Complexity of pebble tree-walking automata

Mathias Samuelides and Luc Segoufin

LIAFA, Paris 7 INRIA, Paris 11

Abstract. We consider tree-walking automata using k pebbles. The pebbles are either *strong* (can be lifted from anywhere) or *weak* (can be lifted only when the automaton is on it). For each k, we give the precise complexities of the problems of emptiness and inclusion of tree-walking automata using k pebbles.

1 Introduction

There are two natural ways to extend the classical finite state string automata on finite binary trees.

In the first one, which is the most studied one in the literature (see for instance [5]), the automata have parallel control and process the tree bottom-up. It forms a robust class of automata (it has minimization and determinization) and the class of languages accepted by them enjoys most of the nice properties of the string case. For instance it is closed under all Boolean operations and it corresponds to MSO definability. Tree languages accepted by bottom-up tree automata are called *regular*.

The second kind of tree automata has only one control. It is a sort of sequential automaton which moves from node to node in a tree, along its edges. They are called tree-walking automata and, in a sense, generalize the notion of two-way string automata by making use of all possible directions allowed in a tree [1, 10]. However they are not determinizable [2] and have a rather weak expressive power [3]. For this reason pebble tree automata were introduced in [7] as a model with an interesting intermediate expressive power between tree-walking automata and bottom-up tree automata. A pebble tree automaton is a tree-walking automaton with a finite set $\{1, \ldots, k\}$ of pebbles which it can drop at and lift from a node. There is a stack discipline restriction though: pebble i can only be dropped at the current node if pebbles $i+1, \ldots, k$ are already on the tree. Likewise, if pebbles $i, \ldots, k$ are on the tree only pebble i can be lifted. In the first model of pebble automata the pebbles were only allowed to be lifted by the automaton when its head is on it, but recently, in order to capture logics with transitive closure on trees, a stronger model of pebble automata was introduced in [9]. In the strong model pebbles are viewed as pointers and can be lifted from everywhere. Perhaps surprisingly, in [4] it was shown that the two models of pebble tree automata have the same expressive power. More precisely it was shown that for each k and each pebble tree automaton using k pebbles with the strong behavior, there exists a pebble tree automaton using k pebbles with the

weak behavior accepting the same tree language. However the current translation yields an automaton whose size is a tower of $k-1$ exponents. It seems, but this has not been proved yet, that pebble tree automata using k strong pebbles are $(k-1)$-exponentially more succinct than pebble tree automata using k weak pebbles.

It is still conceivable that the class of pebble automata forms a robust class of tree languages. The main open issues are whether they are determinizable and whether they are closed under complement (the former would imply the later as mentioned in [14]). If they would be closed under complement, the family of tree languages accepted by pebble automata would also correspond to definability in unary transitive closure logic on trees [7]. The recent new interest in this family comes from their close relationship with some aspects of XML languages: They are a building block of *pebble transducers* which were used to capture XML transformations (cf. [13, 11]).

In this paper we study the complexity of emptiness test for pebble automata and the complexity of testing whether one pebble automaton is included into another.

From each pebble automaton, weak or strong, it is easy to compute an equivalent MSO formula [7]. This shows that they define only regular tree languages and immediately yields a non-elementary test for emptiness and inclusion. This non-elementary complexity is unavoidable as shown in [13].

We are interested in the problem when k, the number of pebbles, is fixed and not part of the input. Emptiness and inclusion for tree-walking automata (the case $k = 0$) are EXPTIME-complete. The upper-bound follows from the exponential time transformations of tree-walking automata and their complement into top-down tree automata given in [6, 16]. The lower-bound is implicit in [15]. For $k > 0$, we extend these results and show that both emptiness and inclusion are k-EXPTIME-complete. For each k, we prove the upper-bounds using the strong model and the lower-bounds using the weak and deterministic model. Therefore all variants considered here yield k-EXPTIME-complete problems.

The upper-bounds are proved by constructing, in time k-exponential, a bottom-up tree automaton for the language recognized by a tree-walking automata using k-pebbles and another one for the complement language. This is done by induction on the number of pebbles using an intermediate model which combines a tree-walking behavior with a bottom-up one. This induction is quite simple in the case where all pebbles have a weak behavior. In this case the subrun between the drop and the lift of the last pebble starts and ends at the same node and can therefore be replaced by a regular test. It is then possible to remove the last pebble by computing a product automaton. In the case of strong pebbles, subruns start and end at different nodes of the tree, and this complicates the construction.

The lower-bounds are proved by simulating a run of an alternating Turing machine using $(k-1)$-EXPSPACE by a deterministic pebble automaton using k pebbles with weak behaviors.

For each k, the complexities obtained for strong and weak pebble automata are the same. However we conjecture that pebble automaton using k strong pebbles is $(k-1)$-exponentially more succinct than pebble automaton using k weak pebbles. Therefore the strong model is more interesting as it achieves similar performances in terms of expressive power and of complexity but with a more succinct presentation.

When restricted to string models, our results show that both emptiness and inclusion for pebble automata using k pebbles are $(k-1)$-EXPSPACE-complete. Pebble string automata were already studied in [12] where it was shown that a pebble string automaton using k weak pebbles is k-exponentially more succinct than an one-way finite state automaton (the use of pebbles in [12] is actually even more restricted than the weak behavior mentioned above). The coding for proving our lower bounds is inspired from this result.

2 Definitions

The trees we consider are finite binary trees, with nodes labeled over a finite alphabet Σ. We insist that each internal node (non-leaf node) has exactly two children. A set of trees over a given alphabet is called a **tree language**.

Definition 2.1. *A* **bottom-up automaton** $\mathcal{B}$ *is a tuple* $(\Sigma, Q, q_0, F, \delta)$*, where* Q *is a finite set of states,* Σ *is a finite alphabet,* q_0 *is the initial state,* F *is the set of accepting states and* $\delta \subseteq (\Sigma \times Q \times Q) \times Q$ *is the transition relation.*

A **run** of a bottom-up automaton $\mathcal{B}$ on a tree t is a function ρ from the set of nodes of t to Q such that for every node x of label σ,

- If x is a leaf, then $((\sigma, q_0, q_0), \rho(x)) \in \delta$.
- If x has two children x_1 and x_2, then $((\sigma, \rho(x_1), \rho(x_2)), \rho(x)) \in \delta$.

A run of $\mathcal{B}$ on t is **accepting**, and the tree t is **accepted** by $\mathcal{B}$, if the state at the root is accepting. The family of tree languages defined by bottom-up automata is called the class of **regular tree languages**.

Pebble tree automata. Informally a pebble tree automaton walks through its input tree from node to node along its edges. Additionally it has a fixed set of pebbles, numbered from 1 to k that it can place in the tree. At each time, pebbles $i, \cdots, k$ are placed on some nodes of the tree, for some i. In one step the automaton can stay at the current node, move to its parent , to its left or to its right child, or it can lift pebble i or place pebble $i-1$ on the current node. Which of these transitions can be applied depends on the current state, the set of pebbles at the current node, the label and the type of the current node (root, left or right child and leaf or internal node).

We consider two kinds of pebble automata which differ in the way they can lift the pebble. In the weak model a pebble can be lifted only if it is on the current node. In the strong model this restriction does not apply.

Remark: In both models the placement of the pebbles follows a stack discipline: only the pebble with the number i can be lifted and only the pebble with number $i-1$ can be placed. This restriction is essential as otherwise we would obtain k-head automata that recognize non-regular tree languages.

We turn to the formal definition of pebble automata. The set $\text{TYPES} = \{r, 0, 1\} \times \{l, i\}$ describes the possible **types** of a node. Here, r stands for the root, 0 for a left child, 1 for a right child, l for a leaf and i for an internal node. We indicate the possible kinds of moves of a pebble automaton by elements of the set $\text{MOVES} = \{\epsilon, \uparrow, \swarrow, \searrow, \text{lift}, \text{drop}\}$, where informally $\uparrow$ stands for 'move to parent', ϵ stands for 'stay', $\swarrow$ stands for 'move to the left child' and $\searrow$ stands for 'move to the right child'. Clearly drop refers to dropping a pebble and lift to lifting a pebble. Finally if S is a set then $\mathcal{P}(S)$ denotes the powerset of S.

Definition 2.2. *A* **pebble tree automaton** *using k pebbles is a tuple $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, where Q is a finite set of states, $I, F \subseteq Q$ are respectively the set of* **initial** *and* **terminal** *states and δ is the transition relation of the form*

$$\delta \subseteq \big(Q \times \text{TYPES} \times \{1, \ldots, k+1\} \times \mathcal{P}(\{1, \cdots, k\}) \times \Sigma\big) \times (Q \times \text{MOVES})$$

A tuple $(q, \beta, i, S, \sigma, q', m) \in \delta$ means that if $\mathcal{A}$ is in state q with pebbles $i, \cdots, k$ on the tree, the current node contains exactly the pebbles from S, has type β and is labeled by σ then $\mathcal{A}$ can enter state q' and move according to m.

A pebble set of $\mathcal{A}$ is a set $P \subseteq \{1, \cdots, k\}$. For a tree t, a P-pebble assignment is a function which maps each $j \in P$ to a node in t. For $0 \leq i \leq k$, an **i-configuration** c of $\mathcal{A}$ on t is a tuple (x, q, f), where x is a node, q a state and f an $\{i+1, \cdots, k\}$-pebble assignment. In this case x is called the current node, q the current state and f the current pebble assignment. We also write $(x, q, x_{i+1}, \cdots, x_k)$ if $f(j) = x_j$ for each $j \geq i+1$. We write $c \xrightarrow{\mathcal{A},t} c'$ when $\mathcal{A}$ can make a (single step) transition from the configuration c to c' according to its transition relation. The relation $\xrightarrow{\mathcal{A},t}$ is defined in the obvious way following the intuition described above for δ. However in the weak model of pebble tree automata there is a restriction on the lift-operation: a lift transition can be applied to an i-configuration (x, q, f) only if $f(i+1) = x$, i.e., if pebble $i+1$ is on the current node. In the strong model this restriction does not hold. A run is a nonempty sequence $c_1, \cdots, c_l$ of configurations such that $c_j \xrightarrow{\mathcal{A},t} c_{j+1}$ holds for each $j < l$. We write $c \xrightarrow{\mathcal{A},t\ *} c'$ when there is a run of A from c to c'.

Instead of having a set of accepting states and acceptance at the root as usual, we assume that a walking automaton has a set of **terminal** states. Once a terminal state is reached, the automaton immediately stops walking the tree. When the automaton is used as an acceptor for trees, we further assume a partition of the terminal states into accepting and rejecting ones (note that the automaton always rejects if no terminal state is ever reached). As an acceptor for tree languages, this hypothesis does not make any difference as it is always possible to go back to the root of the tree once a terminal state is reached.

A pebble tree automaton is **deterministic** if δ is a partial function from $Q \times \text{TYPES} \times \{1, \cdots, k+1\} \times \mathcal{P}(\{1, \cdots, k\}) \times \Sigma$ to $Q \times \text{MOVES}$.

We use PA_k to denote the (strong) pebble automata using k pebbles, wPA_k the weak pebble automata using k pebbles, DPA_k and wDPA_k for the corresponding deterministic automata. By default we assume the strong case. A pebble automaton without pebbles is just a tree-walking automaton. We write TWA and DTWA for PA_0 and DPA_0.

Complexities. In this paper k-EXPTIME refers to the set of problems solvable by a Turing machine using a time which is a tower of k exponentials of a polynomial in the size of its input. In order to avoid a case analysis we sometime write 0-EXPTIME for PTIME. Similarly we define k-EXPSPACE with 0-EXPSPACE for PSPACE.

3 From pebble automaton to bottom-up automaton

Given $\mathcal{A} \in \mathrm{PA}_k$ we construct in this section a bottom-up tree automaton $\mathcal{B}$ recognizing the same language as $\mathcal{A}$ and a bottom-up tree automaton $\mathcal{C}$ recognizing the language of trees rejected by $\mathcal{A}$. The constructions are performed in k-EXPTIME. They are done by induction on k. During the induction we shall make use of the following intermediate model of automata which combines a tree-walking behavior with a bottom-up one.

Intuitively a wPABU_k is a wPA_k that can simulate a bottom-up automaton while placing its last pebble on the current node. More formally we have:

Definition 3.1. *A* wPABU_k *is a pair* $(\mathcal{A}, \mathcal{B})$ *where*

- $\mathcal{B}$ *is a (non deterministic) bottom-up automaton on* $\Sigma \times \mathcal{P}(\{1, \cdots, k\})$
- $\mathcal{A}$ *is a* wPA_k *such that the transitions that drop pebble* 1 *are of the form*

$$(Q_{\mathcal{A}} \times \text{TYPES} \times \{2\} \times \mathcal{P}(\{2, \cdots, k\}) \times \Sigma \times Q_{\mathcal{B}}) \times (Q_{\mathcal{A}} \times \{\text{drop}\})$$

where $Q_{\mathcal{A}}$ *and* $Q_{\mathcal{B}}$ *are the set of states of* $\mathcal{A}$ *and* $\mathcal{B}$.

A wPABU_k $(\mathcal{A}, \mathcal{B})$ behaves like a pebble tree automaton until it wants to drop pebble 1. When it drops pebble 1, it immediately simulates $\mathcal{B}$ on the current pebbled tree and resumes its walking behavior with a state which depends on the state reached by $\mathcal{B}$ at the root of the tree as specified by the transition above.

More formally, for $0 \leq i \leq k$, an i-configuration of $(\mathcal{A}, \mathcal{B})$ on a tree t is a tuple (x, q, f) where x is a node of t, q a state of $\mathcal{A}$ and f an $\{i+1, \cdots, k\}$-pebble assignment. Let t be a tree, x a node of t of type τ and of label a and let $c = (x, q, f)$ and $c' = (x', q', f')$ be i-configurations of $(\mathcal{A}, \mathcal{B})$. A single step transition $c \xrightarrow{(\mathcal{A},\mathcal{B}),t} c'$ of $(\mathcal{A}, \mathcal{B})$ is defined as $c \xrightarrow{\mathcal{A},t} c'$ if $\mathcal{A}$ does not drop pebble 1 while making the transition step from c to c'. Otherwise we must have $x' = x$, $f' = f \cup \{(1, x)\}$ and $((q, \tau, 2, f^{-1}(x), a, q_b), (q', \text{drop}))$ is a transition of $(\mathcal{A}, \mathcal{B})$ where q_b is the state accessed by a run of $\mathcal{B}$ at the root of the pebbled tree (t, f').

In order to handle the strong behaviors of pebbles we need to extend this definition. The idea is to use BU^* automata instead of bottom-up automata. Intuitively a BU^* automaton is a bottom-up automaton that can select a node. More formally this means:

Definition 3.2. *A* BU* *automaton is a tuple* $(Q, q_0, Q_f, Q', \delta)$ *such that* $Q' \subseteq Q$ *and* (Q, q_0, Q_f, δ) *is a bottom-up tree automaton such that for each tree* t *and each accepting run* ρ *on* t *there is an unique node* $x \in t$ *with* $\rho(x) \in Q'$.

A PABU$_k^*$ is like a wPABU$_k$ but the pebble tree automaton part is strong and the bottom-up part is a BU*. When dropping the last pebble, the pebble automaton simulates the bottom-up part and resumes its run at the node selected by the BU* automaton by lifting the last pebble. More formally this gives:

Definition 3.3. *A* PABU$_k^*$ *is a pair* $(\mathcal{A}, \mathcal{B})$ *where*

- $\mathcal{B}$ *is a (non deterministic)* BU* *on* $\Sigma \times \mathcal{P}(\{1, \cdots, k\})$
- $\mathcal{A}$ *is a* PA$_k$ *such that the transitions dropping and lifting the last pebble are replaced by transitions of the form*

$$(Q_{\mathcal{A}} \times \text{TYPES} \times \{2\} \times \mathcal{P}(\{2, \cdots, k\}) \times \Sigma \times Q_{f\mathcal{B}}) \times Q_{\mathcal{A}}$$

where $Q_{\mathcal{A}}$ *is the set of states of* $\mathcal{A}$ *and* $Q_{f\mathcal{B}}$ *is the set of accepting states of* $\mathcal{B}$.

More formally, for $1 \leq i \leq k$, an i-configuration of $(\mathcal{A}, \mathcal{B})$ is a tuple (x, q, f) where x is a node, q a state of $\mathcal{A}$ and f an $\{i+1, \cdots, k\}$-pebble assignment.

Let t be a tree, x be a node of t of type τ and of label a and let $c = (x, q, f)$ and $c' = (x', q', f')$ be configurations of $(\mathcal{A}, \mathcal{B})$. A single step transition $c \xrightarrow{(\mathcal{A},\mathcal{B}),t} c'$ of $(\mathcal{A}, \mathcal{B})$ is defined as $c \xrightarrow{\mathcal{A},t} c'$ if $\mathcal{A}$ does not drop pebble 1 while making the transition step from c to c'. Otherwise we must have $f' = f$, x' is the node selected by $\mathcal{B}$ on the pebbled tree $(t, f \cup \{(1, x)\})$ and $((q, \tau, 2, f^{-1}(x), a, q_b), q')$ is a transition of $(\mathcal{A}, \mathcal{B})$ where q_b is the state accessed by a run of $\mathcal{B}$ at the root of the pebbled tree $(t, f \cup \{(1, x)\})$.

A node x in a tree t is a **marked node** if x is the unique node of t having the type or pebble assignment of x. If m is the marking type or pebble assignment, we then say that t is marked by m, and that x is the m-node of t. For instance a tree is always marked by its root. At any moment during the run of a tree-walking automaton, a tree is marked by any of the pebbles which are currently dropped.

Given a PABU$_k^*$ $(\mathcal{A}, \mathcal{B})$ and a BU* $\mathcal{C}$, we say that $\mathcal{C}$ **simulates exactly** $(\mathcal{A}, \mathcal{B})$ on trees marked by m if: (i) each set of pairs of states of $\mathcal{A}$ is an accepting state of $\mathcal{C}$, (ii) for each tree t marked by node u, $\mathcal{C}$ reaches the root of t after selecting node v, in an accepting state $q_f = \{(q, q') \mid (q, u, \_) \xrightarrow{(\mathcal{A},\mathcal{B}),t\ *} (q', v, \_), q' \text{ is terminal}\}$. In other words the state reached by $\mathcal{C}$ at the root contains exactly the beginning and the ending states of all the terminating runs of $\mathcal{A}$ between nodes u and v. Note that this implies that $\mathcal{C}$ is unambiguous once the choice of v is made.

In a sense, our first lemma below extends the result of [6, 16] for TWA, and translates a wPABU$_1$ and its complement into a bottom-up automaton. This is done with the same complexity bounds as for TWA: One exponential. The proof can be found in the appendix. The idea is classical, the bottom-up tree automaton has to compute all possible loops of the tree-walking automaton while moving up the tree. The main new difficulty is to take care of the loops which involve the use of the pebble.

Lemma 3.4. *Let* $(\mathcal{A}, \mathcal{B})$ *be a* wPABU$_1$*. For any marking* m*, we can construct in time exponential in* $|(\mathcal{A}, \mathcal{B})|$*,* $\mathcal{C} \in \mathrm{BU}^*$ *such that* $\mathcal{C}$ *simulates exactly* $(\mathcal{A}, \mathcal{B})$ *on trees marked by* m*.*

We now extend the previous lemma to the strong pebble case. This is done by reducing the strong pebble case to the weak pebble one. Given two walking automata $\mathcal{A}$ and $\mathcal{B}$ we say that $\mathcal{B}$ *simulates exactly* $\mathcal{A}$ if (i) the set of states of $\mathcal{A}$ is included into the set of states of $\mathcal{B}$, (ii) for all tree t and all nodes u and v of t we have $(u, q, \_) \xrightarrow{\mathcal{A},t} (v, q', \_)$ iff $(u, q, \_) \xrightarrow{\mathcal{B},t} (v, q', \_)$ for all pair (q, q') of states of $\mathcal{A}$.

Lemma 3.5. *Given* $(\mathcal{A}, \mathcal{B}) \in \mathrm{PABU}_1^*$*, we can construct in polynomial time* $(\mathcal{A}', \mathcal{B}')$ *in* wPABU$_1$ *simulating exactly* $(\mathcal{A}, \mathcal{B})$*.*

Proof. The idea is as follows, $\mathcal{A}'$ will simulate $\mathcal{A}$ until the pebble is dropped. Then $\mathcal{A}'$ will move step by step the pebble in the tree until it reaches the position where the pebble is lifted. The difficulty is that $\mathcal{A}'$ cannot find out which node is selected by $\mathcal{B}$, until $\mathcal{A}'$ is on that node, and that as soon as $\mathcal{A}'$ moves the pebble, the simulation of $\mathcal{B}$ is no longer valid. To cope with this situation, $\mathcal{A}'$ will maintain extra information in its state and only simulate $\mathcal{B}$ partially.

Assume now that $\mathcal{A}$ drops the pebble on the node x_d, evaluates $\mathcal{B}$ and resumes its run from node x_l after lifting the pebble. Let $Q_{\mathcal{B}}$ be the set of states of $\mathcal{B}$.

We show how to simulate this behavior using a wPABU$_1$ $(\mathcal{A}', \mathcal{B}')$. On the tree t there is an unique path from x_d to x_l. The goal of $\mathcal{A}'$ is to transfer step by step the pebble on that path. To do this, at any time, assuming its pebble is on position x in the path, it will remember in its state (i) the state q_r reached by $\mathcal{B}$ at the root of the tree when it was simulated by $\mathcal{A}$, (ii) the state q_x reached by $\mathcal{B}$ on the current node x when it was simulated by $\mathcal{A}$, and (iii) the direction from x to the next node on the path from x_d to x_l.

This information is computed and maintained using $\mathcal{B}'$.

To do this $\mathcal{B}'$ will do the following. It first guesses a state $q_x \in Q_{\mathcal{B}}$ and a direction $\Delta \in \{$ DownLeft, DownRight, UpRight, UpLeft, Init, Here$\}$, which are expected to match those currently stored in the state of $\mathcal{A}'$ ($\mathcal{A}'$ will verify this in the next step), except for the first time where Δ is Init. We then distinguish three cases:

- Case 1: Δ is Init. Then $\mathcal{B}'$ simulates $\mathcal{B}$ and ends in a state containing: q_x the state reached by $\mathcal{B}$ at the position of the pebble, $\Delta' \in \{$ DownLeft, DownRight, UpRight, UpLeft, Here$\}$ the direction from the pebble to the selected node and q_r the state reached by $\mathcal{B}$ at the root. Note that Δ' could be Here if the selected position is the current position of the pebble.
- Case 2: Δ is UpRight (the case UpLeft is similar).
  In this case, $\mathcal{B}'$ simulates $\mathcal{B}$ unless it reaches x (marked by the pebble). When x is reached, if $\mathcal{B}$ already selected its node in the left subtree of x, then $\mathcal{B}'$ rejects. Otherwise, the current state is ignored and $\mathcal{B}'$ recomputes the current state assuming the state q_x at the left child of x. It remembers the new state q'_x and the direction Δ' from x to the selected node and resumes

its simulation of $\mathcal{B}$. At the root $\mathcal{B}'$ ends in a state containing q_x, Δ and the state reached by $\mathcal{B}$ at the root during the current simulation together with q'_x and Δ'.

– Case 3: Δ is DownRight (the case DownLeft is similar).
In this case $\mathcal{B}'$ simulates $\mathcal{B}$ until it reaches x. When x is reached $\mathcal{B}'$ knows whether a node has been indeed selected in the right subtree of x. If this is not the case, or if the current state is not q_x, it rejects. If this is the case, $\mathcal{B}'$ remembers the state q'_x $\mathcal{B}$ has reached at the right child of x and also the direction Δ' from this right child to the node x_l. At the root $\mathcal{B}'$ accepts in a state containing q_x and Δ together with the state reached by $\mathcal{B}$ at the root during the current simulation together with q'_x and Δ'.

We can now define $\mathcal{A}'$. $\mathcal{A}'$ simulates $\mathcal{A}$ until the pebble is dropped at position x_d. At position x_d, when the pebble is first dropped it simulates $\mathcal{B}'$, verifies that $\mathcal{B}'$ indeed guessed the Init case and stores the output of $\mathcal{B}'$ in its state. It then does the following until a Here case is reached. It moves the pebble one step according to Δ, simulates $\mathcal{B}'$, verifies that the guessed values of $\mathcal{B}'$ are consistent with what it currently has in its state. If not it rejects, if yes it updates those values according to the output of $\mathcal{B}'$. When $\mathcal{B}'$ outputs Here then $\mathcal{A}'$ lifts the pebble and resumes the simulation of $\mathcal{A}$. □

We are now ready for the main induction loop.

Lemma 3.6. *Let $(\mathcal{A}, \mathcal{B})$ in PABU_k^* . Let m be any marking. We can construct in time k-exponential in $|(\mathcal{A}, \mathcal{B})|$, a BU^* $\mathcal{C}$ that simulates exactly $(\mathcal{A}, \mathcal{B})$ on trees marked by m.*

Proof. This is done by induction on n. The case $k = 1$ is given by combining Lemma 3.5 with Lemma 3.4. Assume now that the lemma is proved for k and we will prove it for $k+1$. Let $\mathcal{A}'$ be the PA_1 defined from $\mathcal{A}$ as follows. The states of A' are all the states of $\mathcal{A}$ corresponding to configurations where all the pebbles $k, \cdots, 2$ are dropped. The transitions of $\mathcal{A}'$ are all the transitions of $\mathcal{A}$ restricted to the states of $\mathcal{A}'$. The terminal states of $\mathcal{A}'$ are exactly those that lift pebble 2. The marking is the position of pebble 2. Then $(\mathcal{A}', \mathcal{B})$ and $(\mathcal{A}, \mathcal{B})$ have exactly the same runs from a node u where pebble 2 is dropped to a node v where it is next lifted. From Lemma 3.5 we obtain $(\mathcal{A}'', \mathcal{B}')$ in wPABU_1 simulating exactly $(\mathcal{A}', \mathcal{B})$ assuming pebbles $k, \cdots, 2$ are already on the tree. Now by Lemma 3.4 we obtain in exponential time a BU^* automaton $\mathcal{C}'$ that simulates exactly $(\mathcal{A}'', \mathcal{B}')$ on trees marked by pebble 2, assuming pebbles $k, \cdots, 2$ are already on the tree. Let now $(\mathcal{A}''', \mathcal{C}')$ be the PABU_{k-1}^* defined from $\mathcal{A}$ as follows. The states of $\mathcal{A}'''$ are all the states of A corresponding to all configurations where pebble 1 is not dropped. The terminal states of $\mathcal{A}'''$ are the terminal states of $\mathcal{A}$. The transitions of $\mathcal{A}'''$ are all the transitions of $\mathcal{A}$ restricted to the states of $\mathcal{A}'''$ where all the transitions dropping pebble 2 are now replaced by a simulation of $\mathcal{C}'$. It is easy to verify that $(\mathcal{A}''', \mathcal{C}')$ simulates exactly $(\mathcal{A}, \mathcal{B})$, and we obtain the desired $\mathcal{C}$ by induction. □

Theorem 3.7. *Let $\mathcal{A}$ in* PA_k*. We can construct in time k-exponential in $|\mathcal{A}|$ a bottom-up automaton C accepting the same language as $\mathcal{A}$ and a bottom-up automaton $\overline{\mathcal{C}}$ accepting the complement of the language accepted by $\mathcal{A}$.*

Proof. We shall make use of this lemma which shows that we can always assume that the last pebble is weak. Its proof is a straightforward adaptation of the proof of Lemma 3.5.

Lemma 3.8. *Let $\mathcal{A}$ be in* PA_k*. We can construct in time polynomial in $|\mathcal{A}|$ an automaton $\mathcal{B} \in \mathrm{PA}_k$ accepting the same tree language as $\mathcal{A}$ and such that the pebble 1 of $\mathcal{B}$ is weak.*

Let $\mathcal{A}$ in PA_k. Let $\mathcal{A}'$ in PA_k recognizing the same language as $\mathcal{A}$ but with pebble 1 weak as given by Lemma 3.8. Let $\mathcal{A}''$ be the wPA_1 defined from $\mathcal{A}'$ as follows. The states of $\mathcal{A}''$ are all the states of $\mathcal{A}'$ corresponding to configurations where all the pebbles $n, \cdots, 2$ are dropped. The transitions of $\mathcal{A}''$ are all the transitions of $\mathcal{A}'$ restricted to states of $\mathcal{A}''$. The terminal states of $\mathcal{A}''$ are exactly those that lift pebble 2. Let $\mathcal{B}$ be the trivial bottom-up tree automaton with only one state that does nothing. Then $(\mathcal{A}'', \mathcal{B})$ is a wPABU_1 having the same runs as $\mathcal{A}'$ from a node u where pebble 2 is dropped to a node v where it is next lifted. Now by Lemma 3.4 we obtain in exponential time a BU^* automaton $\mathcal{C}'$ that simulates exactly $(\mathcal{A}'', \mathcal{B})$ on trees marked by pebble 2. Let now $(\mathcal{A}''', \mathcal{C}')$ be the PABU^*_{k-1} defined from $\mathcal{A}$ as follows. The states of $\mathcal{A}'''$ are all the states of $\mathcal{A}$ corresponding to all configurations where pebble 1 is not dropped. The terminal states of $\mathcal{A}'''$ are the terminal states of $\mathcal{A}$. The transitions of $\mathcal{A}'''$ are all the transitions of $\mathcal{A}$ restricted to the states of $\mathcal{A}'''$ where all the transitions dropping pebble 2 are now replaced with a simulation of $\mathcal{C}'$. It is easy to verify that $(\mathcal{A}''', \mathcal{C}')$ simulates exactly $\mathcal{A}$. Let $\mathcal{D}$ be the BU^* obtained by applying Lemma 3.6 on trees marked by the root. Note that $\mathcal{D}$ always marks the root and therefore can be seen as a bottom-up tree automaton. By construction of $\mathcal{D}$, the state reached by $\mathcal{D}$ at the root of any input tree contains exactly all the pair (q, q') so that if $\mathcal{A}$ starts at the root in state q then it comes back at the root in state q'. It is now immediate to define $\mathcal{C}$ and $\overline{\mathcal{C}}$ from $\mathcal{D}$ by choosing appropriately the set of accepting states. □

The **emptiness problem** for pebble tree automata is the problem of checking, given a pebble tree automaton, whether it accepts at least one tree. The **inclusion problem** for pebble tree automata is the problem of checking, given two tree automata $\mathcal{A}$ and $\mathcal{B}$ whether any tree accepted by $\mathcal{A}$ is also accepted by $\mathcal{B}$. As the emptiness problem for bottom-up tree automata is in PTIME we immediately derive from Theorem 3.7 an upper-bound for the emptiness and inclusion problems for PA, and therefore for wPA.

Theorem 3.9. *Let $k > 0$. The emptiness and the inclusion problems for* PA_k *are in k-*EXPTIME*.*

4 Lower bounds

In this section we show that the complexities obtained previously are tight. We actually prove a stronger result as we show that the lower bounds already hold in the weak pebble model and with deterministic control.

For $k > 0$, let $\exp(k, n)$ be the function defined by $\exp(1, n) = 2^n$ and $\exp(k, n) = 2^{\exp(k-1,n)}$. A **$k$-number** of size n is defined recursively as follows. If $k = 1$ it is a tree formed by a root of label $\sharp$ followed by a unary tree forming a sequence of n bits, defining a number from 0 to $2^n - 1$ (this tree can be made binary by adding enough dummy extra nodes). For $k > 1$ it is a tree t having the following properties: (i) The root of t is labeled by $\sharp$, (ii) The path from the root of t to the rightmost leaf (excluding the root) contains exactly $\exp(k-1, n)$ nodes having label in $\{0, 1\}$. This path will be called: **rightmost branch**, (iii) The left child of each node x of the rightmost branch is a $(k-1)$-number that encodes the distance from $\sharp$ to x (the topmost branching node is assumed to have distance zero from $\sharp$).

We can easily see that the $exp(k-1, n)$ bits in the rightmost branch in t define a number from 0 to $\exp(k, n) - 1$.

In the rest of this section we blur the distinction between the root of a k-number and the integer it encodes. We will make use of the following terminology. If x is a k-number, the nodes of the rightmost branch of x are called the **bits** of x. For each such bit, the $(k-1)$-number branching off that node is called a **position** of x.

Let f be a function associating to a node x of a tree t a set $f(x)$ of nodes of t. Such a function f is said to be **determined** if there is a DTWA with a specific state q_S such that, when started at x in a tree t, it sequentially investigates all the nodes in $f(x)$, being in state q_S at a node y iff $y \in f(x)$. Typically determined functions are the set of positions of the k-number which is located immediately above or below x. In the following we will only use very simple determined functions f. In particular the size of the DTWA involved has a size which will not depend on the parameters k and n.

The main technical lemma is the following one which is inspired by the succinctness result of [12].

Lemma 4.1. *Let $n > 0$ and $k > 0$.*

1. *There exists a* wDPA$_{(k-1)}$*, of size polynomial in n, such that when started on a node x of a tree t, it returns to x in a state which is accepting iff the subtree rooted in x forms a k-number of size n.*
2. *For each determined function f, there exists a* wDPA$_k$*, of size polynomial in n, such that when started on a marked node x of a tree t, it returns to x in a state which is accepting iff there is a node $y \in f(x)$ such that x and y form the same k-number of size n.*
3. *There exists a* wDPA$_k$*, of size polynomial in n, such that when started on a node x of a tree t, it returns to x in a state which is accepting iff the k-number of size n rooted at the left child of x is the successor of the k-number of size n rooted at the left child of the right child of x.*

Proof. Fix $n > 0$. All items are proved simultaneously by induction on k. For point 2 let $\mathcal{A}_{\{}$ be the DTWA for f.

If $k = 1$, point 1 is clear. The automaton uses n states to check that the tree has the correct depth and then comes back to the initial place. For point 2, the automaton successively drops the pebble on each node y of the set of nodes in $f(x)$ using $\mathcal{A}_{\{}$, simulates the automaton obtained for point 1 to check that the subtrees of x and y are indeed 1-number. Once this is done it processes the subtrees of x and y bit per bit, going back and forth between x and y (recall that x is marked by hypothesis and that y is marked by the pebble), checking for equality. The current position being processed is stored in the state and this requires $O(n)$ states. For point 3 the pebble is dropped on the appropriate child of x and we proceed as for point 2, simulating addition with 1 instead of checking equality.

Assume now that $k > 1$. Consider point 1. By induction it is easy to verify with a $\mathrm{wDPA}_{(k-2)}$ that the subtree rooted in x has the right shape: it starts with a $\sharp$ and is a sequence of $(k-1)$-number. It remains to check that this sequence codes all $(k-1)$-number in the order from 0 to $\exp(k-1, n) - 1$. For each node y of the rightmost branch of x the automaton drops pebble k on y and simulates the $\mathrm{wDPA}_{(k-1)}$ obtained from point 3 by induction in order to verify that the positions of y and of the right child of y are successive $(k-1)$-number. Once this is done for each bit y of x the automaton goes back to x by going up in the tree until a $\sharp$ is found.

Consider now point 2. The automaton first checks by induction that x is the root of a k-number. Then for each node $y \in f(x)$ it does the following. It first checks whether y is the root of k-number and if this is the case it drops pebble k successively on each position z of y. Let g be the function which associates to z the set of positions of x. It is easy to see that g is determined by the deterministic automaton which from z goes back to x, which is marked, and then successively investigates all position of x by going down to the right child. By induction on point 2 the automaton can find with the remaining $k-1$ pebbles, among the positions of x, the one with the same $(k-1)$-number as z and checks that the associated bits are equal. If the bits are different the automaton comes back to z, lifts pebble k and moves back to y by going up until it finds a $\sharp$ and then proceeds with the next node of $f(x)$. If the bits match the automaton comes back to z, lifts pebble k and proceeds with the next bit of y. Once all bits of y are processed successfully, it goes back to x and accepts.

Consider finally point 3. This is done as is point 2 above with the following differences. The set of nodes $f(x)$ is a singleton and the node x does not have to be marked anymore as it can be recover from the position of pebble k. Moreover, instead of checking equality of two $(k-1)$-number the automaton simulates addition with 1. □

Using this lemma the coding of alternating Turing machines is rather straightforward.

Theorem 4.2. *Let $k \geq 1$. The emptiness problem (and hence the inclusion problem) for* wDPA_k *(and hence for* PA_k*) is* k-EXPTIME*-hard.*

In the case of *alternating* pebble tree automata, it is possible to reduce the number of pebbles used in Lemma 4.1 by one. Indeed in the case $k = 1$, the second item (and therefore also the third one) can be performed by non-deterministically move to a node y of $f(x)$ and then split into n parallel computations that bitwise check that the subtrees of x and y are the same: the j^{th} computation walks to the j^{th} bit of the subtree of y, stores the bit, walks to the marked node x and then to the j^{th} bit of the subtree of x. Therefore the emptiness problem for altrernating pebble tree automata using k pebbles is $(k+1)$-EXPTIME-hard. A matching upper-bound has been obtained in [8] and therefore this problem is $(k+1)$-EXPTIME-complete.

5 Discussion

It is not too hard to see that when restricted to strings, the techniques developed in this paper imply:

Theorem 5.1. *For $k \geq 1$ (the case $k = 0$ is equivalent to the case $k = 1$).*

1. *The emptiness and inclusion problems for* wDPA_k *over strings are* $(k-1)$-EXPSPACE-*hard.*
2. *The emptiness and inclusion problems for* PA_k *over strings are in* $(k-1)$-EXPSPACE.

Over trees our result and the one of [4] show that both the weak model and the strong model of pebble have the same expressive power and the same complexities. We believe that pebble tree automata using k strong pebbles are $(k-1)$-exponentially more succinct than pebble tree automata using k weak pebbles. It would be interesting to settle this issue.

Acknowledgment. We thanks Joost Engelfriet for his comments and in particular pointing to us the extension to the alternating case.

References

1. A. V. Aho, J. D. Ullman. Translations on a Context-Free Grammar. In *Information and Control*, 19(5): 439-475, 1971.
2. M. Bojańczyk and T. Colcombet. Tree-Walking Automata Cannot Be Determinized. In *Theor. Comput. Sci.*, 350(2-3): 164-173, 2006.
3. M. Bojańczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. In *STOC*, 2005.
4. M. Bojańczyk, M. Samuelides, T. Schwentick, L. Segoufin. Expressive power of pebble automata. In *ICALP*, 2006.
5. H. Comon et al. Tree Automata Techniques and Applications. Available at http://www.grappa.univ-lille3.fr/tata
6. S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis, M.Y. Vardi. Decidable Optimization Problems for Database Logic Programs. In *STOC*, 1988.
7. J. Engelfriet and H.J. Hoogeboom. Tree-walking pebble automata. In *Jewels are forever*, (J. Karhumäki et al., eds.), Springer-Verlag, 72-83, 1999.

8. J. Engelfriet. The complexity of typechecking tree-walking tree transducers. Technical Report 2008-01, Leiden Institute of Advanced Computer Science, Leiden University, 2008.
9. J. Engelfriet and H.J. Hoogeboom. Nested Pebbles and Transitive Closure. In *STACS*, 2006.
10. J. Engelfriet, H.J. Hoogeboom, J.-P.Van Best. Trips on Trees. In *Acta Cybern.* 14(1): 51-64, 1999.
11. J. Engelfriet, S. Maneth. A comparison of pebble tree transducers with macro tree transducers. In *Acta Inf.* 39(9): 613-698, 2003.
12. N. Globerman, D. Harel Complexity Results for Two-Way and Multi-Pebble Automata and their Logics. In *Theor. Comput. Sci.*, 169(2): 161-184, 1996.
13. T. Milo, D. Suciu and V. Vianu. Typechecking for XML transformers. In *J. Comput. Syst. Sci.*, 66(1): 66-97, 2003.
14. A. Muscholl, M. Samuelides and L. Segoufin. Complementing deterministic tree-walking automata. In *IPL*, 99(1): 33-39, 2006.
15. F. Neven. Extensions of Attribute Grammars for Structured Documents Queries. In *DBPL*, 1999.
16. M.Y. Vardi. A note on the reduction of two-way automata to one-way automata. In *IPL*, 30: 261-264, 1989.

Appendix: Missing proofs

Lemma 3.4. *Let $(\mathcal{A}, \mathcal{B})$ be a* wPABU$_1$. *For any marking m, we can construct in* EXPTIME $\mathcal{C} \in$ BU* *such that $\mathcal{C}$ simulates exactly $(\mathcal{A}, \mathcal{B})$ on trees marked by m.*

Proof. (sketch)

Given a tree t and a node x of t, we denote by t_x the subtree of t rooted at x. Let $*$ be a new symbol not in Σ. A **context** is a tree over $\Sigma \cup (\Sigma \times \{*\})$, where the label with $*$ occurs only once at a leaf. This unique leaf whose label contains $*$ is called the **port** of the context. We denote by $\Gamma_{t,x}$ the context resulting from t by removing all proper descendants of x and adding $*$ to the label of x.

Let $\mathcal{B}$ be a bottom-up automaton, t a tree and Γ a context. The evaluation of $\mathcal{B}$ on t is the set of all states reached by $\mathcal{B}$ at the root of t. The transition relation of $\mathcal{B}$ on Γ is the set of pairs of states (q, q') such that $\mathcal{B}$ associates the state q' to the root of Γ if the port of Γ is labeled by state q.

Let a be a letter of Σ, δ the transition relation of $\mathcal{B}$, $Q' \subseteq Q_\mathcal{B}$ be a subset of states of $\mathcal{B}$ and $R \subseteq Q_\mathcal{B} \times Q_\mathcal{B}$, we denote by $Compo(a, R, Q')$ the set of pairs

$$\{(q_1, q_2) \in Q_\mathcal{B} \times Q_\mathcal{B} | \exists q_3 \in Q', \exists q_4 \in \delta(a, q_1, q_3) \text{ and } (q_4, q_2) \in R\}$$

and by $Compo(Q', a, R)$ the set of pairs

$$\{(q_1, q_2) \in Q_\mathcal{B} \times Q_\mathcal{B} | \exists q_3 \in Q', \exists q_4 \in \delta(a, q_3, q_1) \text{ and } (q_4, q_2) \in R\}$$

Remark: Let t be a tree, x an internal node of t labeled by the letter a, moreover, let x_1 and x_2 be respectively the left and right child of x. If Q' is the evaluation of $\mathcal{B}$ on t_{x_2} and R the transition relation of $\mathcal{B}$ on $\Gamma_{t,x}$ then $Compo(a, R, Q')$ is the transition relation of $\mathcal{B}$ on Γ_{t,x_1} and if Q' is the evaluation of $\mathcal{B}$ on t_{x_1} and R the transition relation of $\mathcal{B}$ on $\Gamma_{t,x}$ then $Compo(Q', a, R)$ is the transition relation of $\mathcal{B}$ on Γ_{t,x_2}.

Let $P \subseteq \{1, \cdots, k\}$ be a pebble set. A **P-pebbled tree** is a tree t with an associated P-pebbled assignment. A **pebbled tree** is a P-pebbled tree, for some P. We usually do not explicitly denote f. Analogous notions are defined for contexts. Given a tree t and a node x of t, we denote by $t_x^\bullet$ the $\{1\}$-pebbled tree t_x with the pebble on x and $\Gamma_{t,x}^\bullet$ the $\{1\}$-pebbled context $\Gamma_{t,x}$ with the pebble on x.

Let $(\mathcal{A}, \mathcal{B})$ be a wPABU$_1$ and t a tree.

A 0-configuration of $\mathcal{A}$ is a tuple (x, q, x_1) where $q \in Q_\mathcal{A}$ is the current state, x is the current node and x_1 the position of the pebble. An 1-configuration is a pair (x, q) where $q \in Q_\mathcal{A}$ is the current state and x is the current node. For $i \in \{0, 1\}$ an i-run is a run from an i-configuration to an i-configuration in which pebble $i + 1$ is never lifted. An i-loop is an i-run from a configuration (x, p, f) to a configuration (x, q, f). Therefore an i-loop is determined by the source configuration and the target state q. An i-loop is an i-tree-loop if it involves no i-configuration outside t_x, it is an i-context-loop if it involves no i-configuration outside $\Gamma_{t,x}$.

Let t be a tree marked by m and u its m-node. The automaton $\mathcal{C}$ that we construct in this proof will guess a position v in t and simulate all runs of $\mathcal{A}$ from u to v. To this end it will maintain in its state the expected position of v and all the loops of $\mathcal{A}$ in the subtree below the current position. It will also guess all the loops of $\mathcal{A}$ in the context around the current node. Even though those loops are guessed the automaton can always make sure the guesses are correct while moving up the tree. In the end, at the root of the tree, the automaton will then be able to have exactly all the runs of $\mathcal{A}$ from u to v.

More formally, when at a node x, $\mathcal{C}$ contains in its state the following information:

- the expected (or known) positions of u and v relative to x,
- the evaluation $q(x)$ of $\mathcal{B}$ on t_x,
- the evaluation $q'(x)$ of $\mathcal{B}$ on $t_x^\bullet$,
- the set $\tau(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 0-tree-loops of $\mathcal{A}$ in t_x,
- the set $\tau'(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 0-tree-loops of $\mathcal{A}$ in $t_x^\bullet$,
- the set $R(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 1-tree-loops of $(\mathcal{A}, \mathcal{B})$ in t_x.

Furthermore, at x, $\mathcal{C}$ also guesses and stores in its state, the following information:

- the type of x and the expected positions of u and v relative to x.
- the set $c(x) \subseteq Q_\mathcal{B} \times Q_\mathcal{B}$ of the transition relation of $\mathcal{B}$ on the context $\Gamma_{t,x}$,
- the set $\gamma(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 0-context loops of $\mathcal{A}$ in $\Gamma_{t,x}$,
- the set $\gamma'(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 0-context loops of $\mathcal{A}$ in $\Gamma_{t,x}^\bullet$,
- the set $S(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ of all the 1-context loops of $(\mathcal{A}, \mathcal{B})$ in $\Gamma_{t,x}$.

All the guessed information above, except for the position of v is going to be verified as correct later up in the tree. At the root, $\mathcal{C}$ will have chosen a position for v and will be able to verify that all other choices were correctly guessed.

Using $R(x)$ and $S(x)$, C computes and maintains in its state the set $T(x) \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ such that:

- if u and v are not descendant of x, $T(x)$ is empty
- if u is a descendant of x and v is not a descendant of x, $T(x)$ is the set of 1-runs of $(\mathcal{A}, \mathcal{B})$ from u to x.
- if v is a descendant of x and if u is not a descendant of x, $T(x)$, is the set of the 1-runs of $(\mathcal{A}, \mathcal{B})$ from x to v.
- if u and v are both descendants of x, $T(x)$ is the set of the 1-runs from u to v.

Thus at the root of t, the restriction of $T(x)$ to $Q_\mathcal{A} \times Q_{f_\mathcal{A}}$ gives the desired result.

The automaton $\mathcal{C}$ computes all the relations above bottom-up from the leaves to the root as follows. The case where x is a leaf is completely obvious. If x is an internal node, let a be its label and let x_1 and x_2 be respectively the left and right child of x. Assuming the relations have been computed for x_1 and x_2, $\mathcal{C}$ computes the ones of x as follows.

- $q(x)$ and $q'(x)$ are computed by simulating $\mathcal{B}$.
- $c(x)$ is guessed and $\mathcal{C}$ checks that it is consistent with the previous guessed values of c and q, for example, $\mathcal{C}$ checks that $c(x_1) = Compo(a, c(x), q(x_2))$
- $\tau(x)$ is computed easily from the current label, $\tau(x_1)$, and $\tau(x_2)$. $\tau'(x)$ is computed similarly.
- $\gamma(x)$ is guessed and $\mathcal{C}$ checks that $\gamma(x)$ is consistent with the 0-tree-loops and 0-context-loops computed at x_1 and x_2. Thus at the root $\mathcal{C}$ can make sure all guesses where appropriate. $\gamma'(x)$ is computed similarly.
- Similarly, $R(x)$ is computed by combining (i) 1-tree-loops of $R(x_1)$ (ii) 1-tree-loops of $R(x_2)$ and (iii) a drop on x that depends on $q'(x)$ and $c(x)$ followed by a sequence of 0-loops in $\tau'(x) \cup \gamma'(x)$ and a lift.
- $S(x)$ is guessed and $\mathcal{C}$ checks that it is consistent with $S(x_1)$, $S(x_2)$ and all other relations. For example, $\mathcal{C}$ checks that $S(x_1)$ corresponds to the combination of (i) context 1-loops of $S(x)$ (ii) tree 1-loops of $R(x_2)$ (iii) a drop on x_1 that depends on $q'(x_1)$ and $c(x_1)$ followed by a sequence of 0-loops in $\tau'(x_1) \cup \gamma'(x_1)$ and a lift.
- $T(x)$ is easy to maintain using the above relations once the positions of u and v relative to x are known.

It is clear that the size of $\mathcal{C}$ is exponential in $(\mathcal{A}, \mathcal{B})$ and it can be verified that it has the indented behavior. □

Theorem 4.2. *Let $k \geq 1$. The emptiness problem (and hence the inclusion problem) for* wDPA_k *is* k-EXPTIME*-hard.*

Proof. For DTWA, and hence wDPA_1, EXPTIME-hardness was already implicit in [15] (it can be also obtained by adapting the proof below). Therefore we now assume $k \geq 2$.

We simulate an alternating Turing machine using space $\exp(k-1, n)$ on input of size n with a wDPA_k. This implies the theorem.

More precisely given an alternating Turing machine $\mathcal{M}$ running in space $\exp(k-1, n)$ on an input w of size n, we construct $A_{\mathcal{M}} \in \text{wDPA}_k$ such that the trees accepted by $A_{\mathcal{M}}$ encode accepting runs of $\mathcal{M}$. A configuration of $\mathcal{M}$ is encoded as a k-number. That is each bit of the k-number codes the bit of the same position in the tape. The position of the head and the current state of the configuration are coded by extending slightly (with no harm) the notion of k-number by adding the label q to the bit of the position where the head is.

A run of $\mathcal{M}$ is encoded by concatenating the configurations for existential moves and using the branching structure of the tree for universal moves.

We now construct $\mathcal{A}_{\mathcal{M}}$ accepting exactly the accepting runs of $\mathcal{M}$ on input w. First $\mathcal{A}_{\mathcal{M}}$ checks that the tree has the right form: (i) it is built as a combination of k-number with only one position containing a state (ii) configurations containing an existential state are followed by an unique configuration, (iii) configurations containing an universal state are followed by two configurations, and (iv) all configurations with no successor are accepting and the initial configuration codes w. (i) can be done with $k-1$ pebbles as shown in Lemma 4.1, (ii), (iii), and (iv) are immediate.

Once this is done it remains to check that two successive configurations are valid transitions of $\mathcal{M}$. This is done as follows. $\mathcal{A}_{\mathcal{M}}$ considers successively each configuration by doing a complete traversal of the tree. For each configuration it does the following. It successively marks each position of the tape by dropping pebble k on the corresponding $(k-1)$-number. Let f be the function that maps x to the set of positions of the configuration occurring above x. This function is determined by a DTWA which goes up in the tree until it sees the symbol $\sharp$ and then successively traverses the tree upward in state q_S until it sees the next $\sharp$ symbol. Then it uses the $k-1$ remaining pebbles to find the matching position in the preceding configuration by using part 2 of Lemma 4.1. It then checks that the bits pointed by this two $(k-1)$-number are equal if none contains a state or behave according to a transition of $\mathcal{M}$ otherwise. □

Theorem 5.1. *For $k \geq 1$ (the case $k = 0$ is equivalent to the case $k = 1$).*

1. *The emptiness and inclusion problems for* wDPA_k *over strings are* $(k-1)$-EXPSPACE*-hard.*
2. *The emptiness and inclusion problems for* PA_k *over strings are in* $(k-1)$-EXPSPACE.

Proof. The lower bound is immediate from Theorem 4.2. Indeed the notion of k-number can also be encoded as a string corresponding to an appropriate sequence of pairs (bit,position). However the tree branching structure was necessary in order to encode the alternating behavior of a Turing machine. Without the branches we can only encode non-deterministic behavior. Hence the lower bound.

For the upper-bound we proceed as follows. Assume $k > 1$. Let $\mathcal{A}$ be in PA_k. By iterating on Lemma 3.5 and Lemma 3.4 as in the proof of Lemma 3.6, one can construct in $(k-1)$EXPTIME $\mathcal{C} \in \mathrm{PABU}_1^*$ equivalent to $\mathcal{A}$. Note that in the string case, $\mathcal{C}$ is nothing else but a PA_1. To conclude the proof it is therefore sufficient to prove that emptiness of $\mathcal{C}$ can be checked in PSPACE. This is exactly the case $k = 1$.

Assume now that $k = 1$. By Lemma 3.5 we can assume wlog that $\mathcal{A}$ is in wPA_1. Consider now the proof of Lemma 3.4 constructing from $\mathcal{A}$ a one-way string automaton $\mathcal{C}$ equivalent to $\mathcal{A}$. Instead of constructing $\mathcal{C}$ explicitly and then checking for the existence of a path from the initial state to an accepting state, we simulate $\mathcal{C}$ on the fly, storing the current state of $\mathcal{C}$ in memory, and computing one of its successors on the fly. Each state of $\mathcal{C}$ having a size polynomial in $\mathcal{A}$, this can be done in NPSPACE =PSPACE. □