cdots b_k$. The configuration γ is called *accepting* if the state is a final state in F.

From each configuration γ, the automaton performs legitimate computation according to the transition relation and enters another configuration γ'. If the transition is branching, then it can split into several configurations $\gamma'_1, \ldots, \gamma'_m$.

Similarly, we can define the notion of *leads to acceptance* for a configuration γ as in previous subsection. An input word w is accepted by $\mathcal{A}$ if the initial configuration leads to acceptance. As usual, $L(\mathcal{A})$ denotes the language accepted by $\mathcal{A}$.

3. Equivalence between alternating and deterministic one-way weak 2-PA

For every one-way alternating weak 2-PA, we will construct its equivalent one-way deterministic weak 2-PA. This is done in two steps.

1. First, we transform the one-way alternating weak 2-PA into its equivalent one-way non-deterministic weak 2-PA.
2. Then, we transform the one-way non-deterministic weak 2-PA into its equivalent one-way deterministic weak 2-PA.

We present step 2 first.

3.1. From non-deterministic to deterministic

Let $\mathcal{A} = \langle Q, q_0, F, \mu \rangle$ be a non-deterministic weak 2-PA. We start by normalizing the behavior of $\mathcal{A}$ as follows.

N1. For every configuration γ of $\mathcal{A}$, there exists a transition in μ that applies to it.
N2. There is no `stay` transition in $\mathcal{A}$. On each transition the automaton $\mathcal{A}$ moves the head pebble to the right, lifts the current head pebble, or places a new pebble.
N3. The automaton can only enter a final state when the control is in pebble 2. Furthermore, it does so only after it reads the right-end marker $\triangleright$.
N4. Immediately after pebble 2 moves right, pebble 1 is placed.
N5. Pebble 1 is lifted only when it reaches the right-end marker $\triangleright$.

Such normalization can be done by adding some extra states to $\mathcal{A}$ (or, extra transitions in the case of N2). The normalization N5 is especially crucial, as it implies that non-determinism on pebble 1 is now limited only to deciding which state to enter. There is no non-determinism in choosing which action to take, i.e. either to lift pebble 1 or to keep on moving right.

Next, we note that immediately after pebble 1 is lifted, there can be two choices of actions for pebble 2:

- to place pebble 1 again; or
- to move pebble 2 to the right.

The following sixth normalization handles this situation:

N6. Immediately after pebble 1 is lifted, pebble 2 moves right.

In other words, while pebble 2 is reading a specific position, pebble 1 makes exactly one pass, from the position of pebble 2 to the right end of the input, instead of making several rounds of passes by placing pebble 1 again immediately after it is lifted.

Since there are only finitely many states, there can only be finitely many passes. So, the normalization N6 can be achieved by simultaneously simulating all possible passes in one pass.

With the normalizations N1–N6, there is no non-determinism in choosing which action to take for pebble 2. Similar to pebble 1, non-determinism of pebble 2 is now limited only in deciding which states to take. This is summed up in the following remark.

Remark 5. For each $i = 1, 2$, if $(i, P, V, p) \to (q_1, \mathtt{act}_1)$ and $(i, P, V, p) \to (q_2, \mathtt{act}_2)$, then $\mathtt{act}_1 = \mathtt{act}_2$.

Now that the non-deterministism is reduced to deciding which state to enter, the determinization of $\mathcal{A}$ becomes straightforward. Similar to the classical proof of the equivalence between non-deterministic and deterministic finite state automata, we can take the power set of the states of $\mathcal{A}$ to deterministically simulate $\mathcal{A}$.

Remark 6. We note that the normalization steps N1–N5 can be performed similarly for weak k-PA $\mathcal{A}$, for every $k = 1, 2, \ldots$. The determinization of non-deterministic weak k-PA can then be done in a similar manner.

3.2. From alternating to non-deterministic

Let $\mathcal{A} = \langle \Sigma, Q, q_0, \mu, F, U \rangle$ be a one-way alternating weak 2-PA. Adding some extra states, we can normalize $\mathcal{A}$ as follows.

A1. For every $p \in U$, if $(i, \sigma, V, p) \to (q, \mathtt{act}) \in \mu$, then $\mathtt{act} = \mathtt{stay}$.
A2. Every pebble can be lifted only after it reads the right-end marker $\triangleright$.
A3. Only pebble 2 can enter a final state and it does so only after it reads the right-end marker $\triangleright$.

We assume that Q is partitioned into $Q_1 \cup Q_2$ where Q_i is the set of states, where pebble i is the head pebble, for each $i = 1, 2$. We can further partition each Q_i into four sets of states: $Q_{i,\mathtt{stay}}$, $Q_{i,\mathtt{right}}$, $Q_{i,\mathtt{place}}$, $Q_{i,\mathtt{lift}}$ such that for every $i = 1, 2$, $\sigma \in \Sigma$, $V \subseteq \{1, 2\}$, and $q, p \in Q$,

A4. If $q \in Q_{i,\mathtt{stay}}$ and $(i, \sigma, V, q) \to (p, \mathtt{act}) \in \mu$, then $\mathtt{act} = \mathtt{stay}$.
A5. If $q \in Q_{i,\mathtt{right}}$ and $(i, \sigma, V, q) \to (p, \mathtt{act}) \in \mu$, then $\mathtt{act} = \mathtt{right}$.
A6. If $q \in Q_{i,\mathtt{place}}$ and $(i, \sigma, V, q) \to (p, \mathtt{act}) \in \mu$, then $\mathtt{act} = \mathtt{place\text{-}pebble}$.
A7. If $q \in Q_{i,\mathtt{lift}}$ and $(i, \sigma, V, q) \to (p, \mathtt{act}) \in \mu$, then $\mathtt{act} = \mathtt{lift\text{-}pebble}$.

The intuitive meaning of these states are clear. We partition the states of $\mathcal{A}$ according to all possible actions of $\mathcal{A}$. Especially, by restricting the set U of universal states to be inside $Q_{1,\mathtt{stay}} \cup Q_{2,\mathtt{stay}}$, the non-determinization process of $\mathcal{A}$ will be much easier.

The non-determinization process itself is a pretty straightforward simulation of all possible computation paths of $\mathcal{A}$. On an input $w = \binom{\sigma_1}{a_1} \cdots \binom{\sigma_n}{a_n}$, due to universal branching, the automaton $\mathcal{A}$ can be in several states when it reaches a certain position i, where $1 \leqslant i \leqslant n$. Since the number of states is finite, to simulate the run of $\mathcal{A}$ on w, it is then sufficient to remember all these states and simulates all possible transitions from these states.

Formally, we define the non-deterministic weak 2-PA $\mathcal{A}' = \langle \Sigma, Q', q_0', \mu', F' \rangle$, where

- $Q' = 2^{Q_2} \cup (2^{Q_2} \times 2^{Q_1})$;
- $q_0' = \{q_0\}$;
- $F' = 2^F - \{\emptyset\}$.

The set μ' contains the following transitions.

- The transitions, when pebble 2 is the head, are as follows. For every $S \subseteq Q_2$ and for every $\sigma \in \Sigma$, we have the following transitions.
  - If S contains a state $q \in U$, then
    $$(2, \sigma, \emptyset, S) \to \big(\big(S - \{q\}\big) \cup U_q, \mathtt{stay}\big) \in \mu'$$
    where $U_q = \{p \mid (2, \sigma, \emptyset, q) \to (p, \mathtt{stay}) \in \mu\}$.
  - If S contains a state $q \in Q_{2,\mathtt{stay}}$ and $S \cap U = \emptyset$, then
    $$(2, \sigma, V, S) \to \big(\big(S - \{q\}\big) \cup N_q, \mathtt{stay}\big) \in \mu'$$
    for every $N_q \subseteq \{p \mid (i, \sigma, \emptyset, q) \to (p, \mathtt{stay}) \in \mu\}$ and $N_q \neq \emptyset$.
  - If S contains a state $q \in Q_{2,\mathtt{place}}$ and $S \cap Q_{2,\mathtt{stay}} = \emptyset$, then
    $$(2, \sigma, \emptyset, S) \to \big(\big(S - \{q\}, \{p\}\big), \mathtt{place\text{-}pebble}\big) \in \mu'$$
    where $(2, \sigma, \emptyset, q) \to (p, \mathtt{place\text{-}pebble}) \in \mu$.
  - If $S \subseteq Q_{2,\mathtt{right}}$, then
    $$(2, \sigma, \emptyset, S) \to \big(S', \mathtt{right}\big) \in \mu'$$
    where $S' = \{p \mid (2, \sigma, \emptyset, q) \to (p, \mathtt{right}) \in \mu \text{ and } q \in S\}$.

- The transitions, when pebble 1 is the head, are as follows. For every $S_2 \subseteq Q_2$, $S_1 \subseteq Q_1$, $V \subseteq \{2\}$ and for every $\sigma \in \Sigma$, we have the following transitions.
  - If S_1 contains a state $q \in U$, then
  $$\big(1, \sigma, V, (S_2, S_1)\big) \to \big(\big(S_2, \big(S_1 - \{q\}\big) \cup U_q\big), \texttt{stay}\big) \in \mu'$$
  where $U_q = \{p \mid (1, \sigma, V, q) \to (p, \texttt{stay}) \in \mu\}$.
  - If S_1 contains a state $q \in Q_{1,\texttt{stay}}$ and $S \cap U = \emptyset$, then
  $$\big(1, \sigma, V, (S_2, S_1)\big) \to \big(\big(S_1, \big(S_1 - \{q\}\big) \cup N_q\big), \texttt{stay}\big) \in \mu'$$
  for every $N_q \subseteq \{p \mid (1, \sigma, V, q) \to (p, \texttt{stay}) \in \mu\}$ and $N_q \neq \emptyset$.
  - If $S_1 \subseteq Q_{1,\texttt{right}}$, then
  $$\big(1, \sigma, V, (S_1, S_1)\big) \to \big(\big(S_1, S_1'\big), \texttt{right}\big) \in \mu'$$
  where $S_1' = \{p \mid (1, \sigma, V, q) \to (p, \texttt{right}) \in \mu \text{ and } q \in S \cap Q_1\}$.
  - If $S_1 \subseteq Q_{1,\texttt{lift}}$, then
  $$\big(1, \triangleright, V, (S_2, S_1)\big) \to (S_2 \cup R, \texttt{lift-pebble}) \in \mu'$$
  where $R = \{p \mid (1, \sigma, V, q) \to (p, \texttt{lift-pebble}) \in \mu \text{ and } q \in S_1\}$.

The proof that $L(\mathcal{A}) = L(\mathcal{A}')$ is pretty straightforward, thus, omitted.

Remark 7. We note that the normalizations A1–A7 can be performed similarly on alternating weak k-PA, for every $k = 1, 2, \ldots$. Converting an alternating weak k-PA $\mathcal{A}$ to the non-deterministic version $\mathcal{A}'$ can be done as follows. The set of states of $\mathcal{A}'$ for general k is the set

$$2^{Q_k} \cup \big(2^{Q_k} \times 2^{Q_{k-1}}\big) \cup \cdots \cup \big(2^{Q_k} \times 2^{Q_{k-1}} \times \cdots \times 2^{Q_1}\big)$$

The initial state and the set of final states are the same $\{q_0\}$ and $2^F - \{\emptyset\}$, respectively. The set of transitions can be defined in a similar manner as above.

4. Decidability and undecidability of weak PA

In this section we will discuss the decidability issue of weak PA. We show that the emptiness problem for weak 3-PA is undecidable, while the same problem for weak 2-PA is decidable. The proof of the decidability of the emptiness problem for weak 2-PA will be the basis of the proof of the decidability of the same problem for top view weak PA.

Theorem 8 is similar to the proof of the undecidability of the emptiness problem for weak 5-PA in [13]. The main technical step in the proof of the undecidability of the emptiness problem of weak 3-PA is to show that the following Σ-data language

$$L_{\text{ord}} = \left\{ \binom{\sigma}{a_1} \cdots \binom{\sigma}{a_n} \binom{\$}{d} \binom{\sigma}{a_1} \cdots \binom{\sigma}{a_n} \colon a_1, \ldots, a_n \text{ are pairwise different} \right\}$$

is accepted by weak 3-PA, where $\Sigma = \{\sigma, \$\}$. This is similar to the proof of the same problem for weak 5-PA. Our main observation is that weak 3-PA are sufficient to accept the language L_{ord}. From this step, the undecidability can be easily obtained via a reduction from the Post Correspondence Problem (PCP). The formal details are presented in the following theorem.

Theorem 8. *The emptiness problem for weak* 3*-PA is undecidable.*

Proof. As mentioned above, the proof uses a reduction from the Post Correspondence Problem (PCP), which is well known to be undecidable [8].

An *instance* of PCP is a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$, where each $x_1, y_1, \ldots, x_n, y_n \in \{\alpha, \beta\}^*$. This instance has a *solution* if there exist indexes $i_1, \ldots, i_m \in \{1, \ldots, n\}$ such that $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$. The PCP asks whether a given instance of the problem has a solution.

In the following we show how to encode a solution of an instance of PCP into a data word which possesses properties that can be checked by a weak 3-PA. Let $\Sigma = \{1, \ldots, n, \alpha, \beta, \$\}$ and $x_i = \nu_{i,1} \cdots \nu_{i,l_i}$, for each $i = 1, \ldots, n$. Each string x_i is encoded as $\mathsf{Enc}(x_i) = \binom{\nu_{i,1}}{a_{i,1}} \cdots \binom{\nu_{i,l_i}}{a_{i,l_i}}$ where $a_{i,1}, \ldots, a_{i,l_i}$ are pairwise different.

The string $x_{i_1} x_{i_2} \cdots x_{i_m}$ can be encoded as

$$\mathsf{Enc}(x_{i_1}, x_{i_2}, \ldots, x_{i_m}) = \binom{i_1}{b_1} \mathsf{Enc}(x_{i_1}) \binom{i_2}{b_2} \mathsf{Enc}(x_{i_2}) \cdots \binom{i_m}{b_m} \mathsf{Enc}(x_{i_m})$$

where all the data values that appear in it are pairwise different. Note that even if $i_j = i_{j'}$ for some j, j', the data values that appear in $\mathsf{Enc}(x_{i_j})$ do not appear in $\mathsf{Enc}(x_{i_{j'}})$ and vice versa. The idea is that each data value is used to mark a place in the string.

Similarly, the string $y_{j_1} y_{j_2} \cdots y_{j_l}$ can be encoded as

$$\mathsf{Enc}(y_{j_1}, y_{j_2}, \ldots, y_{j_l}) = \binom{j_1}{c_1}\mathsf{Enc}(y_{j_1})\binom{j_2}{c_2}\mathsf{Enc}(y_{j_2})\cdots\binom{j_l}{c_l}\mathsf{Enc}(y_{j_l})$$

where the data values that appear in it are pairwise different.

The data word

$$\binom{i_1}{b_1}\mathsf{Enc}(x_{i_1})\cdots\binom{i_m}{b_m}\mathsf{Enc}(x_{i_m})\binom{\$}{d}\binom{j_1}{c_1}\mathsf{Enc}(y_{j_1})\cdots\binom{j_l}{c_l}\mathsf{Enc}(y_{j_l})$$

constitutes a solution to the instance of PCP if and only if

$$i_1 i_2 \cdots i_m = j_1 j_2 \cdots j_l \tag{1}$$

$$\mathsf{Proj}_\Sigma\big(\mathsf{Enc}(x_{i_1})\cdots\mathsf{Enc}(x_{i_m})\big) = \mathsf{Proj}_\Sigma\big(\mathsf{Enc}(y_{j_1})\cdots\mathsf{Enc}(y_{j_l})\big) \tag{2}$$

Now, in order to be able to check such property with weak 3-PA, we demand the following additional criteria.

1. $b_1 \cdots b_m = c_1 \cdots c_l$.
2. $\mathsf{Proj}_{\mathfrak{D}}(\mathsf{Enc}(x_{i_1})\cdots\mathsf{Enc}(x_{i_m})) = \mathsf{Proj}_{\mathfrak{D}}(\mathsf{Enc}(y_{j_1})\cdots\mathsf{Enc}(y_{j_l}))$.
3. For any two positions h_1 and h_2 where h_1 is to the left of the delimiter $\binom{\$}{d}$ and h_2 is to the right of the delimiter $\binom{\$}{d}$, if both of them have the same data value, then both of them are labeled with the same label.

All the criteria 1–3 imply Eqs. (1) and (2). Also note that criteria 1–2 resemble the language L_{ord}.

Because the data values that appear in $\mathsf{Proj}_{\mathfrak{D}}(\mathsf{Enc}(x_{i_1}), \ldots, \mathsf{Enc}(x_{i_m}))$ are pairwise different, all of them are checkable by three pebbles in the "weak" manner. For example, to check criterion 1, the automaton does the following.

- Check that $b_1 = c_1$.
- Check that for each $i = 1, \ldots, m-1$, there exists j such that $a_i a_{i+1} = b_j b_{j+1}$.
  It can be done by placing pebble 3 to read a_i and pebble 2 to read a_{i+1}, then using pebble 1 to search on the other side of \$ for the index j.
- Finally, check that $b_m = c_l$.

Criterion 2 can be checked similarly and criterion 3 is straightforward. The reduction is now complete and we prove that the emptiness problem for weak 3-PA is undecidable.

Now we are going to show that the emptiness problem for weak 2-PA is decidable. The proof is by simulating weak 2-PA by one-way alternating one register automata (1-RA). See [6] for the definition of alternating RA.

In fact, the simulation can be easily generalized to arbitrary number of pebbles. That is, weak k-PA can be simulated by one-way alternating $(k-1)$-RA. (Here $(k-1)$-RA stands for $(k-1)$-register automata.) This result settles a question left open in [13]: Can weak PA be simulated by alternating RA?

Theorem 9. *For every weak k-PA $\mathcal{A}$, there exists a one-way alternating $(k-1)$-RA $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$. Moreover, the construction of $\mathcal{A}'$ from $\mathcal{A}$ is effective.*

Now, by Theorem 9, we immediately obtain the decidability of the emptiness problem for weak 2-PA because the same problem for one-way alternating 1-RA is decidable [6, Theorem 4.4].

Corollary 10. *The emptiness problem for weak 2-PA is decidable.*

We devote the rest of this section to the proof of Theorem 9 for $k = 2$. Its generalization to arbitrary $k \geqslant 3$ is pretty straightforward.

Recall that we may always assume that weak PA are deterministic. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \mu, F \rangle$ be a weak 2-PA. As in Section 3.1, we normalize the behavior of $\mathcal{A}$ as follows.

- Pebble 1 is lifted only after it reads the right-end marker $\triangleright$.
- The automaton can only enter a final state when the control is in pebble 2. Furthermore, it does so only after pebble 2 reads the right-end marker $\triangleright$.

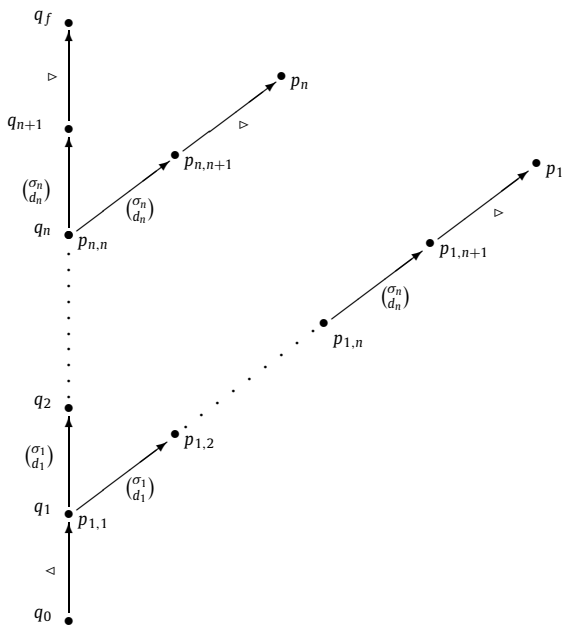

Fig. 1. The tree representation of a run of $\mathcal{A}$ on the data word $w = \binom{\sigma_1}{d_1}\cdots\binom{\sigma_n}{d_n}$.

- Immediately after pebble 2 moves right, pebble 1 is placed.
- Immediately after pebble 1 is lifted, pebble 2 moves right.

On input word $w = \binom{\sigma_1}{d_1}\cdots\binom{\sigma_n}{d_n}$, the run of $\mathcal{A}$ on $\triangleleft w \triangleright$ can be depicted as a tree shown in Fig. 1.
The meaning of the tree is as follows.

- $q_0, q_1, \ldots, q_n, q_{n+1}$ are the states of $\mathcal{A}$ when pebble 2 is the head pebble reading the positions $0, 1, \ldots, n, n+1$, respectively, that is, the symbols $\triangleleft, \binom{\sigma_1}{d_1}, \ldots, \binom{\sigma_n}{d_n}, \triangleright$, respectively.
- q_f is the state of $\mathcal{A}$ after pebble 2 reads the symbol $\triangleright$.
- For $1 \leqslant i \leqslant j \leqslant n$, $p_{i,j}$ is the state of $\mathcal{A}$ when pebble 1 is the head pebble reading the position j, while pebble 2 is above the position i.
- For $1 \leqslant i \leqslant n$, the state p_i is the state of $\mathcal{A}$ immediately after pebble 1 is lifted and pebble 2 is above the position i.
  It must be noted that there is a transition $(2, \sigma_i, \emptyset, p_i) \to (q_{i+1}, \texttt{right})$ applied by $\mathcal{A}$ that is not depicted in the figure.

Now the simulation of $\mathcal{A}$ by a one-way alternating 1-RA $\mathcal{A}'$ becomes straightforward if we view the tree in Fig. 1 as a tree depicting the computation of $\mathcal{A}'$ on the same word w. Fig. 2 shows the corresponding run of $\mathcal{A}'$ on the same word. Roughly, the automaton $\mathcal{A}'$ is defined as follows.

- The states of $\mathcal{A}'$ are elements of $Q \cup (Q \times Q)$;
- the initial state is q_0; and
- the set of final states is $F \cup \{(p, p) \colon\ p \in Q\}$.

For each placement of pebble 1 on position i, the automaton performs the following "Guess–Split–Verify" procedure which consists of the following steps.

1. From the state q_i, the automaton $\mathcal{A}'$ "guesses" (by disjunctive branching) the state in which pebble 1 is eventually lifted, i.e. the state p_i. It stores p_i in its internal state and simulates the transition $(2, \sigma_i, \emptyset, \emptyset, q_i) \to (p_{i,i}, \texttt{place-pebble}) \in \mu$ to enter into the state $(p_{i,i}, p_i)$.
2. The automaton $\mathcal{A}'$ "splits" its computation (by conjunctive branching) into two branches.
   - In one branch, assuming that the guess p_i is correct, $\mathcal{A}'$ moves right and enters into the state q_{i+1}, simulating the transition $(2, \emptyset, p_i) \to (q_{i+1}, \texttt{right})$.
     After this, it recursively performs the Guess–Split–Verify procedure for the next placement of pebble 1 on position $(i+1)$.
   - In the other branch $\mathcal{A}'$ stores the data value d_i in its register and simulates the run of pebble 1 on $\binom{\sigma_i}{d_i}\cdots\binom{\sigma_n}{d_n}$, starting from the state $p_{i,i}$, to "verify" that the guess p_i is correct.

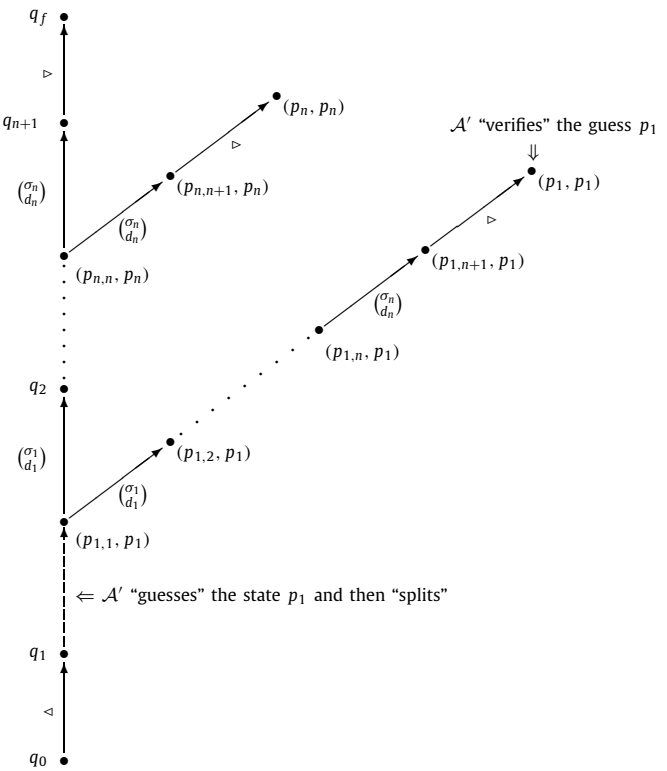

Fig. 2. The corresponding run of $\mathcal{A}'$ the data word $w = \binom{\sigma_1}{d_1} \cdots \binom{\sigma_n}{d_n}$ to the one in Fig. 1.

During the simulation, the states of $\mathcal{A}'$ are $(p_{i,i}, p_i), \ldots, (p_{i,n+1}, p_i)$. $\mathcal{A}'$ accepts when the simulation ends in the state (p_i, p_i), that is, when the guess p_i is "correct."

5. Complexity of weak 2-PA

In this section we study the time complexity of three specific problems related to weak 2-PA.

Emptiness problem. The emptiness problem for weak 2-PA. That is, given a weak 2-PA $\mathcal{A}$, is $L(\mathcal{A}) = \emptyset$?

Labeling problem. Given a deterministic weak 2-PA $\mathcal{A}$ over the labels Σ and a sequence of data values $d_1 \cdots d_n \in \mathfrak{D}^n$, is there a sequence of labels $\sigma_1 \cdots \sigma_n \in \Sigma^n$ such that $\binom{\sigma_1}{d_1} \cdots \binom{\sigma_n}{d_n} \in L(\mathcal{A})$?

Data value membership problem. Given a deterministic weak 2-PA $\mathcal{A}$ over the labels Σ and a sequence of finite labels $\sigma_1 \cdots \sigma_n \in \Sigma^n$, is there a sequence of data values $d_1 \cdots d_n \in \mathfrak{D}^n$ such that $\binom{\sigma_1}{d_1} \cdots \binom{\sigma_n}{d_n} \in L(\mathcal{A})$?

The emptiness problem, as we have seen in the previous section, is decidable. The labeling and data value membership problem are in NP. To solve the labeling problem, one simply guesses a sequence $\sigma_1 \cdots \sigma_n \in \Sigma^n$ and runs $\mathcal{A}$ to check whether $\binom{\sigma_1}{d_1} \cdots \binom{\sigma_n}{d_n} \in L(\mathcal{A})$. Similarly, to solve the data value membership problem, one can guess a sequence of data values $d_1 \cdots d_n$ and run $\mathcal{A}$ to check whether $\binom{\sigma_1}{d_1} \cdots \binom{\sigma_n}{d_n} \in L(\mathcal{A})$.

We will show that the emptiness problem is not primitive recursive, while both the labeling and data value membership problems are NP-complete.

Theorem 11. *The emptiness problem for weak* 2*-PA is not primitive recursive.*

The proof of Theorem 11 is by simulation of incrementing counter automata and follows closely the proof of a similar lower bound for one-way alternating 1-RA [6, Theorem 2.9].

An *incrementing counter automaton* is a finite, non-deterministic automaton extended with n counters, numbered from 1 to n. The transition relation is a subset of

$$Q \times \Sigma \times \left\{ \texttt{inc}(i), \texttt{dec}(i), \texttt{if-zero}(i) \;\middle|\; i = 1, \ldots, n \right\} \times Q$$

In each step, the automaton can change its state and modify the counters (either increment `inc` or decrement `dec` them), or testing whether they are zero (`if-zero`), according to the current state and the current symbol. Moreover, in each step

an incremental error may occur and the content of the counters may increase, hence the name *incrementing*. As usual, the automaton starts the computation in the initial state and accepts a word if it ends in one of the designated final states.

The main technical step of the proof of Theorem 11 is to show that the following Σ-data language L_{inc} which consists of the data words of the form: $\binom{\alpha}{a_1}\cdots\binom{\alpha}{a_m}\binom{\beta}{b_1}\cdots\binom{\beta}{b_n}$; where

- $\Sigma = \{\alpha, \beta\}$;
- the data values $a_1, \ldots, a_m$ are pairwise different;
- the data values $b_1, \ldots, b_n$ are pairwise different;
- each a_i appears among $b_1, \ldots, b_n$

is accepted by a weak 2-PA. The intuition of this language is to represent the inequality $m \leqslant n$, which is used to simulate the incrementing error of the counters. Since the emptiness problem of incrementing counter automata is not primitive recursive [11] (but still decidable [14]), Theorem 11 follows.

Now we are going to show the NP-hardness of the labeling problem. It is by a reduction from graph 3-colorability problem which states as follows. Given an undirected graph $G = (V, E)$, is G 3-colorable?

In the following we may assume that the data values are taken from the set of natural numbers $\{1, 2, \ldots\}$. Let $V = \{1, \ldots, n\}$ and $E = \{(i_1, j_1), \ldots, (i_m, j_m)\}$. Assuming that $\mathfrak{D}$ contains the natural numbers, we take $i_1 j_1 \cdots i_m j_m$ as the sequence of data values. Then, we construct a weak 2-PA $\mathcal{A}$ over the alphabet $\Sigma = \{\vartheta_R, \vartheta_G, \vartheta_B\}$ that accepts data words of even length in which the following hold.

- For all odd position x, the label on position x is different from the label on position $x + 1$.
- For every two positions x and y, if they have the same data value, then they have the same label. (See Example 4.)

Thus, the graph G is 3-colorable if and only if there exists

$$\sigma_1 \cdots \sigma_{2m} \in \{\vartheta_R, \vartheta_G, \vartheta_B\}^*$$

such that $\binom{\sigma_1}{i_1}\binom{\sigma_2}{j_1}\cdots\binom{\sigma_{2m-1}}{i_m}\binom{\sigma_{2m}}{j_m} \in L(\mathcal{A})$, and the NP-hardness, hence the NP-completeness, of the labeling problem follows.

The NP-hardness of data value membership problem can be established in a similar spirit. The reduction is from the following variant of graph 3-colorability, called 3-colorability with constraint. Given a graph $G = (V, E)$ and three integers n_r, n_g, n_b, is the graph G 3-colorable with the colors R, G and B such that the numbers of vertices colored with R, G and B are n_r, n_g and n_b, respectively?

The polynomial time reduction to data value membership problem is as follows. Let $V = \{1, \ldots, n\}$ and $E = \{(i_1, j_1), \ldots, (i_m, j_m)\}$.

We define $\Sigma = \{\vartheta_R, \vartheta_G, \vartheta_B, \nu_1, \ldots, \nu_n\}$ and take

$$\nu_{i_1}\nu_{j_1}\cdots\nu_{i_m}\nu_{j_m}\underbrace{\vartheta_R\cdots\vartheta_R}_{n_r \text{ times}}\underbrace{\vartheta_G\cdots\vartheta_G}_{n_g \text{ times}}\underbrace{\vartheta_B\cdots\vartheta_B}_{n_b \text{ times}}$$

as the sequence of finite labels. Then, we construct a weak 2-PA over Σ that accepts data words of the form

$$\binom{\nu_{i_1}}{c_1}\binom{\nu_{j_1}}{d_1}\cdots\binom{\nu_{i_m}}{c_m}\binom{\nu_{j_m}}{d_m}\binom{\vartheta_R}{a_1}\cdots\binom{\vartheta_R}{a_{n_r}}\binom{\vartheta_G}{a'_1}\cdots\binom{\vartheta_G}{a'_{n_g}}\binom{\vartheta_B}{a''_1}\cdots\binom{\vartheta_B}{a''_{n_b}}$$

where

- $\nu_{i_1}, \nu_{j_1}, \ldots, \nu_{i_m}, \nu_{j_m} \in \{\nu_1, \ldots, \nu_n\}$;
- in the sub-word $\binom{\nu_{i_1}}{c_1}\binom{\nu_{j_1}}{d_1}\cdots\binom{\nu_{i_m}}{c_m}\binom{\nu_{j_m}}{d_m}$, every two positions with the same labels have the same data value (see Example 4);
- the data values $a_1, \ldots, a_{n_r}, a'_1, \ldots, a'_{n_g}, a''_1, \ldots, a''_{n_b}$ are pairwise different;
- for each $i = 1, \ldots, m$, the values c_i, d_i appear among $a_1, \ldots, a_{n_r}, a'_1, \ldots, a'_{n_g}, a''_1, \ldots, a''_{n_b}$ such that the following holds:
  - if c_i appears among

    $$a_1, \ldots, a_{n_r}$$

    then d_i appears either among

    $$a'_1, \ldots, a'_{n_g} \quad \text{or} \quad a''_1, \ldots, a''_{n_b}$$

  - if c_i appears among

    $$a'_1, \ldots, a'_{n_g}$$

then d_i appears either among

$$a_1, \ldots, a_{n_r} \quad \text{or} \quad a''_1, \ldots, a''_{n_b}$$

– if c_i appears among

$$a''_1, \ldots, a''_{n_b}$$

then d_i appears either among

$$a_1, \ldots, a_{n_r} \quad \text{or} \quad a'_1, \ldots, a'_{n_g}$$

Note that we can store the integers r, g, b and m in the internal states of $\mathcal{A}$, thus, enable $\mathcal{A}$ to "count" up to n_r, n_g, n_b and m. We have each state for the numbers $1, \ldots, n_r$, $1, \ldots, n_g$, $1, \ldots, n_b$ and $1, \ldots, m$. The number of states in $\mathcal{A}$ is still polynomial on n.

Now the graph G is 3-colorable with the constraints n_r, n_g, n_b if and only if there exists $c_1 d_1 \cdots c_m d_m a_1 \cdots a_{n_r} a'_1 \cdots a'_{n_g} a''_1 \cdots a''_{n_b}$ such that

$$\binom{v_{i_1}}{c_1}\binom{v_{j_1}}{d_1}\cdots\binom{v_{i_m}}{c_m}\binom{v_{j_m}}{d_m}\binom{\vartheta_R}{a_1}\cdots\binom{\vartheta_R}{a_{n_r}}\binom{\vartheta_G}{a'_1}\cdots\binom{\vartheta_G}{a'_{n_g}}\binom{\vartheta_B}{a''_1}\cdots\binom{\vartheta_B}{a''_{n_b}}$$

is accepted by $\mathcal{A}$. The NP-hardness, hence the NP-completeness, of the data value membership problem then follows.

6. Top view weak k-PA

In this section we are going to define *top view* weak PA. Roughly speaking, top view weak PA are weak PA where the equality test is performed only between the data values seen by the last and the second to last placed pebbles. That is, if pebble i is the head pebble, then it can only compare the data value it reads with the data value read by pebble $(i+1)$. It is not allowed to compare its data value with those read by pebble $(i+2), (i+3), \ldots, k$.

Formally, a top view weak k-PA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \mu, F \rangle$ where Q, q_0, F are as usual and μ consists of transitions of the form: $(i, \sigma, V, q) \to (q', \mathtt{act})$, where V is either $\emptyset$ or $\{i+1\}$.

The criterion for the application of the transitions of top view weak k-PA is defined by setting

$$V = \begin{cases} \emptyset, & \text{if } a_{\theta(i+1)} \neq a_{\theta(i)} \\ \{i+1\}, & \text{if } a_{\theta(i+1)} = a_{\theta(i)} \end{cases}$$

in the definition of transition relation in Section 2.1. Note that top view weak 2-PA and weak 2-PA are the same.

Remark 12. We can also define an alternating version of top view weak k-PA. However, just like in the case of weak k-PA, alternating, non-deterministic and deterministic top view weak k-PA have the same recognition power. Furthermore, by using the same proof presented in Section 5, it is straightforward to show that the emptiness problem, the labeling problem, and the data value membership problem have the same complexity lower bound for top view weak k-PA, for each $k = 2, 3, \ldots$.

The following theorem is a stronger version of Theorem 9, as weak 2-PA are just special cases of top view weak PA.

Theorem 13. *For every top view weak k-PA $\mathcal{A}$, there is a one-way alternating* 1*-RA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A})$. Moreover, the construction of $\mathcal{A}'$ is effective.*

Proof. The proof is a straightforward generalization of the proof of Theorem 9. Each placement of a pebble is simulated by "Guess–Split–Verify" procedure. Since each pebble i can only compare its data value with the one seen by pebble $(i+1)$, $\mathcal{A}'$ does not need to store the data values seen by pebbles $(i+2), \ldots, k$. It only needs to store the data value seen by pebble $(i+1)$, thus, one register is sufficient for the simulation. □

Following Theorem 13, we immediately obtain the decidability of the emptiness problem for top view weak k-PA.

Corollary 14. *The emptiness problem for top view weak k-PA is decidable.*

Since the emptiness problem for ordinary 2-PA (see [10, Theorem 4]) and for weak 3-PA is already undecidable, it seems that top view weak PA is a tight boundary of a subclass of PA languages for which the emptiness problem is decidable.

Remark 15. In [16] it is shown that for every sentence $\psi \in \mathrm{LTL}_1^{\downarrow}(\Sigma, \mathtt{X}, \mathtt{U})$, there exists a weak k-PA $\mathcal{A}_\psi$, where $k = \mathsf{fqr}(\psi)+1$, such that $L(\mathcal{A}_\psi) = L(\psi)$. We remark that the proof actually shows that the automaton $\mathcal{A}_\psi$ is a top view weak k-PA. Thus, it shows that the class of top view weak k-PA languages contains the languages definable by $\mathrm{LTL}_1^{\downarrow}(\Sigma, \mathtt{X}, \mathtt{U})$.

7. Top view weak PA with unbounded number of pebbles

This section contains our quick observation on top view weak PA. We note that the finiteness of the number of pebbles for top view weak PA is not necessary. In fact, we can just define top view weak PA with unbounded number of pebbles, which we call top view weak unbounded PA.

We elaborate on it in the following paragraphs. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \mu, F \rangle$ be top view weak unbounded PA. The pebbles are numbered with the numbers $1, 2, 3, \ldots$. The automaton $\mathcal{A}$ starts the computation with only pebble 1 on the input word. The transitions are of the form: $(\sigma, \chi, q) \to (p, \mathtt{act})$, where $\chi \in \{0, 1\}$ and $\sigma, q, p, \mathtt{act}$ are as in the ordinary weak PA.

Let $w = \binom{\sigma_1}{a_1} \cdots \binom{\sigma_n}{a_n}$ be an input word. A *configuration of $\mathcal{A}$ on* $\lhd w \rhd$ is a triple $[i, q, \theta]$, where $i \in \mathbb{N}$, $q \in Q$, and $\theta : \mathbb{N} \to \{0, 1, \ldots, n, n+1\}$. The *initial* configuration is $[1, q_0, \theta_0]$, where $\theta_0(1) = 0$. The accepting configurations are defined similarly as in ordinary weak PA.

A transition $(\sigma, \chi, p) \to \beta$ *applies to a configuration* $[i, q, \theta]$, if

(1) $p = q$, and $\sigma_{\theta(i)} = \sigma$,
(2) $\chi = 1$ if $a_{\theta(i-1)} = a_{\theta(i)}$, and $\chi = 0$ if $a_{\theta(i-1)} \neq a_{\theta(i)}$,

Similarly, the transition relation $\vdash$ can be defined as follows: $[i, q, \theta] \vdash_{\mathcal{A}} [i', q', \theta']$, if there is a transition $\alpha \to (p, \mathtt{act}) \in \mu$ that applies to $[i, q, \theta]$ such that $q' = p$, for all $j < i$, $\theta'(j) = \theta(j)$, and

- if $\mathtt{act} = \mathtt{right}$, then $i' = i$ and $\theta'(i) = \theta(i) + 1$,
- if $\mathtt{act} = \mathtt{lift\text{-}pebble}$, then $i' = i - 1$,
- if $\mathtt{act} = \mathtt{place\text{-}pebble}$, then $i' = i + 1$, $\theta'(i+1) = \theta'(i) = \theta(i)$.

The acceptance criteria can be defined similarly.

It is straightforward to show that 1-way deterministic 1-RA can be simulated by top view weak unbounded PA. Each time the register automaton changes the content of the register, the top view weak unbounded PA places a new pebble.

Furthermore, top view weak unbounded PA can be simulated by 1-way alternating 1-RA. Each time a pebble is placed, the register automaton performs the procedure "Guess–Split–Verify" described in Section 4. Thus, the emptiness problem for top view unbounded weak PA is still decidable.

8. Concluding remark

In this paper we study pebble automata for data languages. In particular, we establish a fragment of PA languages for which the emptiness problem is decidable, the so-called top view weak PA. As shown in this paper, top view weak PA inherit some nice properties mentioned in Section 1.

1. Expressiveness: Top view weak PA contain the languages expressible by $\mathrm{LTL}_1^{\downarrow}(\Sigma, \mathtt{X}, \mathtt{U})$.
2. Decidability: The emptiness problem is decidable.
3. Efficiency: The model checking problem, that is, testing whether a given string of length n is accepted by a specific deterministic top view weak k-PA can be solved in $O(n^k)$ computation time.
4. Closure properties: Top view weak k-PA languages are closed under all boolean operations.
5. Robustness: Alternation and non-determinism do not add expressive power to top view weak k-PA languages.

There is still a lot of work to be done. In order to be applicable in program verification and XML settings, the model should work on infinite strings and unranked trees, respectively. Thus, the question remains whether it is possible to extend top view weak PA to the settings of infinite strings and unranked trees, while still preserving the five properties mentioned above.

Acknowledgments

The author would like to thank Michael Kaminski for his invaluable directions and guidance related to this paper and for pointing out the notion of unbounded pebble automata.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Mayank Saksena, A survey of regular model checking, in: Proceedings of the 15th International Conference on Concurrency Theory (CONCUR) 2004, in: Lecture Notes in Comput. Sci., vol. 3170, Springer, 2004, pp. 35–48.
[2] Marcelo Arenas, Wenfei Fan, Leonid Libkin, Consistency of XML specifications, in: Inconsistency Tolerance [Dagstuhl Seminar], in: Lecture Notes in Comput. Sci., vol. 3300, Springer, 2005, pp. 15–41.
[3] Henrik Björklund, Thomas Schwentick, On notions of regularity for data languages, in: Proceedings of the 16th International Symposium on Fundamentals of Computation Theory (FCT) 2007, in: Lecture Notes in Comput. Sci., vol. 4639, Springer, 2007, pp. 88–99.

[4] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, Claire David, Two-variable logic on words with data, in: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS) 2006, IEEE Computer Society, 2006, pp. 7–16.
[5] Mikolaj Bojanczyk, Mathias Samuelides, Thomas Schwentick, Luc Segoufin, Expressive power of pebble automata, in: Proceedings of Automata, Languages and Programming, 33rd International Colloquium (ICALP) 2006, in: Lecture Notes in Comput. Sci., vol. 4051, Springer, 2006, pp. 157–168.
[6] Stéphane Demri, Ranko Lazic, LTL with the freeze quantifier and register automata, in: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS) 2006, IEEE Computer Society, 2006, pp. 17–26.
[7] Allen Emerson, Kedar Namjoshi, Reasoning about rings, in: Proceedings of the 22nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) 1995, 1995, pp. 85–94.
[8] John Hopcroft, Jeffrey Ullman, Introduction to Automata Theory, Languages and Computation, Addison–Wesley, 1979.
[9] Michael Kaminski, Nissim Francez, Finite-memory automata, Theoret. Comput. Sci. 134 (2) (1994) 329–363.
[10] Michael Kaminski, Tony Tan, A note on two-pebble automata over infinite alphabets, Fund. Inform. 98 (4) (2010) 379–390.
[11] Richard Mayr, Undecidable problems in unreliable computations, Theoret. Comput. Sci. 297 (1–3) (2003) 337–354.
[12] Frank Neven, Automata, logic, and XML, in: Proceedings of the 11th Annual Conference of the EACSL Computer Science Logic (CSL) 2002, in: Lecture Notes in Comput. Sci., vol. 2471, Springer, 2002, pp. 2–26.
[13] Frank Neven, Thomas Schwentick, Victor Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Log. 5 (3) (2004) 403–435.
[14] Philippe Schnoebelen, Verifying lossy channel systems has nonprimitive recursive complexity, Inform. Process. Lett. 83 (5) (2002) 251–261.
[15] Luc Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Proceedings of the 15th Annual Conference of the EACSL Computer Science Logic (CSL) 2006, in: Lecture Notes in Comput. Sci., vol. 4207, Springer, 2006, pp. 41–57.
[16] Tony Tan, Graph reachability and pebble automata over infinite alphabets, in: Proceedings of the 24th IEEE Symposium on Logic in Computer Science (LICS) 2009, IEEE Computer Society, 2009, pp. 157–166.
[17] Tony Tan, On pebble automata for data languages with decidable emptiness problem, in: Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS) 2009, in: Lecture Notes in Comput. Sci., vol. 5734, Springer, 2009, pp. 712–723.