Kybernetika

Jan Janoušek; Bořivoj Melichar; Martin Poliak

Tree compression pushdown automaton

Kybernetika, Vol. 48 (2012), No. 3, 429--452

Persistent URL: http://dml.cz/dmlcz/142947

KYBERNETIKA — VOLUME *48* (2012), NUMBER 3, PAGES 429–452

TREE COMPRESSION PUSHDOWN AUTOMATON

Martin Poliak, Jan Janoušek and Bořivoj Melichar

A new kind of a deterministic pushdown automaton, called a *Tree Compression Automaton*, is presented. The tree compression automaton represents a complete compressed index of a set of trees for subtrees and accepts all subtrees of given trees. The algorithm for constructing our pushdown automaton is incremental. For a single tree with n nodes, the automaton has at most $n + 1$ states, its transition function cardinality is at most $4n$ and there are $2n + 1$ pushdown store symbols. If hashing is used for storing automaton's transitions, thus removing a factor of $log\ n$, the construction of the automaton takes linear time and space with respect to the length n of the input tree(s). Our pushdown automaton construction can also be used for finding all subtree repeats without augmenting the overall complexity.

Keywords: trees, pushdown automata, compression, indexing trees, arbology

Classification: 05C05, 68Q68

1. INTRODUCTION

A new kind of a deterministic pushdown automaton, called a *Tree Compression Automaton* (TCA), is presented in this paper. The tree compression automaton represents a complete and compressed index of a set of trees for subtrees and accepts all subtrees of the given trees. The paper is one from the series of arbology papers, which represent a unifying approach to tree algorithms [4, 6, 8] whose model of computation is the standard string pushdown automaton reading trees in a linear notation [10].

The TCA is proved to be deterministic and suitable for tree compression. A decompression algorithm is provided. Our result corresponds to a related result [5], which describes a technique for grammar compression of trees, where the result of the compression is a context–free grammar. Our TCA accepts the same language as is generated by that grammar, and works in a bottom–up way with the basic principle that each nonterminal symbol of the slightly modified grammar corresponds to one pushdown symbol of TCA. In comparison with [5], we further show some other properties and possible applications of the TCA, such as its use for an efficient finding of subtree repeats in the tree.

With TCA as the tree index using hashing for accessing values of the transition function, an occurrence of subtree can be searched in time $\mathcal{O}(m)$, where m is the length of the linear notation of the subtree.

Another interesting property of the TCA is that it can be used for searching of a

subtree in a given tree with n nodes in time $\mathcal{O}(n)$, similarly to the pushdown automaton [10], which does not perform any compression.

Section 2 provides definitions of notions used throughout the paper. The definitions are based on [10] for greater compatibility with our previous arbology results.

Section 3 presents the *Tree Compression Automaton* (*TCA*) and provides formal proofs of its properties. An example of a TCA is shown.

A tree compression and decompression algorithm is presented in the fourth Section.

The fifth Section shows how the construction algorithm for TCA can be exploited for finding exact repeats in a tree and for building a tree repeats index. A formal proof of the correctness of the algorithm is given. The time and space complexity of the algorithm is established.

2. BASIC NOTIONS

The definitions of the basic notions are extracted from [10]. The following theoretical concepts are introduced: alphabet, language, context–free grammar, pushdown automaton, graph, tree, tree bar notation.

2.1. Alphabet, language, context–free grammar, pushdown automaton

Notions from the theory of string languages are defined similarly as they are defined in [2].

An *alphabet* is a nonempty finite set of symbols. A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Expression $\mathcal{A}^*$ stands for the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly, for string $x \in \mathcal{A}^*$, notation x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$ and $x^+ = x^* \setminus \{\varepsilon\} = \{x^m : m \geq 1\}$. Given a string x, $|x|$ denotes the length of x.

A *context-free grammar* (CFG) is a 4-tuple $G = (N, \mathcal{A}, P, S)$, where N and $\mathcal{A}$ are finite disjoint sets of *nonterminal* and *terminal symbols*, respectively. P is a finite set of *rules* of the form $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup \mathcal{A})^*$. $S \in N$ is the *start symbol*. Relation $\Rightarrow$ is called *derivation*: if $\alpha A \gamma \Rightarrow \alpha\beta\gamma$, $A \in N$, and α, β, $\gamma \in (N \cup \mathcal{A})^*$, then rule $A \to \beta$ is in P. Symbols $\Rightarrow^+$, and $\Rightarrow^*$ are used for the *transitive* closure, and the *transitive and reflexive* closure of $\Rightarrow$, respectively. The language generated by a G, denoted by $L(G)$, is the set of strings $L(G) = \{w : S \Rightarrow^* w, w \in \mathcal{A}^*\}$.

An (extended) *nondeterministic pushdown automaton* (nondeterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, G is a *pushdown store alphabet*, δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, and $F \subseteq Q$ is the set of final (accepting) states. Triple $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. In this paper we will write the top of the pushdown store x on its left hand side. The initial configuration of a pushdown automaton is a triple (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$.

The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $\delta(q, a, \alpha) \ni (p, \gamma)$. A transition $\vdash_M \subset (Q \times \varnothing \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is called an *ε-transition*. The kth power, transitive

closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively. A pushdown automaton M is a *deterministic* pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A} \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, a, \beta) \neq \varnothing$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, \varepsilon, \beta) \neq \varnothing$, then α is not a suffix of β and β is not a suffix of α.

A language L accepted by a pushdown automaton M is for the purposes of this article defined by:

1. *Accepting by empty pushdown store:*

$$L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}.$$

When PDA accepts the language by empty pushdown store, the set F of final states is the empty set.

2.2. Graph, tree, prefix notation, bar notation

Notions on trees are defined similarly as they are defined in [2, 8, 9] and [10].

Based on concepts from graph theory (see [2]), a labelled, ordered tree over an alphabet $\mathcal{A}$ can be defined as follows:

An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of linearly ordered lists of edges such that each element of R is of the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. This element will indicate that, for node f, there are n edges leaving f, the first entering node g_1, the second entering node g_2, and so forth.

A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \ldots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. A *labelling* of an ordered graph $G = (A, R)$ is a mapping of A into a set of labels. We use a_f for a short declaration of node f labelled by symbol a.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in A$. By analogy, the *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in A$.

A *labelled, ordered and rooted tree* t over a ranked alphabet $\mathcal{A}$ is an ordered dag $t = (N, R)$ with a special node $r \in A$, called the *root*, such that

(1) r has in-degree 0,

(2) all other nodes of t have in-degree 1,

(3) there is just one path from the root r to every $f \in N$, where $f \neq r$,

(4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$.

Nodes with out-degree 0 are called *leaves*.

The subtree definition is inspired by [8]. A tree s with root r_s is a *subtree* of a tree t with root r_t if:

1. There exists a path p in the tree t in form $(r_t, \ldots, r_s)$. Path p' is a copy of path p without the last node r_s.
2. A sequence of nodes formed by concatenating path p' with a path from node r_s to any node n of tree s is a path that exists in tree t.
3. Let $k = length(p)$. Any path q in tree t of $length(q) > k$ that has as its prefix path p can be created by concatenation of path p' with a path from root r_s to a node n of tree s.

Path p is called a *position path* of subtree s in tree t.

A tree s with root r_s is a *child subtree* of tree t with root r_t if s is its subtree and its *position path* in t has length 1.

Two trees t, t' are identical if their roots r, r' are labeled with the same label, the roots have the same number k of child subtrees s_i, s'_i for $1 \leq i \leq k$ and every two child subtrees s_i, s'_i for $1 \leq i \leq k$ are identical.

The *prefix bar notation* $pref\_bar(t)$ of tree t is defined as follows:

1. $pref\_bar(a) = a\ |$
2. $pref\_bar(t) = a\ pref\_bar(b_1)\ pref\_bar(b_2) \ldots pref\_bar(b_n)\ |$, where a is the root of tree t and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

3. TREE COMPRESSION AUTOMATON

3.1. Construction of Tree Compression Automaton

The tree compression automaton $TCA(\{t\})$ is a deterministic pushdown automaton that accepts a tree t and all of its subtrees. The structure of the TCA must satisfy certain conditions in order to accept the language in a space-efficient manner. These conditions are described in the General TCA definition and in the TCA-construction algorithm. TCA is proved to be deterministic and to accept exactly all subtrees of the given tree(s).

Example 3.1 Figure 1 shows a tree t_1 and its prefix bar notation.

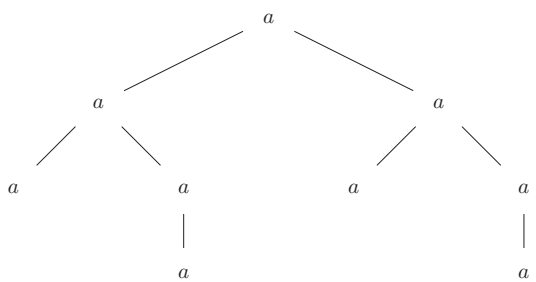

Fig. 1. Tree t_1 and its prefix bar notation.

Definition 3.2 Let T be a set of trees. Let I be a set of symbols. Let μ be an injective mapping from the set of all subtrees of all trees from set T into set I such that two subtrees are assigned the same element from set I if and only if they are identical. The triplet (T, I, μ) is called *subtree identification mapping for set* T.

Example 3.3 An example of a subtree identification mapping is shown in Figure 2.

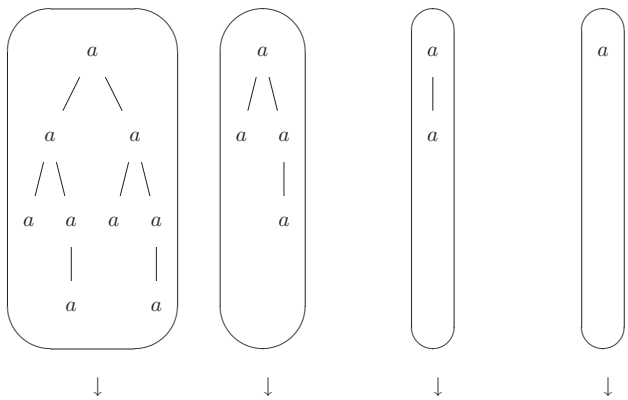

Fig. 2. Subtree identification mapping $(\{t_1\}, \{1, 2, 3, 4\}, \mu_1)$ maps every unique subtree of tree t_1 from Example 3.1 to a unique identifier.

Definition 3.4 Let T be a set of trees. Let (T, I, μ) be a subtree identification mapping for set T. Let t be any ordered subtree of any tree in set T. Let it have k child subtrees $child\_subtree_i$ for i from 1 to k in this order. Let tree t be written as a pair (r, L), where r is the root of t and L is an ordered list $(\mu(child\_subtree_1), \mu(child\_subtree_2), \ldots)$.

Given (T, I, μ), the pair (r, L) is called a *tree stub* of tree t.

Example 3.5 Tree stubs that are derived from the subtree identification mapping $(\{t_1\}, \{1, 2, 3, 4\}, \mu_1)$ from Example 3.3:

- tree stub of tree t_I, $t_I = a|$, $\mu_1(t_I) = 1$: $(a, ())$,
- tree stub of tree t_{II}, $t_{II} = aa||$, $\mu_1(t_{II}) = 2$: $(a, (1))$,
- tree stub of tree t_{III}, $t_{III} = aa|aa|||$, $\mu_1(t_{III}) = 3$: $(a, (1, 2))$,
- tree stub of tree t_{IV}, $t_{IV} = aaa|aa|||aa|aa||||$, $\mu_1(t_{IV}) = 4$: $(a, (3, 3))$.

Theorem 3.6 Let (T,I,μ) be a subtree identification mapping for a set of ordered trees T. Let t be any ordered subtree of any tree in set T. Let pair (r,L) be the tree stub of tree t. Tree t can be reconstructed from mapping (T,I,μ) and tree stub (r,L). At the same time, given mapping (T,I,μ), exactly one tree stub exists for every subtree from set of trees T.

The proof shows that the prefix bar notation of tree t can be reconstructed from the subtree identification mapping and its tree stub. Then it shows that a tree stub exists for every subtree of any of the trees from the set of trees T. Lastly, it proves that the tree stub can be only one for a given tree.

Proof. (1) Let tree t with k child subtrees be written as tree stub $(r, (\ \mu(child\_subtree_1), \mu(child\_subtree_2), \ldots))$. The prefix bar notation of t is defined as $r\ pref\_bar(child\_subtree_1)\ \ pref\_bar(child\_subtree_2)\ \ldots\ pref\_bar(child\_subtree_k)\ |$. Since mapping μ assigns the same symbol only to identical trees, it is safe to use μ to rewrite the tree stub that describes tree t into $(r,(pref\_bar(child\_subtree_1),$ $pref\_bar(child\_subtree_2), \ldots))$. This pair can then be easily transformed into $pref\_bar(t)$. Tree t can be reconstructed.

(2) Suppose that there is a subtree t' from set of trees T for which there exists no tree stub. Tree t' has a prefix bar notation and a root r. If the depth of tree t' is 0 and therefore $pref\_bar(t') = r|$, tree t' can be rewritten into a tree stub $(r, ())$, which contradicts the initial assumption and therefore this tree t' cannot exist. Suppose the depth of t' is $d+1$. The $pref\_bar(t')$ can be rewritten into a pair $(r,\ (pref\_bar(child\_subtree_1),$ $pref\_bar(child\_subtree_2), \ldots, pref\_bar(child\_subtree_k)))$. But $pref\_bar\ (child\_subtree_i)$ for i from 1 to k is a prefix bar notation of a child subtree of the tree t' that has depth at most d and for which there is a tree stub. This child subtree was assigned an element from set I. The pair can therefore be rewritten into $(r,(\mu(child\_subtree_1), \mu(child\_subtree_2),$ $\ldots, \mu(child\_subtree_k)))$, which is a tree stub of tree t'. Tree t' therefore cannot exist.

(3) There can be only one tree stub $(r,(\mu(child\_subtree_1),\ \mu(child\_subtree_2),\ \ldots,$ $\mu(child\_subtree_k)))$ for an ordered tree t. This is because the root of tree t is always the same node r, the identifier $\mu(child\_subtree_i)$ for i from 1 to k maps to a single value by definition and the order of the subtrees is only one for one tree t. □

Corollary 3.7 (extension of Theorem 3.6) Given a subtree identification mapping (T, I, μ), there exists a unique mapping between the set of identifiers I and the set of tree stubs of T.

Proof. By the previous theorem, every tree has exactly one tree stub and every tree also has exactly one identifier from the set of identifiers I. □

Example 3.8 Reconstruction of tree t_1 from Example 3.1 from the tree stubs from Example 3.5:

- $t_I = a|$, $\mu(t_I) = 1$: $(a, ()) \rightarrow a|$,
- $t_{II} = aa||$, $\mu(t_{II}) = 2$: $(a, (1)) \rightarrow (a, (a|)) \rightarrow aa||$,
- $t_{III} = aa|aa|||$, $\mu(t_{III}) = 3$: $(a, (1, 2)) \rightarrow (a, (a|, aa||)) \rightarrow aa|aa|||$,
- $t_{IV} = aaa|aa|||aa|aa||||$, $\mu(t_{IV}) = 4$: $(a, (3, 3)) \rightarrow (a, (aa|aa|||, aa|aa|||)) \rightarrow$ $aaa|aa|||aa|aa||||$.

Definition 3.9 A *General Tree Compression Automaton* for a set of trees T – *GTCA(T)* – is a pushdown automaton $M = (Q, \mathcal{A} \cup \{|, \dashv\}, \mathcal{A} \cup I \cup \{\#\}, \delta, q_0, \#, \varnothing)$. $\mathcal{A}$ is a set of labels of the nodes of the trees from set T and I is a set of symbols. Symbol $\dashv$ is a

marker symbol for the end of the input string. The automaton accepts input by an empty pushdown store. $\mathcal{A} \cap \{\#, \dashv\} = \varnothing$, $(\mathcal{A} \cup \{\#\}) \cap I = \varnothing$.

GTCA constructed for a set T of trees is defined to accept exactly all subtrees of the trees in set T in the prefix bar notation.

Definition 3.10 *Initial General Tree Compression Automaton* (*Initial GTCA*) is a $GTCA(\varnothing) = (q_0, \{|, \dashv\}, \{\#\}, \varnothing, q_0, \#, \varnothing)$.

Definition 3.11 Tree Compression Automaton – TCA
A $GTCA(T)$ is a *Tree Compression Automaton* ($TCA(T)$) if

1. $T = \varnothing$, or
2. $T = T' \cup \{t\}$ and $GTCA(T)$ is the output of the TCA-construction algorithm with input $TCA(T')$ and t.

The following algorithm describes an online algorithm extending $TCA(T)$ to create an automaton $TCA(T \cup \{t\})$. As proved later, this automaton is a $GTCA(T \cup \{t\})$.

If we consider pushdown store P used by the algorithm to be the pushdown store of automaton $TCA(T)$, then it is clear that the algorithm simulates automaton $TCA(T)$ while trying to accept input tree t. If a transition is missing in automaton $TCA(T)$, the algorithm extends the transition function to enable it. If automaton $TCA(T)$ does not accept a subtree that it should accept, the transition function is extended in step 6.

Algorithm 1 TCA-construction
Input: A tree t in prefix bar notation and an automaton $M = TCA(T)$
Output: The automaton $TCA(T \cup \{t\})$
Method:
Let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the input pushdown automaton. Let P be a pushdown store. Let q_{act} mark the current state.

1. Let the pushdown store P contain the symbol $\#$.
2. Read a (next) symbol a from $pref\_bar(t)$:
   - If $a \neq |$, then $\mathcal{A} := \mathcal{A} \cup \{a\}$, $G := G \cup \{a\}$, $\delta(q_0, a, \varepsilon) = (q_0, a)$. Push the symbol a on top of the pushdown store P. Repeat step 2.
   - If $\delta(q_0, |, \varepsilon) = (q_1, \varepsilon)$, continue with step 3. Otherwise, set $Q := Q \cup \{q_1\}$ and $\delta(q_0, |, \varepsilon) := (q_1, \varepsilon)\}$.
3. $q_{act} := q_1$.
4. Pop a symbol b from the top of the pushdown store P.
5. If $b \notin \mathcal{A}$, then:
   (a) If $\delta(q_{act}, \varepsilon, b) = (q_b, \varepsilon)$, then $q_{act} := q_b$. Otherwise, create a state q_{new}. $Q := Q \cup \{q_{new}\}$, $\delta(q_{act}, \varepsilon, b) := (q_{new}, \varepsilon)$, $q_{act} := q_{new}$.
   (b) Go back to step 4.
6. Else, if $b \in \mathcal{A}$:
   (a) If $\delta(q_{act}, \varepsilon, b) = (q_b, c)$, then push symbol c on top of the pushdown store P. Otherwise, create a new pushdown store symbol s_{new}. Set $G := G \cup \{s_{new}\}$, $\delta(q_{act}, \varepsilon, b) := (q_0, s_{new})$ and push s_{new} on the pushdown store P. $\delta(q_0, \dashv, s_{new}\#) := (q_0, \varepsilon)$.

(b) If the pushdown store P contains only $s\#$, $s \notin \mathcal{A}$, then exit and the output is the automaton $M = TCA(T \cup \{t\})$. Else go to step 2.

Theorem 3.12 The output of the TCA-construction algorithm is a deterministic pushdown automaton.

P r o o f . The input of the TCA-construction algorithm is either an Initial GTCA, which is trivially deterministic, or its own output automaton. The output automaton is shown to be a deterministic pushdown automaton if one can show that it is deterministic at every step of the algorithm. Assume that the input automaton is a deterministic pushdown automaton. The output automaton M has a certain structure that follows from the TCA-construction algorithm:

- There are three groups of "transitions" going out of the state q_0:
  1. $\delta(q_0, |, \varepsilon) = (q_1, \varepsilon)$,
  2. $\delta(q_0, a, \varepsilon) = (q_0, \varepsilon)$, $a \in (\mathcal{A} \setminus \{|, \dashv\})$,
  3. $\delta(q_0, \dashv, a\#) = (q_0, \varepsilon)$, $a \in G$.

  If the automaton M is in the state q_0, then δ is clearly a mapping for any input symbol s. The algorithm ensures that the relation δ remains a mapping by carefully checking if $\delta(q, a)$ is defined before attempting to define a value for $\delta(q, a)$.
- There are two types of "transitions" going out of all states q other than q_0:
  1. $\delta(q, \varepsilon, a) = (q_0, i)$, $a \in \mathcal{A}$, $i \in G$,
  2. $\delta(q, \varepsilon, a) = (q', \varepsilon)$, $a \notin \mathcal{A}$.

  Again, if automaton M is in state q, then it is unambiguous which group of pairs from mapping δ to choose from when looking for a transition for an input symbol a. As before, the algorithm ensures that relation δ remains a mapping.

The trivial input automaton, Initial GTCA, is deterministic. The relation δ remains a mapping throughout the algorithm. The output automaton of the TCA-construction algorithm is a deterministic pushdown automaton. □

Theorem 3.13 *(No cyclic configurations in a TCA)* Let an automaton M be the output of the TCA-construction algorithm. Let it be in configuration (q, α, β). The sequence of transitions $(q, \alpha, \beta) \vdash^+ (q, \alpha, \beta)$ is not possible.

P r o o f . There are two types of states that automaton M can be in: q_0 and the others.

- Let automaton M be in configuration (q_0, α, β). There are no ε-transitions from this configuration. If automaton M reads a symbol from the input string, it cannot get back into configuration (q_0, α, β). Note that any input symbol $a \in \mathcal{A}$ is put on the pushdown store only if it is also read from the input string.
- Let automaton M be in configuration (q, α, β), where $q \neq q_0$. While $q \neq q_0$, every transition removes an element from the pushdown store and pushes no element back. This means that even if automaton M can get from state q back to state q by a nonempty sequence of transitions, the pushdown store contents in these two configurations will be different, unless automaton M passes through state q_0.

□

Definition 3.14 Let $\delta(q, \varepsilon, a) = (q_0, i)$ be a transition of an automaton $TCA(T)$. The symbol i is called a *subtree identifier*.

It is shown later in this Section that for an automaton $TCA(T)$ there exists a subtree identification mapping (T, I, μ) such that I is a set of the subtree identifiers from automaton $TCA(T)$.

Theorem 3.15 Let t be a tree and T be a set of trees. Let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the output automaton of the TCA-construction algorithm (Algorithm 1) for input $TCA(T)$ and t. If automaton M is in the configuration $(q_0, pref\_bar(t)\alpha, \beta)$, then there exists a finite sequence of deterministic transitions such that $(q_0, pref\_bar(t)\alpha, \beta) \vdash (q_0, \alpha, i\beta)$, where i is the subtree identifier of tree t.

P r o o f. Automaton M is a deterministic automaton. See the proof of Theorem 3.12.

The TCA-construction algorithm simulates the pushdown automaton it creates. Whenever a transition is missing, the algorithm first extends the transition function to enable it and then continues simulation. This proof shows that the TCA-construction algorithm does not extend the transition function in a way that would contradict the Theorem.

State q_0 is the only state in which automaton M reads symbols from the input string.

Let $pref\_bar(t) = a|$. Automaton M can only be in configuration $(q_0, a|\alpha, \beta)$, if it starts reading $a|$ from its input string. It then takes the transition $(q_0, a|\alpha, \beta) \vdash (q_0, |\alpha, a\beta)$. Since automaton M is deterministic, the TCA-construction has to take this transition while simulating TCA for input string $a|$. Symbol a is now on the top of the pushdown store. A bar is the next input symbol, which forces the TCA-construction to take the transition $(q_0, |\alpha, a\beta) \vdash (q_1, \alpha, a\beta)$.

While in the configuration (q, α, γ), $q \neq q_0$, $\gamma = \gamma' a\beta$, $\gamma' \in G*$, automaton M has to behave deterministically independent of the input, deciding the transitions only based on the symbols from γ. The only transitions that lead automaton M back into the state q_0 are the transitions that read a symbol a, $a \in \mathcal{A}$, from the top of the pushdown store. The sequence of transitions $(q_1, \alpha, \gamma) \dashv^+ (q_x, \alpha, a\beta)$ is thus completely determined by γ down to the first symbol $a \in \mathcal{A}$. This sequence of transitions is finite because there can be no "cycle" in the sequence of configurations (see the proof of Theorem 3.13).

Automaton M makes transition from configuration to configuration until it pops a symbol $a, a \in \mathcal{A}$ from the top of the pushdown store. At that moment automaton M has performed a sequence of transitions that uniquely identifies subtree $a|$. The automaton M assigns a subtree identifier i to $a|$ and pushes i on the pushdown store. Automaton M performed the following transition: $(q_1, \alpha, a\beta) \vdash (q_0, \alpha, i\beta)$.

Let $pref\_bar(t) = a\ pref\_bar(t_1) \dots pref\_bar(t_k)|$. Let automaton M be in the configuration $(q_0, a\ pref\_bar(t_1) \dots pref\_bar(t_k)|\alpha, \beta)$ when it starts reading symbols from the input string. After pushing symbol a on top of the pushdown store, it will process $pref\_bar(t_1)$ from the input string, eventually making a transition to the configuration $(q_0, pref\_bar(t_2) \dots pref\_bar(t_k)|\alpha, i_1 a\beta)$. This is repeated for the remaining subtrees in the input string until automaton M makes a transition to the configuration $(q_0, |\alpha, i_k \dots i_1 a\beta)$. At this moment, the same reasoning as in the previous paragraph can be applied. There is exactly one finite sequence of transitions that leads automaton M from the configuration $(q_0, |\alpha, i_k \dots i_1 a\beta)$ to the configuration $(q_0, \alpha, i\beta)$. □

Theorem 3.16 Algorithm TCA-construction constructs a $GTCA(T \cup \{t\})$ for a given $TCA(T)$ and a tree t.

The proof consists of two parts. In the first part, it is shown that the output of the TCA-construction algorithm still recognizes all subtrees of the trees from T.

In the second part of the proof, it is first proved that the constructed automaton accepts all subtrees of tree t. Then it is shown that the constructed automaton does not accept anything else than subtrees from $T \cup \{t\}$.

P r o o f. Firstly, the input automaton $TCA(T)$ and the output automaton as well are pushdown automata, which directly follows from the definition of the TCA. Throughout the algorithm nothing is deleted from the input $TCA(T)$. The $TCA(T)$ is modified only through additions to its sets $Q, \mathcal{A}, G$ and extension of the mapping δ. Since nothing is deleted from the transition function, the language that the $GTCA(T \cup T')$ pushdown automaton accepts must contain the language that $TCA(T)$ accepts.

Secondly, let $pref\_bar(t) = a|$. Let $T' = \{t\}$. Let the input of the algorithm be a $TCA(T)$ automaton M and set T'. Let the contents of the pushdown store of automaton M be β. The algorithm starts simulating automaton M for input $a|\alpha$. In this case $\beta = \#, \alpha = \dashv$. The algorithm puts the node label a on the pushdown store (simulating either an existing mapping or a newly created mapping $\delta(q_0, a, \varepsilon) = (q_0, a)$). It then reads a bar from the input and has to transition to state q_1. The tree that was read is present on pushdown store P. On the top of pushdown store P is root a of a subtree, $a \in \mathcal{A}$. The automaton then performs a sequence of transitions that ends in state q_0 and stores an identifier i for the just read subtree on the pushdown store. The content of the input string is now α, the pushdown store holds $i\beta$ on the top. The transition function is extended in automaton M to accept tree t by emptying the pushdown store if it is the only tree on input. If the depth of tree t is zero, automaton M is constructed to accept it.

Any tree or subtree on input of the TCA-construction algorithm should be accepted by the constructed TCA. For input starting with $pref\_bar(t)$, the automaton makes a sequence of transitions from the configuration $(q_0, pref\_bar(t)\alpha, \beta)$ to the configuration $(q_0, \alpha, i\beta)$. The transition function is extended if necessary: $\delta(q_0, \dashv, i\#) = (q_0, \varepsilon)$. Automaton M makes a transition using δ to accept a tree t if $\alpha = \dashv$.

Let us assume that every tree t' of depth at most k that is a subtree of a tree t of depth $k+1$ is accepted by a $TCA(T \cup \{t\})$ thanks to the TCA-construction algorithm. This assumption implies that the TCA will have the subtree identifier of tree t' on top of its pushdown store after a recognized tree t' of depth at most k is read by it from the input string.

Let t denote a tree of depth $k+1$ with n child subtrees that is put on the input of $TCA(T)$ by the TCA-construction algorithm. When a bar symbol b that corresponds to the root symbol a of tree t is reached while reading the $pref\_bar(t)$, the subtree identifiers of child subtrees $subtree_j$ of tree t for $j = 1$ to $j = n$ of depth at most k present between symbol a and its bar b are already present on the pushdown store in the reverse order of appearance in the original tree, in the form of subtree identifiers. This follows from Theorem 3.15.

The pushdown store looks like this: $P = \alpha_n \alpha_{n-1} ... \alpha_1 a$, $a \in \mathcal{A}$, $\alpha_j \in Q$ for $j = n$ to $j = 1$, i. e. on and just below the top of the pushdown store are subtree identifiers

of all the subtrees that were sequentially identified after symbol a was read but before bar symbol b was reached. The subtree identifiers are stored on the pushdown store in reverse order of the subtrees of tree t.

The TCA-construction creates a sequence of new states and extends mapping δ for the sequence of subtree identifiers, if necessary. If state q is the last state of this sequence, the algorithm ensures that there will be a transition $\delta(q,\varepsilon,a)=(q_0,m)$, $a \in \mathcal{A}$, with a unique m. As there exists a transition that empties the pushdown store in the case that only $m\#$ is on it and there is nothing left on the input, the automaton M will accept tree t.

It remains to show that $TCA(T)$ does not accept anything else but the subtrees of trees from set T.

- Identifier i that is pushed on the pushdown store is always the same for the same input tree t – that is based on the determinism of the automaton and on the fact that if the automaton accepts tree t, then the sequence of transitions from the configuration $(q_0,pref\_bar(t)\alpha,\beta)$ to the configuration $(q_0,\alpha,i\beta)$ must be possible within automaton M.
- Identifier i is also unique for tree t. That is, it is not possible for automaton M to get from configuration $(q_0,\ pref\_bar(t')\alpha,\ \beta)$ to configuration $(q_0,\ \alpha,\ i\beta)$ if $pref\_bar(t) \neq pref\_bar(t')$. It is easy to see that the in-degree (number of transitions leading to the state) of every state except state q_0 of automaton M is 1. Whenever a transition is missing in automaton M (except for a transition that should lead to state q_0), the algorithm extends the transition function. The algorithm also extends the mapping $\delta(q,\varepsilon,a \in \mathcal{A}) = (q_0,m)$, if necessary, using a unique m. Therefore for two different $\delta(q_1,\varepsilon,a) = (q_0,m), \delta(q_2,\varepsilon,b) = (q_0,n)$, $a,b \in \mathcal{A}$, it holds $m \neq n$. Identifier i is unique for tree t.
- The algorithm assigns only as many subtree identifiers as necessary – based on the number of unique subtrees in input tree t that were not present in set T. Therefore if a tree t' is present in the input string of automaton M constructed by the TCA-construction algorithm and t' is not one of the subtrees of $T \cup \{t\}$, then after automaton M reads tree t' from the input string and makes a transition out from state q_0, it cannot get back into state q_0 and therefore cannot accept this different tree t'.

Automaton M is a $GTCA(T \cup \{t\})$. □

Corollary 3.17 A pushdown automaton $TCA(T)$, where T is a set of trees, is a deterministic pushdown automaton.

P r o o f. The proof follows directly from the fact that the output of the TCA-construction algorithm is a deterministic pushdown automaton. □

Theorem 3.18 Let T be a set of trees and let t be a subtree of any of the trees in T. Let i be the subtree identifier of t. Let $M = (Q, \mathcal{A} \cup \{|,\dashv\}, \mathcal{A} \cup I \cup \{\#\}, \delta, q_0, \#, \varnothing)$ be a $TCA(T)$ and there exists a subtree identification mapping (T, I, μ). If the mapping δ contains the values $\delta(q_0,|,\varepsilon) = (q_1,\varepsilon)$, $\delta(q_1,\varepsilon,C) = (q_2,\varepsilon)$, …, $\delta(q_{k-1},\varepsilon,X) = (q_k,\varepsilon)$, $\delta(q_k,\varepsilon,r) = (q_0,i)$, then $(r,(X,\ldots,C))$ is a tree stub of tree t.

P r o o f . The set of pushdown store symbols G of automaton M consists of $\mathcal{A} \cup I \cup \{\#\}$, where every element from I is a subtree identifier of some subtree of some tree from set T. As shown in the proof to the previous Theorem, every unique tree is assigned a different element from set I. Also, if two subtrees are the same, automaton M will assign them the same element from set I. This implies that automaton M defines a bijective mapping μ between the set of subtrees of the trees in set T and set I. That means that there exists a subtree identification mapping (T,I,μ).

Let t be a subtree of any of the trees in set T. There must exist the following unique sequence of transitions between configurations of automaton M:
$(q_0, pref\_bar(t), \alpha) \vdash^+ (q_0, \varepsilon, i\alpha)$. Symbol i is then the subtree identifier of tree t. It is unique because automaton M is deterministic.

Let $pref\_bar(t) = rpref\_bar(t_1) \ldots pref\_bar(t_k)|$. There must exist a transition $(q_0, pref\_bar(t), \alpha) \vdash (q_0, pref\_bar(t_1) \ \ldots \ pref\_bar(t_k)|, r\alpha)$. Then if i_1,...,i_k are the subtree identifiers of t_1,...,t_k respectively, there must exist a sequence of transitions $(q_0, pref\_bar(t), \alpha) \vdash (q_0, |, i_k \ldots i_1 r\alpha)$. Since every subtree identifier is an element of the I and there is a bijective mapping between set I and the subtrees of the trees in the T, $(r, (i_1, \ldots, i_k))$ is a tree stub of tree t, given the subtree identification mapping (T,I,μ). □

At this moment TCA is defined and it is shown to be a deterministic pushdown automaton that accepts all subtrees of a given set of trees. An example TCA for a given input is shown in the following Subsection.

3.2. TCA-construction: example

The input string of an example output of the *TCA-construction* algorithm is the prefix bar notation of tree t_1 from Example 3.1 and an Initial $TCA(\varnothing)$. The output TCA is shown on Figure 3.

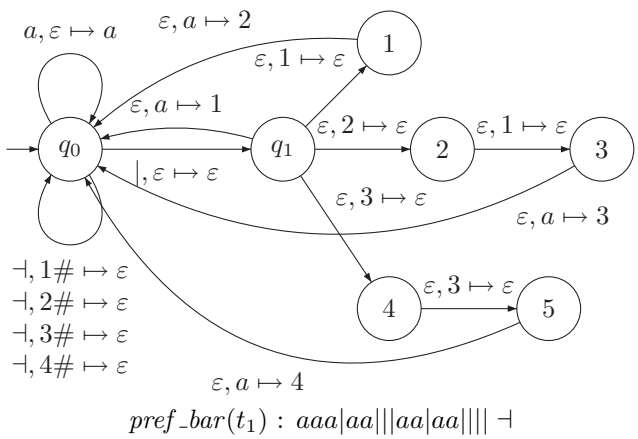

Fig. 3. TCA$\{t_1\}$ constructed by Algorithm 1.

3.3. Size of the output of the TCA-construction algorithm

Let automaton M be a $TCA(T)$, where T is a set of trees. Let t be a tree with n nodes. When the TCA-construction algorithm is executed with tree t and automaton M as input, it may add new states, transitions and pushdown store symbols to automaton M. This Subsection computes the maximum and minimum number of states, transitions and pushdown store symbols that can be added to automaton M by the TCA-construction algorithm.

There are two cycles in Algorithm 1. It is first shown how many times each cycle is executed, and then the size of the output is established.

The first cycle (step 2 of the algorithm) reads symbols from the input string. A symbol $a \neq |$ is read and pushed on the pushdown store n times in total. Every time this happens the transition function is extended and one pushdown store symbol can be added to automaton M, adding at maximum n pushdown store symbols to the pushdown store alphabet of automaton M.

Symbol $a = |$ is read n times as well. The execution then enters the second cycle that starts with step 3. In total, the first cycle is executed $2n$ times.

The second cycle is the pushdown store contents processing cycle at steps 3 through 6 in the algorithm. It is reached n times from the first cycle, whenever a bar is read.

The execution of the TCA-construction algorithm returns from the second cycle to the first cycle every time that a symbol from $\mathcal{A}$ is present on top of the pushdown store. When and only when the execution returns to the first cycle, the algorithm pushes a symbol $b \notin \mathcal{A}$ onto the pushdown store and can add symbol b to the pushdown store alphabet. Execution returns to the first cycle n times.

Exactly n symbols $a \in \mathcal{A}$ and n symbols $b \notin \mathcal{A}$ are pushed onto the pushdown store in total. At every execution of the body of the second cycle one symbol $b \notin \mathcal{A}$ is popped from the pushdown store. The algorithm leaves one symbol $b \notin \mathcal{A}$ and one symbol $\#$ on the pushdown store before exiting. Only $2n - 1$ symbols can be and are popped throughout the execution of the algorithm. Therefore the body of the second cycle is executed $2n - 1$ times.

This case of maximal output size is encountered when there are no subtree repeats in input tree T.

Theorem 3.19 Let t be a tree with n nodes. Let M be the output TCA of the TCA-construction algorithm for input consisting of tree t and an Initial GTCA. TCA M has at most $n+1$ states, $2n+1$ pushdown store symbols and the cardinality of the transition function is $4m$.

P r o o f . A new state can be added to automaton M in the body of the second cycle only when a symbol $b \notin \mathcal{A}$ is on top of the pushdown store. That happens $n-1$ times during the execution of the algorithm (the nth time that symbol b is on top of the pushdown store, the whole tree is accepted and no state is created). Only two states can be added to automaton M outside the body of the second cycle (states q_0 and q_1). This means that at maximum $n + 1$ states can be added to automaton M by the algorithm.

The pushdown store alphabet of an Initial GTCA contains one symbol, $\#$. At maximum n symbols $a \in \mathcal{A}$ can be added to the pushdown store alphabet. Symbols $b \notin \mathcal{A}$

can be added to the pushdown store alphabet every time a symbol $a \in \mathcal{A}$ is popped from the pushdown store. This happens n times. Therefore at maximum n symbols $b \notin \mathcal{A}$ can be added to the pushdown store alphabet. In total, the pushdown store alphabet has at most $2n + 1$ symbols.

The transition function is extended $\delta(q_0, a, \varepsilon) = (q_0, a)$ for every symbol $a \in \mathcal{A}$ that is read from the input. This can be at most n times. The transition function is extended $\delta(q_0, \varepsilon, b\#) = (q_0, \varepsilon)$ for every pushdown store symbol $b \notin \mathcal{A}$. This can be again at most n times. One extension of δ can be added from state q_0 to state q_1. One extension of δ can be added by step 6 for every symbol $a \in \mathcal{A}$ on top of the pushdown store. This is again at most n extensions. One extension of δ is added by the second cycle every time a new state is added. This is at most $n - 1$ times. In total, automaton M will have cardinality of δ at most $4n$. □

Trivially, the algorithm adds no new states, pushdown store symbols and extensions of δ to the $TCA(T)$ if input tree t is a subtree of any of the trees in set T.

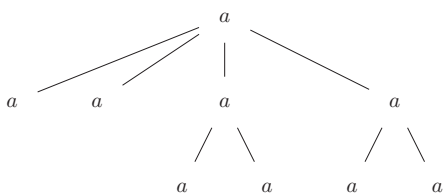

Fig. 4. Tree t_2 and its prefix bar notation from Example 3.20.

Example 3.20 Figure 4 shows a tree t_2 for which a TCA provides the best compression it is capable of.

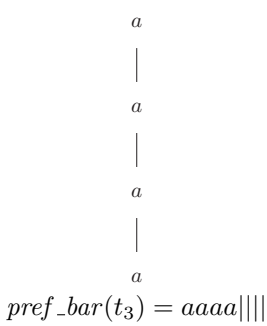

Fig. 5. Tree t_3 and its prefix bar notation from Example 3.21.

Example 3.21 Figure 5 shows a tree t_3 for which a TCA provides the worst compression it is capable of.

Theorem 3.22 Let t be a tree with n nodes. Let an automaton M be the output $TCA(\{t\})$ of the TCA-construction algorithm with input consisting of tree t in its prefix bar notation and an Initial GTCA. If $n = 3^d$, where d is the depth of tree t, $TCA(\{t\})$ M can have only $2 * log_3(n) + 2$ states, $log_3(n) + 2$ pushdown store symbols and cardinality of δ $4 * log_3(n) + 4$. In general, automaton M cannot have less than $\lceil log_6(n) \rceil + 2$ states, $\lceil log_6(n) \rceil + 2$ pushdown store symbols and $3\lceil log_6(n) \rceil + 4$ δ cardinality.

P r o o f. Let t be a tree for which the ratio $r =$ (number of states plus δ cardinality in $TCA(\{t\})$)/(number of nodes in t) is minimal.

If two subtrees of tree t have the same depth, they have to be identical. If they were not, then one could imagine a tree t' in which the occurrences of the smaller of the two subtrees are replaced by the greater one. $TCA(\{t'\})$ would not need to create states and extend δ, which would be needed to encode the smaller of the two subtrees while encoding a tree with at least as many nodes as there are in tree t.

Let t_{k+1} be a subtree of depth $k + 1$. As child subtrees, this subtree will have all the child subtrees of subtree t_k of depth k. The child subtrees will be in the same order as in subtree t_k. Such a tree requires no new states and no extension of δ in automaton M to encode the child subtrees copied from child subtree t_k. The subtree t_{k+1} will also have a number n_k of copies of subtree t_k as its leftmost child subtrees. To encode n_k child subtrees t_k in subtree t_{k+1}, the $TCA(\{t\})$ requires n_k new states and n_k δ extensions. To encode the root of subtree t_{k+1} and allow acceptance of subtree t_{k+1}, the $TCA(\{t\})$ needs 2 more δ extensions.

Let tree t have depth d. Let n_k be the number of subtrees of depth k in the subtree of depth $k + 1$. The tree t has $(n_{d-1} + 1)(n_{d-2} + 1) \ldots (n_0 + 1)$ nodes. $TCA(\{t\})$ will have $4 + 2d + (n_{d-1} + n_{d-2} + ... + n_0)$ cardinality of δ, $2 + (n_{d-1} + n_{d-2} + ... + n_0)$ states and $2 + d$ pushdown store symbols.

Number n_k cannot be higher than 5. If there were 6 child subtrees of depth k in subtree s, the subtree s would have $7 * nodes(k)$ nodes, where $nodes(k)$ is the number of nodes in the subtree of depth k. The encoding of s would require 6+2 additional δ extensions, 6 additional states and one additional pushdown store symbol in a TCA that already accepts the subtree of depth k. One could imagine a subtree s_1 with 2 child subtrees of depth k and a subtree s_2 with 2 child subtrees of depth $k + 1$. s_2 would have $3 * 3 * (nodes(k))$ nodes, but s_1 and s_2 would require only 4+4 additional δ extensions, 4 additional states and 2 additional pushdown store symbols in the TCA. Therefore s cannot be present in tree t. The same reasoning can be applied if s has 7 and 8 child subtrees of depth k.

If s has 3^2 child subtrees of depth k and therefore has $10 * (nodes(k))$ nodes, one would imagine a subtree s_3 with two child subtrees of depth $k + 2$, which again requires less space in the TCA and encodes the tree with more nodes, $27 * (nodes(k))$. Subtree s has more than $27 * (nodes(k))$ nodes if it has at least 3^3 child subtrees of depth k. Adding more $(3^3, 3^4, \ldots)$ child subtrees to s allows encoding of trees with $3^3, 3^4, \ldots$ times more nodes while requiring $3^3, 3^4, \ldots$ times more space in TCA. Adding subtrees s_i with

increasing depth ($i = k + 3, i = k + 4, \ldots$) allows encoding of trees with as many nodes while requiring only $3 * 2, 4 * 2, \ldots$ times more space in the TCA.

Ratio r is equal to $8+3d+2(n_{d-1}+n_{d-2}+...+n_0)/(n_{d-1}+1)(n_{d-2}+1)\ldots(n_0+1)$. Ratio r is minimal for n_k with values from $\{1, 2, 3, 4, 5\}$, where k ranges from 1 to $d-1$. The minimal value of ratio r therefore cannot be higher than $(8 + 3d + 2 * (d * 5))/2^d$. In fact, it cannot be higher than $(8 + 3d + 2 * (d * 2))/3^d$, the value achieved if $n_k = 2$. Ratio r also cannot be lower than $(8 + 3d + 2 * (d))/6^d$. The exact lower bound for r is not known. □

The best compression ratio is achieved for example for Fibonacci trees and complete k-ary trees.

4. TREE COMPRESSION AND DECOMPRESSION

4.1. Algorithm TCA-decompression

The output of the *TCA-construction* algorithm (Algorithm 1), a $TCA(T)$, will be shown as suitable for compression of trees that contain repeating subtrees. The compression ratio ranges from linear to logarithmic. The worst case is encountered if the compressed tree contains no repeating subtrees, as in Example 3.21. The best case is a tree that has all subtrees of the same depth identical, while keeping number of child subtrees between 2 and 6. Such trees are found in the proof to Theorem 3.22. Example 3.20 shows such a tree.

The decompression algorithm reconstructs tree t from $TCA(T)$ if t is a subtree of any of the trees in set T. The main idea of the decompression algorithm is to transform $TCA(T)$ into a straight-line grammar [5] that generates exactly one string, tree t. The decompression algorithm needs two things on the input: $TCA(T)$ and the subtree identifier i that was assigned to tree t by the TCA-construction algorithm.

The subtree identifier i of tree t has to be remembered if set $T \neq \{t\}$. Else it can be found as the only subtree identifier for which $\delta(q, \varepsilon, i) = \varnothing$ for all states $q \in Q$.

Algorithm 2 TCA-decompression
Input: Automaton $TCA(T)$. Tree t, which is a subtree of one of the trees in set T. The subtree identifier i assigned to tree t by the TCA-construction algorithm
Output: Tree t in prefix bar notation.
Method:

1. Let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the $TCA(T)$. Let q_{act} be a marked state.
2. Create a grammar $R = (N, T, P, S)$, where $N = G \setminus \mathcal{A} \setminus \{\#\}$, $T = \mathcal{A} \cup \{|\}$, $S = i$ and P is an empty set of rules.
3. Reverse the transition function in $TCA(T)$, that is, replace every $(p, v, \alpha) \to (q, \beta)$ by $(q, v, \alpha) \to (p, \beta)$.
4. For every element from the mapping δ $\delta(q_0, \varepsilon, x) = (q_y, C)$, $C \in G$, do:
   (a) Create a rule $r = C \to x$ and set $q_{act} := q_y$.
   (b) For every element D such that $\delta(q_{act}, \varepsilon, D) = (q_z, \varepsilon)$, append D to the right-hand side of rule r and set $q_{act} := q_z$. If $q_{act} = q_0$, set the rules $P := P \cup \{r\}$ and continue step 4 for the next element from mapping δ.

5. The output is the content of language $L(R)$. This language will be shown to contain exactly one string, $pref\_bar(t)$.

Theorem 4.1 Let t be a tree. Let M be a $TCA(T)$ such that tree t is a subtree of one of the trees in set T. Let i be a subtree identifier assigned to tree t by automaton M. Let w be the output of the *TCA-decompression* algorithm whose input was automaton M and subtree identifier i. Then $w = pref\_bar(t)$.

Proof. Let u be any subtree of any of the trees in set T. Let r_u be the root of tree u. Let i_u be the subtree identifier of subtree u. Let $pref\_bar(u) = r\ pref\_bar(s_1) \dots pref\_bar(s_k)|$ and let $i_1, \dots, i_k$ be the subtree identifiers of subtrees $s_1, \dots, s_k$, respectively. Then for input $< pref\_bar(u) > \alpha$, automaton M must be able to take a sequence of transitions into a configuration $(q_0, |\alpha, i_k \dots i_1 r)$. From the TCA-construction algorithm, it holds that
$(q_0, |\alpha, i_k \dots i_1 r) \vdash (q_1, \alpha, i_k \dots i_1 r) \vdash (q_2, \alpha, i_{k-1} \dots i_1 r) \vdash \dots \vdash (q_0, \alpha, i_u)$
and all states $q_i, q_j, i \neq j$ are pairwise different.

Let M' be a pushdown automaton equivalent to automaton M that has its δ mapping reversed. Directly following from the TCA-construction algorithm, $\delta(q_0, \varepsilon, x) = \{(q_y, C)\}$ exists only if C is a subtree identifier of a subtree of any of the trees in T. Also based on the TCA-construction algorithm, for every state other than q_0 in automaton M' there is exactly one outgoing transition $\delta(q, \varepsilon, x) = \{(q_y, C)\}$. Based on the previous paragraph, it must then hold that $\delta(q_0, \varepsilon, r_u) = (q_k, i_u), \delta(q_k, \varepsilon, i_1) = (q_{k-1}, \varepsilon), \dots, \delta(q_2, \varepsilon, i_k) = (q_1, \varepsilon), \delta(q_1, |, \varepsilon) = (q_0, \varepsilon)$.

The TCA-decompression algorithm thus constructs the rules r of grammar R in the form $r = i_u \rightarrow r_u i_1 \dots i_k|$ for every subtree u. If u is a subtree of depth 0, then the right-hand side of rule r is the prefix bar notation of u. If u is a subtree of depth $d+1$, then again the right-hand side of rule r is the prefix bar notation of u if the nonterminals $i_1, \dots, i_k$ are "transitively" rewritten to their right-hand sides. That is exactly how grammar R generates its language if the initial symbol is set to be the nonterminal i_u. □

4.2. Time and space complexity of compression and decompression by TCA-construction and TCA-decompression

The time and space complexity of the TCA-construction algorithm with input consisting of a $TCA(T)$ and a tree t, $length(pref\_bar(t)) = n$, depends on the implementation of the δ mapping lookup.

If the algorithm is provided with a zeroed space of size $(n+1) \times |Q|$, a lookup table can be created for mapping δ. Finding an appropriate mapping δ in such table takes constant time. The space taken by mapping δ is $(n+1) \times |Q|$.

If the algorithm does not construct a lookup table for mapping δ, then a searching algorithm must be used for finding the appropriate mapping δ. The maximum size of a set of outgoing "transitions" from a state is $n+1$ in the case of state q_0. The transition lookup time is therefore at worst $log_2(n+1)$. If each element of mapping δ takes up space $log2(n+1)$, space taken by mapping δ is at worst $4n * log_2(n+1)$.

The total time and space requirements of the TCA-construction depend directly on the implementation of mapping δ. The Section that computes the size of the output TCA showed that there were two cycles in the algorithm. The first cycle is run n times, and the second cycle is also run n times.

Every run of the body of the first cycle has to decide $\delta(q_0, a, \varepsilon)$. If a lookup table is available, then the body of the first cycle is executed in constant time. If a lookup table is not available, then one run of the body of the first cycle requires $c + log_{2+1}n$ time, where c is a constant.

One run of the body of the second cycle has to decide $\delta(q_{act}, \varepsilon, b)$ and can add a new transition to mapping δ. The rest of the body executes in constant time and can require constant space. Again depending on the implementation of mapping δ, the execution of the body of the second cycle can take either constant or $c + log_{2+1}n$ time.

Depending on the implementation of mapping δ, the TCA-construction algorithm can either require at worst $c_1 * n$ time and $(n+1) * (|Q| + c_2) \leq (n+1) * (n + c_2)$ space or at worst $2n * log_2(n+1) + c_3$ time and $4n * log_2(n+1) + c_4 n$ space. c_1, c_2, c_3, c_4 are constants of the algorithm.

If mapping δ is implemented through a lookup table of size c, the δ lookup time is constant and the transition reversal takes time n. If δ is not implemented through a lookup table, then the transition lookup time is at worst $log_2(n+1)$ and reversal of the transitions will take time at worst $(4n * log_2(n+1))$.

All operations in step 3 can be made within constant time c_1. In total, step 3a can be performed at most $n/2$ times and step 3b can be performed at most $n/2$ times. This in total requires at most $c_1 * n$ time.

The grammar $R = (N, T, P, S)$ will generate its language in time less than or equal to $c_2 * |N| * log_2|N|$. The size of the grammar will be $c_3 * |N| * log_2|N|$. The size of the generated language will be n.

Let $|TCA|$ be the size of the TCA on input. The TCA-decompression algorithm requires operational space $|TCA| + c_3 * |N| * log_2|N| + n$. If the TCA on the input of the TCA-decompression algorithm uses a lookup table to store mapping δ, the TCA-decompression algorithm requires time $(1+c_1)n + c_2 * |N| * log_2|N|$. Otherwise it requires time $(log_2(n+1) + c_1)n + c_2 * |N| * log_2|N|$. $|N|$ is guaranteed to be less than or equal to $n/2$.

4.3. Compression and decompression conclusion

When TCA is transformed into a grammar using the TCA-decompression algorithm, it is obvious that the compression method that uses the TCA-construction algorithm is similar to a basic technique for grammar compression of trees [5].

The proposed compression algorithm for trees offers a good compression ratio for trees with repeating subpatterns. It does not achieve such a good compression ratio as the comparable LZ methods [16, 17], but exchanges this drawback for an output that is easy to work with if one requires for example to search for a pattern in the compressed tree.

4.4. TCA as index of a tree

It is important to note at this moment that the $TCA(t)$ is quite naturally an index of tree t. If hashing is used for storing the transition function δ, deciding whether any tree u is a subtree of tree t is an operation that takes time at most $|u|$. Even more, finding all occurrences of tree t in some other tree v, given $TCA(t)$, is an operation that requires time $|v|$ – a result comparable with [10].

5. EXACT REPEATS BY THE TREE COMPRESSION AUTOMATON

The Tree Compression Automaton can be easily used for searching subtree repeats in tree t. For every subtree t_s of t, a list of its occurrences in tree t can be created using an extension of the *TCA-construction* algorithm.

The algorithm works by simulating the $TCA(\{t\})$ automaton. Whenever the automaton reads a non-bar character from the input string, the index of the character is remembered. This character is a root node of some subtree of tree t. After the last symbol of this subtree is read from the input string, the subtree is identified and the position of its root node is associated with the subtree identifier of the subtree.

Algorithm 3 TCA-repeats-search
Input: A tree t and $TCA(\{t\}) = (Q, \mathcal{A}, G, \delta, q_0, \#, \varepsilon)$
Output: A relation $occ \subset S \times \mathbb{N}$, $S = G \setminus (\mathcal{A} \cup \{\#\})$. $(s, i) \in occ$ only if s is a subtree identifier of a subtree t_s of tree t and t_s has a root at index i in $pref\_bar(t)$.
Method:
Let P be a pushdown store. Let i be a counter.

1. Set $i := 0$.
2. Simulate automaton M for input string $pref\_bar(t)$:
   (a) Whenever a transition $(q_0, a\alpha, \beta) \vdash (q_0, \alpha, a\beta)$ is taken, increment i. If $a \neq |$, push the value of counter i on top of pushdown store P.
   (b) Whenever a transition $(q, \alpha, a\beta) \vdash (q_0, \alpha, b\beta)$ is taken, pop number x from the top of the pushdown store and set $x \in occ(b)$.
   (c) When automaton M accepts the input string, output occ and exit.

This algorithm can be modified to accept $TCA(T)$ on the input, where t is a subtree of any of the trees in T. The size of its output is affected by the ratio $|TCA(T)|/|TCA(\{t\})|$. Its running time is not affected by this ratio if mapping δ is implemented by a lookup table.

Theorem 5.1 Relation *occ* maps the position of every subtree root to a subtree identifier. If two subtrees are the same, the indices of their roots are mapped to the same subtree identifier.

P r o o f . The TCA-repeats-search algorithm can be viewed as a modified TCA-construction algorithm that:

- pushes a pair $(a, index(a))$ on top of the pushdown store whenever a symbol $a \in \mathcal{A}, a \neq |$ is read from the input string
- pops a pair $(a, index(a))$ from the top of the pushdown store and replaces it there with a whenever the transition $(q, \alpha, a\beta) \vdash (q_0, \alpha, b\beta)$ is to be taken.

Since symbol b is the subtree identifier of the tree whose root is symbol a, $index(a) \in occ(b)$ is set for all roots a that are read from the input. This means that every subtree root is mapped to a subtree identifier.

Let t_s be a subtree of tree t that is the input of the algorithm. Let i_s be the subtree identifier of t_s. Let $TCA(\{t\})$ be in the configuration $(q_0, pref\_bar(t_s)\alpha, \beta)$. There is a sequence of transitions that $TCA(\{t\})$ can take ending in the configuration $(q_0, \alpha, i_s\beta)$. It must hold that $index(r_s) \in occ(i_s)$. If two subtrees are identical, their prefix bar notations are identical and therefore their subtree identifiers are identical. Therefore their root indices must be mapped to the same subtree identifier. □

5.1. Time and space complexity of *TCA-repeats-search*

The time complexity depends directly on the complexity of the TCA-construction algorithm. It also depends on the complexity of the simulation of the TCA automaton and on the complexity of adding an element into $occ(b)$.

The TCA automaton is constructed in linear time if a lookup table is used for mapping δ. Otherwise it is constructed in $n * log_2 n$ time. If n is the length of an input tree, the TCA automaton can be simulated in time either equal to n if a lookup table exists for the mapping δ, or $n * log_2 n$ otherwise.

The complexity of adding an element into $occ(b)$ is constant if a linked list is used for holding elements of $occ(b)$.

In total, the complexity of the TCA-repeats-search algorithm copies the complexity of the TCA-construction algorithm (linear or $\mathcal{O}(n*log_2 n)$, depending on implementation of a lookup table by the TCA construction algorithm).

The size of the output is $n * log_2 n$, which is the space required for storing pointers to the subtrees of tree t.

The total space required by the algorithm again depends on the space required by the TCA automaton – the number of repeats is bounded by n.

6. COMPARISON WITH RELATED RESULTS

As stated in the Introduction, a similar approach to tree compression was investigated in [5]. There the tree is compressed into a grammar. We show the relationship between the two methods on the example tree from Example 3.1. The grammar $G = (N = \{1, 2, 3, 4\}, T = \{a\}, P, 4)$ created by [5] that generates this tree has the following rules in P:

$$\begin{array}{lcl} 4 & \rightarrow & a33| \\ 3 & \rightarrow & a12| \\ 2 & \rightarrow & a1| \\ 1 & \rightarrow & a| \end{array}$$

When we look at the TCA created for the example tree, we see that its pushdown store symbols are the nonterminals of grammar G together with initial symbol #. The right-hand side of the rules of grammar G is preserved in the form of states and mapping δ. For example, when considering tree stub $(a, (3, 3))$, the rule $4 \rightarrow a33|$ corresponds to the

states $q_1, 4, 5$ and mapping δ that involves them: $\delta(q_0, |) = q_1$, $\delta(q_1, 3) = 4$, $\delta(4, 3) = 5$, $delta(5, a) = q_0$. It holds that all words in the set $\{w : X \in N, X \Rightarrow^* w, w \in T^*\}$ are accepted by the TCA. If the smallest grammar extension from [5] is omitted, this set are exactly the words accepted by the TCA. Figure 6 illustrates this relationship.

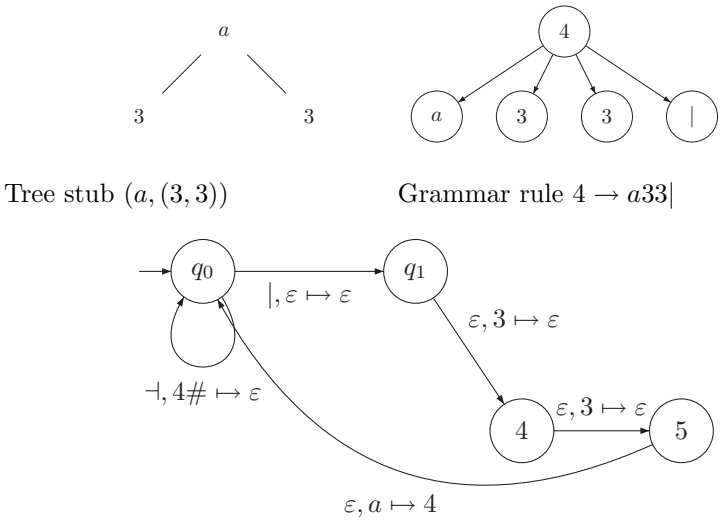

Fig. 6. A tree stub, its rule in a straight line grammar, its states and transitions in TCA.

6.1. Experimental compression results

The TCA was compressed into two separate sections: the transition table and a label lookup table. The transition table first indicates the incoming edges to each state that is not $q0$ or $q1$. Then it indicates all incoming edges to state $q0$. The label lookup table stores a mapping between node labels and internal alphabet symbols. The compressed binary file also contains a header that indicates how many bits a fixed-length binary representation of an automaton state, alphabet symbol and subtree identifier takes. The compression achieved by TCA was compared with compression of the BPLEX algorithm. As these two approaches are very close in nature, the compression ratios follow a similar curve. The compression performance was tested on pruned XML files, where text was removed from nodes. The sample data was retrieved from protein databases, linguistic records and was generated using specialized tools. The protein xml data was obtained from Swiss-Prot at UniProtKB [13] (first 1000 files) and Ligand Expo at Protein Data Bank [3] (also first 1000 files). Sample auction web-site xml data was generated by xMark [14], scaling factor was set to /f 0.1. Linguistic structures stored in xml were retrieved from the Alpino Treebank [15] (first 1000 files again). All sample xml files

Tab. 1. Compression performance compared to BPLEX. Total = total size of xml files to compress. Stripped = total size of xml files to compress, with text data removed from nodes.

	Total	Stripped	BPLEX c.r.	mod. BPLEX c.r.	TCA c.r.
Swiss-Prot	77.27M	61.05M	2.43	3.72	2.55
Ligand Expo	16.95M	12.55M	4.58	5.54	4.49
xMark	11.88M	3.71M	3.48	7.61	6.25
Alpino Treeb.	2.54M	2.06M	4.27	5.58	3.00

were stripped of text data (but not node labels) so that the tests measure solely the compression performance on the tree structures themselves.

The column BPLEX c.r. stands for compression ratio achieved by BPLEX, whereas modified BPLEX c.r. stands for compression ratio achieved by modified BPLEX algorithm when no further compression of the straight-line grammar is performed.

The compression performance of the TCA is comparable with that of BPLEX, the worse performance is some cases is expected due to the fact that TCA-compression algorithm does not perform, compared to BPLEX, further compression of the generated straight-line grammar (especially notable on the xMark benchmark). However, the TCA outperforms BPLEX when compressing small XML files found in the Alpino Treebank database, due to a better representation of the output. The same reason explains why the TCA outperforms BPLEX when compressing protein data from the Ligand Expo databank.

The TCA-construction algorithm does not apply any compression to the produced TCA, similarly as the modified BPLEX algorithm which does not to perform any further compression of the generated straight-line grammar. TCA outperforms the modified BPLEX. Due to similarities in nature between the TCA and BPLEX compression approaches, it is probable that after applying further compression to the generated automaton, TCA could perform as well as unmodified BPLEX.

7. CONCLUSION AND FUTURE WORK

A method has been proposed for compression of trees based on finding subtree repeats and using redundancy contained within such repeats. A special form of a pushdown automaton, a *Tree Compression Automaton (TCA)* was introduced. An algorithm for searching for subtree repeats based on TCA was shown. It has a linear time and space complexity wrt the number of nodes of the subject tree.

There is much more work to be done in research on TCA. It is to be shown formally that TCA can be used for searching for tree templates. Also, how efficiently it can be used for searching for approximate repeats when the Relabel operation [4] is considered is an open question.

It should be found out whether TCA can be used by efficient algorithms that search for general approximate repeats, when not only the Relabel [4] operation is considered,

but also Delete and Insert.

More information and sources related to the field of arbology can be found at [1].

ACKNOWLEDGEMENT

This research has received support from the Czech Technical University as project No. SGS12/092/OHK3/1T/18.

(Received July 6, 2011)

REFERENCES

[1] Arbology www pages: Available on `http://www.arbology.org/`, March 2012.

[2] A. V. Aho and J. D. Ullman: The Theory of Parsing, Translation, and Compiling. Prentice-Hall Englewood Cliffs, N.J. 1972.

[3] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne: The protein data bank. Nucleic Acids Res. *28* (2000), 235–242.

[4] P. Bille: Pattern Matching in Trees and Strings. Ph.D. Thesis, FIT University of Copenhagen 2008.

[5] G. Busatto, M. Lohrey, and S. Maneth: Grammar-based Tree Compression. Technical Report 2004.

[6] L. Cleophas: Tree Algorithms. Two Taxonomies and a Toolkit. Ph.D. Thesis, Technische Universiteit Eindhoven 2008.

[7] R. Cole, R. Hariharan, and P. Indyk: Tree pattern matching and subset matching in deterministic log^3-time. SODA (1999), 245–254.

[8] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (2007).

[9] F. Gecseg and M. Steinby: Tree languages. In: Handbook of Formal Languages (G. Rozenberg and A. Salomaa, eds.), Vol. 3 Beyond Words. Springer–Verlag, Berlin 1997, pp. 1–68.

[10] T. Flouri, J. Janoušek, and B. Melichar: Subtree and tree pattern pushdown automata for trees in prefix notation (2009). Comput. Sci. Inform. Systems *7* (2010), 2, 331–357.

[11] Ch. Hoffmann and M. O'Donnell: Pattern matching in trees. J. Assoc. Comput. Mach. *29* (1982), 1, 68–95.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman: Introduction to Automata Theory, languages, and Computation. Second edition. Addison-Wesley, Boston 2001.

[13] C. H. Wu, R. Apweiler, A. Bairoch, D. A. Natale, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez et al: The Universal Protein Resource (UniProt) – An Expanding Universe of Protein Information. Database issue, Nucleic Acids Research, Oxford University Press, D187–D191 2006.

[14] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse: XMark – A benchmark for XML data management. In: Proc. International Conference on Very Large Data Bases (VLDB), Hong Kong 2002, pp. 974–985.

[15] L. Van Der Beek, G. Bouma, R. Malouf, G. Van Noord, and Rijksuniversiteit Groningen: The Alpino dependency treebank. In: Computational Linguistics in the Netherlands (CLIN) 2002, pp. 1686–1691.

[16] T. Welch: A technique for high-performance data compression. Computer *17* (1984), 6, 8–19.

[17] J. Ziv and A. Lempel: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory *23* (1977), 3, 337–343.

Jan Janoušek, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Praha 6. Czech Republic
e-mail: Jan.Janousek@fit.cvut.cz

Bořivoj Melichar, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Praha 6. Czech Republic
e-mail: Borivoj.Melichar@fit.cvut.cz

Martin Poliak, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Praha 6. Czech Republic
e-mail: Martin.Poliak@fit.cvut.cz