RAIRO-Theor. Inf. Appl. 43 (2009) 779–790
DOI: 10.1051/ita/2009018

Available online at:
www.rairo-ita.org

# A PERFECT HASHING INCREMENTAL SCHEME FOR UNRANKED TREES USING PSEUDO-MINIMAL AUTOMATA

Rafael C. Carrasco[1] and Jan Daciuk[2]

**Abstract.** We describe a technique that maps unranked trees to arbitrary hash codes using a bottom-up deterministic tree automaton (DTA). In contrast to other hashing techniques based on automata, our procedure builds a pseudo-minimal DTA for this purpose. A pseudo-minimal automaton may be larger than the minimal one accepting the same language but, in turn, it contains proper elements (states or transitions which are unique) for every input accepted by the automaton. Therefore, pseudo-minimal DTA are a suitable structure to implement stable hashing schemes, that is, schemes where the output for every key can be determined prior to the automaton construction. We provide incremental procedures to build the pseudo-minimal DTA and the mapping that associates an integer value to every transition that will be used to compute the hash codes. This incremental construction allows for the incorporation of new trees and their hash codes without the need to rebuild the whole DTA from scratch.

**Mathematics Subject Classification.** 37E25.

## 1. Introduction

In many applications, there is a need to associate every piece of data with an integer which is in turn used as an identifier to access efficiently related information stored in another structure. This number or *hash code* is computed from a part of the data (often called *key*) that holds enough information to identify the whole

*Keywords and phrases.* Perfect hashing, deterministic tree automata, pseudo-minimal automata, incremental automata.

[1] Departamento de Lenguajes y Sistemas Informáticos, Universidad de Alicante, 03071 Alicante, Spain; carrasco@dlsi.ua.es

[2] Knowledge Engineering Department, Gdańsk University of Technology, Poland; jandac@eti.pg.gda.pl

Article published by EDP Sciences

piece. As *hashing* [10] transforms the key into an integer in a limited range, the domain of the key is often orders of magnitude larger than the allowed range and different keys may occasionally be mapped to the same hash code. Such situation is called a *conflict* or *collision*. However, for some static sets of keys [24], there exist collision-free hashing schemes. A function that implements such conflict-free mapping is called a *perfect hash* function. If, additionally, the function maps $n$ keys into a consecutive range of $n$ integers (*e.g.*, from 1 to $n$), it is called a *minimal perfect hash* function.

In particular, if keys are strings, an acyclic deterministic finite-state automaton (or acyclic DFA for short) can be used to implement minimal perfect hashing [17,21]. Using minimal DFA for this purpose provides both a very compact representation for the keys, and a very fast way to access the data; however, hash codes cannot be modified arbitrarily and they change every time strings are inserted or removed from the set of keys.

In contrast, *pseudo-minimal automata* – first introduced by Revuz [21] as a side-effect of a failed attempt at incremental construction of minimal acyclic DFA [2,13,22,25] – can implement an arbitrary mapping from strings to integers (not limited, for instance, to the mapping implied by the lexicographical order of key strings). This potential was discovered by Maurel [19] and has been applied to *dynamic perfect hashing* [14], where new keys may be continually added (and deleted) while the original mapping is preserved, that is, renumbering of keys is not allowed.

A traditional approach to build a minimal automaton recognizing a set of elements is to use a standard method to construct a non-minimal automaton, and then minimize it. The disadvantage of this procedure is that the size of the intermediate non-minimal automaton can be large, as well as the space and time needed to compute the minimized automaton. To alleviate these problems, incremental methods have been developed so that one element is added to the language of the minimal automaton, and then the automaton is minimized again. Since the addition modifies only a small fraction of the automaton, the minimization can be local, acting only on those parts that have changed. The incremental construction of minimal DFA has been addressed in a number of papers [1,5,8,11,13,22]. Procedures for the incremental construction of pseudo-minimal DFA have also been described before [15].

Trees, which are ubiquitous in computer science theory and applications, are another common source of keys for hashing. For instance, they are often used in the implementation of indexes in databases [3]. They also appear as the result of parsing code and whenever structured information, such as that contained in XML documents and in configuration files, is processed. In natural language processing, trees are used to store annotated corpora [18] and to represent syntactic constraints or connectivity of words in grammars [16].

Trees can be processed by deterministic tree automata. Every deterministic tree automaton (DTA) recognizes a tree language with a single-pass procedure, *i.e.*, no backtracking is needed. In contrast to top-down (also called root-to-frontier) deterministic tree automata, bottom-up (also called frontier-to-root) deterministic

tree automata can recognize all finite tree languages [9] and, thus, they are more suitable for the implementation of perfect hashing on trees.

In this paper, we show how pseudo-minimal DTA can be used to map trees to arbitrary hash codes. Relevant definitions are presented in Section 2 and pseudo-minimal DTA are introduced in Section 3. Section 4 shows how proper transitions in a pseudo-minimal DTA can be identified and how they can be used to associate a unique hash code for every tree in the recognized language. Incremental procedures to build the pseudo-minimal DTA and the associated hash codes are described in Section 5. Finally, conclusions are presented in Section 6.

## 2. Bottom-up deterministic tree automata

For every finite alphabet $\Sigma$, the language of *unranked ordered trees* $T_\Sigma$ can be defined as follows:

1. Each symbol $\sigma \in \Sigma$ is a tree in $T_\Sigma$.
2. Every $t = \sigma(t_1 \cdots t_m)$, such that $\sigma \in \Sigma$, $m > 0$ and $t_1, \ldots, t_m$ are in $T_\Sigma$, is also a tree in $T_\Sigma$.

The *valence* of a tree $\sigma(t_1 \ldots t_m)$ is $m$ and the valence of $\sigma \in \Sigma$ is 0.

Any subset of $T_\Sigma$ is called a *tree language*. In particular, the language of subtrees of $t \in T_\Sigma$ is defined as:

$$\operatorname{sub}(t) = \begin{cases} \{\sigma\} & \text{if } t = \sigma \in \Sigma \\ \{t\} \cup \bigcup_{k=1}^{m} \operatorname{sub}(t_k) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma. \end{cases} \tag{2.1}$$

The number of *occurrences* of a subtree $u$ in a tree $t$ is given by

$$\operatorname{matches}(u, t) = \begin{cases} 1 & \text{if } t = u \\ 0 & \text{if } t \in \Sigma \wedge t \neq u \\ \sum_{k=1}^{m} \operatorname{matches}(s_k, t) & \text{if } t = \sigma(s_1 \cdots s_m) \neq u. \end{cases} \tag{2.2}$$

A *finite-state tree automaton* [9] is defined as $A = (Q, \Sigma, \Delta, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of symbols called the *alphabet*, $\Delta \subset \bigcup_{m=0}^{\infty} \Sigma \times Q^{m+1}$ is a finite set of *transitions*, and $F \subseteq Q$ is a set of *acceptance states*. This definition differs from that of automata operating on strings in two remarkable aspects: first, there is no start state and, second, a transition is a relation between an alphabet symbol and an arbitrary number of states (not necessarily two). A finite-state tree automaton is *bottom-up deterministic* (and, in the following, will be simply denoted as DTA) if for all $m \geq 0$ and for all $(\sigma, i_1, \ldots, i_m) \in \Sigma \times Q^m$, there is at most one $j \in Q$ such that $(\sigma, i_1, \ldots, i_m, j) \in \Delta$. For every transition $\tau = (\sigma, i_1, \ldots, i_m, j)$, the states $i_1, \ldots, i_m$ are the *source* or *input states* of $\tau$ and state $j$ its *target* or *output*.

For every DTA, one can define a collection of transition functions $\delta_m : \Sigma \times Q^m \to Q$ as follows:

$$\delta_m(\sigma, i_1, \ldots, i_m) = \begin{cases} j & \text{if } \exists j \in Q : (\sigma, i_1, \ldots, i_m, j) \in \Delta \\ \perp & \text{otherwise} \end{cases} \tag{2.3}$$

where $\perp \in Q - F$ is the special *absorption state*, a non-acceptance state which is input or output of no transition in $\Delta$.

The result of the operation of the DTA $A$ on a tree $t$ is denoted as $A(t)$ and defined recursively:

$$A(t) = \begin{cases} \delta_0(\sigma) & \text{if } t = \sigma \in \Sigma \\ \delta_m(\sigma, A(t_1), \ldots, A(t_m)) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma. \end{cases} \tag{2.4}$$

The language of *visited states* when $A$ operates on $t$ is $\text{visited}(A, t) = \{A(s) : s \in \text{sub}(t)\}$; the *language accepted at state* $q$ is the set of trees such that $A(t)$ returns $q$,

$$\mathcal{L}_A(q) = \{t \in T_\Sigma : A(t) = q\}; \tag{2.5}$$

and the *language accepted by the automaton* $A$ is the sum of the languages accepted at its acceptance states:

$$L(A) = \bigcup_{q \in F} \mathcal{L}_A(q). \tag{2.6}$$

A state $q$ such that $\mathcal{L}_A(q) = \emptyset$ is *unreachable*. Unreachable states can be safely removed from the automaton[1] and, in the following, all DTA will be assumed to contain no unreachable states, that is, $|\mathcal{L}_A(q)| > 0$ for all $q \in Q$.

The languages $\mathcal{L}_A(q)$ play the analogous role of the left languages in a DFA [23]: in both cases, $q$ is the output obtained when the automaton operates on the trees or strings in the language and, for different states $p$ and $q$, the languages $\mathcal{L}_A(p)$ and $\mathcal{L}_A(q)$ are disjoint. It is also possible to define the DTA analogue of the right languages in a DFA, $\mathcal{R}_A(q)$, as follows:

$$\mathcal{R}_A(q) = \{t \in T_\Sigma^\odot : \exists s \in \mathcal{L}_A(q) : t \cdot s \in L(A)\} \tag{2.7}$$

where $T_\Sigma^\odot \subset T_{\Sigma \cup \{\odot\}}$ is the language of trees $t$ such that $\text{matches}(\odot, t) = 1$, and $t \cdot s$ represents the tree in $T_\Sigma$ that is obtained after replacing the subtree $\odot$ with the subtree $s$[2]. A state $q$ such that $\mathcal{R}_A(q) = \emptyset$ is *useless*. In particular, all unreachable states are useless.

The *fanin set* of a state $q \in Q$ is the subset of transitions in $\Delta$ with output $q$:

$$\text{fanin}(q) = \{(\sigma, i_1, \ldots, i_m, j) \in \Delta : j = q\} \tag{2.8}$$

and its *fanout set* contains one element of the type $(\tau, k)$ for every occurrence of $q$ as input state (in transition $\tau$ at position $k$) in $\Delta$:

$$\text{fanout}(q) = \{(\sigma, i_1, \ldots, i_m, j, k) \in \Delta \times \mathbb{N} : 0 < k \leq m \wedge i_k = q\}. \tag{2.9}$$

[1] In a DTA without unreachable states, the absorption state remains in $Q$, as $\mathcal{L}_A(\perp) = T_\Sigma$.

[2] The symbol $\odot$ marks the leaf (a valence-0 subtree) where subtree attachment takes place and the replacement of this "pointed border node" [20] with a subtree can be seen as a generalization of string concatenation.

## 3. Pseudo-minimal tree automata

A DTA is *minimal* if it is smaller (when comparing its number of states) than any other DTA accepting the same language [9]. There is only one minimal DTA (up to isomorphisms) accepting the language $L(A)$ and it can be obtained by merging pairs of equivalent states in $A$ [6,9]. Two states $p$ and $q$ in $A$ are *equivalent* ($p \equiv q$) if

$$\mathcal{R}_A(p) = \mathcal{R}_A(q).$$

This equivalence relation is a *congruence* [4], that is, all pairs of equivalent states $(p, q)$ consist of two acceptance or two non-acceptance states which are interchangeable as input states at any transition:

1. $p \in F \ \leftrightarrow q \in F$ and
2. for all $m > 0$, for all $k \leq m$ and for all $(\sigma, i_1, \ldots, i_m) \in \Sigma \times Q^m$

$$\delta_m(\sigma, i_1, \ldots, i_{k-1}, p, i_{k+1}, \ldots, i_m) \equiv \delta_m(\sigma, i_1, \ldots, i_{k-1}, q, i_{k+1}, \ldots, i_m). \tag{3.1}$$

In a minimal DTA $A_{\min}$, every equivalence class contains a single state, *i.e.*, all pairs of states are inequivalent; moreover, the languages $\mathcal{R}_{A_{\min}}$ satisfy $|\mathcal{R}_{A_{\min}}(q)| > 0$ for all states $q \neq \perp$.

We will call a DTA $A = (Q, \Sigma, \Delta, F)$ *proper* if $Q$ contains no unreachable states and for all $q \in Q$

$$|\mathcal{L}_A(q)| \leq 1 \ \vee \ |\mathcal{R}_A(q)| \leq 1. \tag{3.2}$$

As will be seen later, a proper DTA $A$ has the useful property that every tree $t$ accepted by $A$ can be associated with at least one proper element in the DTA (the element being a transition in $\Delta$ or an acceptance state in $F$). Each proper element is associated to a single tree in $L(A)$ and can be used to store related information, *e.g.*, an arbitrary number to implement perfect hashing.

A proper DTA must be *acyclic* in the following sense: $A(t) = A(s)$ and $s \in \mathrm{sub}(t)$ implies $s = t$ or $A(t) \equiv \perp$. This can be easily proved by *reductio ad absurdum*. Indeed, if there are $t$ and $s \in \mathrm{sub}(t)$ such that $s \neq t$ and $A(t) = A(s) = q$, then $|\mathcal{L}_A(q)| > 1$ as it contains, at least, two different trees ($s$ and $t$). Furthermore, there exists at least one $u \in T_\Sigma^{\odot}$ such that $t = u \cdot s$ (with $u \neq \odot$) and, then, for all $v \in \mathcal{R}_A(q)$ (which is empty only if $q \equiv \perp$), $v \cdot u \neq v$ is also in $\mathcal{R}_A(q)$. Therefore, $A$ cannot be proper because $|\mathcal{R}_A(q)| > 1$ and $|\mathcal{L}_A(q)| > 1$ simultaneously.

As a consequence, trees in the language $L(A)$ accepted by a proper DTA cannot be deeper than $|Q| - 1$ and, thus, $L(A)$ is finite. It is also possible to define a *topological order* for the states in $A$, so that $p \leq q$ if there exist $t \in L(A)$ and $s \in \mathrm{sub}(t)$ such that $A(s) = p$ and $A(t) = q$.

A proper DTA $A$ is *pseudo-minimal* if it is smaller than any other proper DTA accepting $L(A)$. For every proper DTA $A$ there is a unique (up to isomorphisms) pseudo-minimal automaton accepting $L(A)$ and it can be obtained by merging pairs of pseudo-equivalentstates in $A$. Two states $p$ and $q$ in a proper DTA are

*pseudo-equivalent* ($p \simeq q$) if they are equivalent in $A$ and $\mathcal{R}_A(p) = \mathcal{R}_A(q)$ contains at most one tree, that is,

$$p \simeq q \iff \mathcal{R}_A(p) = \mathcal{R}_A(q) \ \wedge \ |\mathcal{R}_A(p)| \leq 1. \tag{3.3}$$

The size restriction is necessary to fulfill condition (3.2): recall that in a DTA with no unreachable states, $|\mathcal{L}_A(q)| > 0$ for all states and, thus, merging two states $p$ and $q$ with identical languages $\mathcal{R}_A(q)$ would always lead to a new state $r$ whose language $\mathcal{L}_A(r) = \mathcal{L}_A(p) \cup \mathcal{L}_A(q)$ contains more than one tree. Pseudo-equivalence is a refinement of equivalence and, thus, also a congruence and an equivalence relation; moreover, in a pseudo-minimal automaton, there is only one state in each equivalence class.

Of course, a pseudo-minimal DTA will in general be larger than the equivalent minimal DTA. For instance, let $A$ with $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Delta = \{(a, q_1), (b, q_2), (a, q_1, q_1, q_3), (a, q_1, q_2, q_3), (a, q_2, q_1, q_3), (a, q_2, q_2, q_3)\}$ and $F = \{q_3\}$. This DTA is proper, because the languages $\mathcal{L}_A(q_1) = \{a\}$, $\mathcal{L}_A(q_2) = \{b\}$ and $\mathcal{R}_A(q_3) = \{\odot\}$ contain a single tree. However, the minimal equivalent DTA $A_{\min}$ with $Q_{\min} = \{q_1, q_3\}$, $\Delta_{\min} = \{(a, q_1), (b, q_1), (a, q_1, q_1, q_3)\}$ and $F_{\min} = \{q_3\}$ is not proper because both $\mathcal{L}_{A_{\min}}(q_1) = \{a, b\}$ and $\mathcal{R}_{A_{\min}}(q_1) = \{a(\odot a), a(\odot b), a(a\odot),$
$a(b\odot)\}$ contain more than one tree. Indeed, $q_1$ and $q_2$ are not pseudo-equivalent in $A$ as

$$\mathcal{R}_A(q_1) = \mathcal{R}_A(q_2) = \{a(\odot a), a(\odot b), a(a\odot), a(b\odot)\}$$

does not fulfill the second requirement in (3.3).

## 4. Identification of proper transitions

A state $q \in F$ is said to be a *proper state* in the DTA $A$ for $t \in L(A)$ if $\mathcal{L}_A(q) = \{t\}$. Thus, removing the proper state $q$ from $F$ just removes $t$ from $L(A)$. A transition $(\sigma, i_1, \ldots, i_m, j) \in \Delta$ is said to be a *proper transition* in the DTA $A$ for $t \in L(A)$ if $|\mathcal{R}_A(j)| = 1$, for all $k \leq m$ $|\mathcal{L}_A(j)| = 1$ and $t = u \cdot \sigma(s_1 \cdots s_m)$ with $\mathcal{R}_A(j) = \{u\}$ and $\mathcal{L}_A(i_k) = \{s_k\}$. Clearly, removing $(\sigma, i_1, \ldots, i_m, j)$ from $\Delta$ only removes $t$ from $L(A)$.

Note that a proper transition for $t$ is used at most once when $A$ operates on $t$. For instance, if $\Delta = \{(a, q_1), (b, q_1), (a, q_1, q_1)\}$ then, $(a, q_1)$ is not a proper transition for $a(aa)$, even if $(a, q_1)$ is only used when $A$ operates on $a(aa)$, because $\mathcal{R}_A(q_1) = \{a(\odot a), a(a(\odot)\}$. This is a convenient feature when hash codes are defined as the addition of values associated to the transitions used when the DTA operates on the tree: if the proper transition could be used more than once, it would require an associated fractional value in order to sum up to the predefined integer hash code.

For the sake of simplicity, we will assume in the following that all proper DTA contain a single acceptance state ($|F| = 1$). Note that any DTA $A$ can be easily transformed into a DTA $A' = (Q', \Sigma', \Delta', F')$ with a single acceptance state $q_\$$

accepting $L(A') = \{\$(t) : t \in L(A)\}$, where \$ is a super-root symbol not in $\Sigma$ (the analogous to the end-of-string marker in DFA) and $Q' = Q \cup \{q_\$\}$, $\Sigma' = \Sigma \cup \{\$\}$, $\Delta' = \Delta \cup \{(\$, q, q_\$) : q \in F\}$ and $F' = \{q_\$\}$. Clearly, $\mathcal{R}_{A'}(q_\$) = \{\odot\}$ and $A'(\$(t)) = q_\$$ if and only if $t \in L(A)$.

Every proper DTA $A$ with a single accepting state $q_f$ trivially satisfies $|\mathcal{R}_A(q_f)| = \{\odot\}$. On the one hand, from its definition (2.7), $\odot \in \mathcal{R}_A(q_f)$. On the other hand, if $|L(A)| > 1$, then $|\mathcal{R}_A(q_f)| = 1$; otherwise, there is $s$ such that $\mathcal{L}_A(q_f) = \{s\}$ and, then, $\mathcal{R}_A(q_f)$ cannot contain $t \neq \odot$ because $t \cdot s \notin L(A)$.

Under the assumption that the proper DTA $A$ has $|F| = 1$, one can select a proper transition in $A$ for every $t \in L(A)$, which can be found with a recursive traversal of $t$. Indeed, for every $t = \sigma(s_1 \cdots s_m) \in L(A)$, $|\mathcal{R}(A(t))| = 1$. If $m = 0$ or $|\mathcal{R}_A(A(s_k))| > 1$ (and, thus, from Eq. (3.2), $\mathcal{L}_A(A(s_k)) = \{s_k\}$) for all $k \leq m$ then $(\sigma, A(s_1), \ldots, A(s_m), A(t))$ is a proper transition for $t$; otherwise, there is $k \leq m$ such that $|\mathcal{R}_A(A(s_k))| = 1$ and one can search recursively in $s_k$ until a proper transition for $t$ is found. In case a leaf (that is, a valence-0 subtree) $\sigma$ is reached with $|\mathcal{R}_A(A(\sigma))| = 1$, then $(\sigma, A(\sigma))$ is proper for $t$.[3]

For all proper DTA that contain no other useless states than the absorption state $\perp$, the predicate $|\mathcal{R}_A(q)| \leq 1$ used to identify proper transitions can be computed, based on fanout sets (2.9), as follows:

$$|\mathcal{R}_A(q)| \leq 1 \equiv \begin{cases} \text{true} & \text{if } \text{fanout}(q) = \emptyset \\ \text{false} & \text{if } |\text{fanout}(q)| > 1 \\ \text{false} & \text{if } |\text{fanout}(q)| = 1 \wedge q \in F \\ |\mathcal{R}_A(j)| \leq 1 \wedge & \\ \forall_{n \neq k} |\mathcal{L}_A(i_n)| \leq 1 & \text{if } \text{fanout}(q) = \{(\sigma, i_1, \ldots, i_m, j, k)\} \wedge q \notin F \end{cases} \tag{4.1}$$

where $|\mathcal{L}_A(q)| \leq 1$ is also easily computed, based on fanin sets (2.8), as follows:

$$|\mathcal{L}_A(q)| \leq 1 \equiv \begin{cases} \text{false} & \text{if } |\text{fanin}(q)| \neq 1 \\ \text{true} & \text{if } \text{fanin}(q) = \{(\sigma, q)\} \\ \forall_k |\mathcal{L}_A(i_k) \leq 1| & \text{if } \text{fanin}(q) = \{(\sigma, i_1, \ldots, i_m, q)\}. \end{cases} \tag{4.2}$$

This procedure can be used to select a proper transition for every $t \in L(A)$ and to create a mapping $\eta : \Delta \to \mathbb{N}$ between transitions and integer values, as shown in algorithm 1, which assigns a hash code $h_A(t)$ to every tree $t \in L(A)$ as follows:

$$h_A(t) = \begin{cases} \eta(\sigma, A(s_1), \ldots, A(s_m), A(t)) + \sum_{k=1}^{m} h_A(s_k) & \text{if } t = \sigma(s_1 \cdots s_m) \\ \eta(\sigma, A(\sigma)) & \text{if } t = \sigma. \end{cases} \tag{4.3}$$

A DTA $A = (Q, \Sigma, \Delta, F, \eta)$ equipped with such a mapping $\eta$ will be called a *hash DTA*, or HDTA for short. If, additionally, the hash DTA satisfies (3.2), we will call $A$ a *proper HDTA*, and, if its pseudo-minimal, *pseudo-minimal hash DTA*.

[3] In such case, the leaf $\sigma$ is unique in $L(A)$ for $t$.

**Algorithm 1** `SetHash`$(A, \sigma(s_1 \cdots s_m), n)$

**Input:** a proper HDTA $A$, a tree $t = \sigma(s_1 \cdots s_m)$ such that $h_A(t) = 0$, and an integer $n > 0$.
**Output:** a proper HDTA $A'$ such that $h_{A'}(u) = h_A(u)$ for all $u \neq t$ and $h_{A'}(t) = n$.

```
for k = 1 to m do
  if |R_A(A(s_k))| ≤ 1 then
    return SetHash(A, s_k, n)
  end if
end for
η(σ, A(s_1), ..., A(s_m), A(t)) ← n
return A
```

## 5. Incremental hashing

Carrasco *et al.* [7] describe an incremental procedure to build a minimal DTA $A'$ which, given a minimal DTA $A$ and a tree $t \notin L(A)$, accepts $L(A) \cup \{t\}$. This section describes how that algorithm for the incremental construction of minimal DTA can be adapted to manipulate pseudo-minimal HDTA and to maintain simultaneously the mapping $\eta$ between transitions and integer values generating predefined hash codes.

**Algorithm 2** `AddTree`$(A, t, n)$

**Input:** a pseudo-minimal HDTA $A$, a tree $t$ and an integer $n > 0$.
**Output:** a pseudo-minimal HDTA $A'$ with $L(A') = L(A) \cup \{t\}$ and $h_{A'}(u) = h_A(u)$ for all $u \in L(A)$ and $h_{A'}(t) = n$.

```
A† ← Split(A, t)
F† ← F ∪ {A†(t)}
A' ← Pseudominim(A†, t)
return SetHash(A', t, n)
```

The basic procedure to obtain a pseudo-minimal HDTA $A'$ which accepts a tree $t$ in addition to the language accepted by the input HDTA $A$, and assigns $t$ a predefined hash code $n$, is shown in Algorithm 2. First, it creates the Cartesian product HDTA from the HDTA $A$ and the DTA accepting $\{t\}$, that is, it builds a HDTA $A^\dagger$ such that $L(A^\dagger) = L(A)$ and $\mathcal{L}_{A^\dagger}(A^\dagger(t)) = \{t\}$. Then, $A^\dagger$ is modified to accept $t$ by incorporating the state $A^\dagger(t)$ to the subset of acceptance states. Later, a local pseudo-minimization is performed where those states in $A^\dagger$ with transitions modified with respect to those in $A$ are considered as candidates to be pseudo-equivalent to other states. Finally, a proper transition for $t$ is selected and the mapping $\eta'$ between transitions and integers is updated.

**Algorithm 3** $\texttt{Split}(A, \sigma(s_1 \cdots s_m))$

**Input:** a proper HDTA $A$ and a tree $\sigma(s_1 \cdots s_m)$.
**Output:** a proper HDTA $A^\dagger$ with $L(A^\dagger) = L(A)$, $\mathcal{L}_{A^\dagger}(A^\dagger(t)) = \{t\}$, and $h_{A^\dagger} = h_A$.

1: **for** $k \leftarrow 1 \dots m$ **do**
2: $A \leftarrow \texttt{Split}(A, s_k)$
3: $q_k \leftarrow A(s_k)$
4: **end for**
5: $q \leftarrow \delta_m(\sigma, q_1, \dots, q_m)$
6: **if** $\eta((\sigma, q_1, \dots, q_m, q) \neq 0$ **then**
7: Let $(\sigma, i_1, \dots, i_n, j, k)$ be the only element in $\texttt{fanout}(q)$.
8: $\eta(\sigma, i_1, \dots, i_n, j) \leftarrow \eta(\sigma, q_1, \dots, q_m, q)$; $\eta(\sigma, q_1, \dots, q_m, q) \leftarrow 0$
9: **end if**
10: **if** $|\texttt{fanin}(q)| \neq 1$ **then**
11: Add a new state $p$ to $Q$
12: **if** $|\texttt{fanin}(q)| > 1$ **then**
13: **if** $q \in F$ **then**
14: Add $p$ to $F$
15: **else**
16: Let $(\sigma, i_1, \dots, i_n, j, k)$ be the only element in $\texttt{fanout}(q)$.
17: $\Delta \leftarrow \Delta \cup \{(\sigma, i_1, \dots, i_{k-1}, p, i_k, \dots, i_n, j)\}$
18: **end if**
19: **end if**
20: $\Delta \leftarrow \Delta \cup \{(\sigma, q_1, \dots, q_m, p)\} - \{(\sigma, q_1, \dots, q_m, q)\}$
21: **end if**
22: **return** $A$

The product HDTA $A^\dagger$ is computed by function `Split`, shown[4] in Algorithm 3, which performs a recursive traversal of the input tree $t$ and, as will be justified later, also updates the mapping $\eta$ (lines 6–9). This function guarantees that – see proposition 1.3 in [7] –, for every $s \in \text{sub}(t)$, $\mathcal{L}_{A^\dagger}(A(s)) = \{s\}$ . Therefore, all states $q \in Q^\dagger$ visited when $A^\dagger$ operates on $t$ satisfy $|\mathcal{L}_{A^\dagger}(q)| = 1$.

In contrast, unvisited states $q$ – see proposition 1.4 in [7] – satisfy $\mathcal{L}_{A^\dagger}(q) \subseteq \mathcal{L}_A(q)$. However, if $q \neq \perp$ is not visited when $A^\dagger$ operates on $t$ then $A^\dagger(v) = q$ implies $v \notin \text{sub}(t)$ and $\mathcal{R}_{A^\dagger}(q)$ cannot contain $u$ such that $u \cdot v = t$. As $L(A^\dagger) = L(A) \cup \{t\}$, then $\mathcal{R}_{A^\dagger}(q) = \mathcal{R}_A(q)$. Therefore, $|\mathcal{L}_{A^\dagger}(q)| \leq |\mathcal{L}_A(q)|$ and $|\mathcal{R}_{A^\dagger}(q)| \leq |\mathcal{R}_A(q)|$ for all states $q \in Q^\dagger$ and the output HDTA $A^\dagger$ remains proper if $A$ is proper.

The local pseudo-minimization is performed by function `Pseudominim` shown in Algorithm 4. The only difference with respect to local minimization is that, according to equation (3.3), it does not merge a visited state $n$ if $|\mathcal{R}_A(n)| > 1$

[4] Note that this algorithm is simpler than the one presented in [7]. Here, lines 13 to 18 replace the cloning procedure specified in that algorithm because in a proper HDTA the condition $|\text{fanin}(q)| > 1$ implies that $|\text{fanout}(q)| \leq 1$ and $q \in F \leftrightarrow \text{fanout}(q) = \emptyset$.

and, thus, the size of $\mathcal{L}_A(n)$ will remain 1. In contrast, a visited state $n$ such that $|\mathcal{R}_A(n)| = 1$ can be merged with a pseudo-equivalent state but, as merging does not modify $\mathcal{R}_A(n)$, its size will remain 1. Therefore, the output $A'$ is a pseudo-minimal HDTA.

**Algorithm 4** `Pseudominim`$(A, t)$

```
1: Create a list Ω containing visited(A, t) in topological order.
2: R ← Q − visited(A, t)
3: while Ω ≠ ∅ do
4:    Pop the first state q in Ω.
5:    if there is p ∈ R : q ≃ p then
6:       Let (σ, q_1, ..., q_m, q) be the unique transition in fanin(q)
7:       Δ ← Δ − {(σ, q_1, ..., q_m, q)} ∪ {(σ, q_1, ..., q_m, p)}
8:       Remove q from Q, F and Δ.
9:       η(σ, q_1, ..., q_m, p) ← η(σ, q_1, ..., q_m, q)
10:   else
11:      R ← R ∪ {q}
12:   end if
13: end while
14: return A
```

The pseudo-minimization requires using an agenda $\Omega$ with the sequence of states (in topological order and without duplicates) visited when the HDTA operates on the tree. For efficiency, both $\Omega$ and the register $R$ containing unvisited states can be computed during the recursive traversal performed by function `Split`. The equivalence test in line 5 of algorithm 4 is non-recursive (and has, thus, a straightforward implementation), that is, transitions in fanout$(q)$ and fanout$(n)$ must be identical when $q$ and $n$ are exchanged.

After pseudo-minimization, the HDTA $A'$ obtained using the incremental construction algorithm 2 accepts $L(A') = L(A) \cup \{t\}$, remains proper, and is pseudo-minimal. The method can be trivially modified [7] to deal with the removal of trees, the only difference being that line 2 in function `AddTree` reads $F^\dagger = F - \{A^\dagger(t)\}$.

Algorithm 1 modifies the mapping $\eta$ so that the new HDTA $A'$ accepting $L(A) \cup \{t\}$ outputs the required hash code for $t$. Trees not in $L(A)$ will be assumed to output a null hash code (indicating no hash code was found) and, consistently, the required hash codes will be strictly positive integers.

If $\tau_u = (\sigma, i_1, \ldots, i_m, j)$ is the proper transition in $A$ storing the value $\eta(\tau_u)$ associated to a tree $u \in L(A)$, it will remain proper for $u$ after posterior incremental additions leading to a new HDTA $A'$ provided that $i_1, \ldots, i_m$ and $j$ remain unvisited after every new addition. However, if $\tau_u$ includes visited states as arguments, it may be removed from $\Delta$ (at line 20 in function `Split`) or become *shared* with the added tree $t$, in the sense that $|\mathcal{R}_{A^\dagger}(j)|$ becomes larger than one once $A^\dagger(t)$ is added to $F^\dagger$ and, then, $h_{A'}(t) > 0$ before `SetHash` is called.

Both situations can be addressed by selecting a proper transition for $u$ higher in the tree (see lines 6–9 in algorithm 3). This updating can be repeated recursively

until a new transition for $u$ is found which is not shared with $t$. It is also worth to be remarked that:

- The addition of new transitions at line 17 only takes place if $|\mathcal{L}_A(q)| > 1$ and, therefore, the only transition $\tau_q \in \Delta$ containing $q$ as input cannot be a proper transition.
- Pseudo-minimization does not modify the languages $\mathcal{R}_{A^\dagger}$, so a transition $\tau_u = (\sigma, i_1, \ldots, i_m, j)$ which is proper for $u \neq t$ will remain proper for $u$ if it remains in $\Delta$. If its output $j$ is merged with a pseudo-equivalent state $p$, then $(\sigma, i_1, \ldots, i_m, p)$ becomes a proper transition for $u$ (line 9 in algorithm 4). In contrast, source states $i_1, \ldots, i_m$ cannot be merged as $|\mathcal{R}_{A^\dagger}(i_k)| > 1$.

In conclusion, the mapping $\eta$ can be also efficiently updated with some local operations during the addition of a new tree $t$ to the HDTA without the need to reprocess all trees in the language.

## 6. Conclusions

We have presented a method to implement perfect hashing on unranked trees using bottom-up deterministic tree automata. Instead of using minimal DTA – as in [12] –, this method uses a class of DTA, called pseudo-minimal hash DTA which, at the cost of a possible size increase, can implement arbitrary hashing. The algorithm presented in [7] has been adapted for the incremental construction of pseudo-minimal hash DTA and, based on this construction, a procedure to identify the proper elements and their associated codes has been devised.

*Acknowledgements.* This research was partially supported by the Spanish CICyT through grants TIN2006-15071-C03-01 and TIN2009-14009-C02-01. The authors thank M.L. Forcada for his useful suggestions.

## References

[1] J. Aoe, K. Morimoto and M. Hase, An algorithm for compressing common suffixes used in trie structures. *Trans. IEICE*, J75-D-II (1992) 770–779

[2] J. Aoe, K. Morimoto and M. Hase, An algorithm for compressing common suffixes used in trie structures. *Systems and Computers in Japan* **24** (1993) 31–42. Translated from Trans. IEICE, J75-D-II (1992) 770–779.

[3] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica* **1** (1972) 173–189.

[4] W.S. Brainerd, The minimalization of tree automata. *Information and Control* **13** (1968) 484–491.

[5] R.C. Carrasco and M.L. Forcada, Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics* **28** (2002) 207–216.

[6] R.C. Carrasco, J. Daciuk and M.L. Forcada, An implementation of deterministic tree automata minimization, edited by J.Holub and J. Zdárek, *CIAA2007, 12th International Conference on Implementation and Application of Automata Proceedings. Lect. Notes Comput. Sci.* **4783** (2007) 122–129.

[7] R.C. Carrasco, J. Daciuk and M.L. Forcada, Incremental construction of minimal tree automata. *Algorithmica* **55** (2009) 95–110.
[8] M. Ciura and S. Deorowicz, How to squeeze a lexicon. *Software – Practice and Experience* **31** (2001) 1077–1090.
[9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi, (1997). Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata. release October, 1st 2002.
[10] Z.J. Czech, G. Havas and B.S. Majewski, Perfect hashing. *Theoret. Comput. Sci.* **182** (1997) 1–143.
[11] J. Daciuk, Comments on incremental construction and maintenance of minimal finite-state automata by C. Rafael Carrasco and L. Mikel Forcada. *Computational Linguistics* **30** (2004) 227–235.
[12] J. Daciuk, Perfect hashing tree automata, edited by T. Hanneforth and K.-M.Würzner, *Proceedings of Finite-State Methods and Natural Language Processing: 6th International Workshop, FSMNLP 2007*, 97–106, Potsdam, September 14–16 (2007).
[13] J. Daciuk, S. Mihov, B.W. Watson and R.E. Watson, Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics* **26** (2000) 3–16.
[14] J. Daciuk, D. Maurel and A. Savary, Dynamic perfect hashing with finite-state automata, edited by M.A. Kłopotek, S. Wierzchoń and K. Trojanowski, *Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM'05 Conference held in Gdańsk, Poland, June 13-16 (2005). Advances in Soft Computing* **31** (2005) 169–178.
[15] J. Daciuk, D. Maurel and A. Savary, Incremental and semi-incremental construction of pseudo-minimal automata, edited by J. Farre, I. Litovsky and S. Schmitz, *Implementation and Application of Automata: 10th International Conference, CIAA 2005. Lect. Notes Comput. Sci.* **3845** (2006) 341–342.
[16] C. Doran, D. Egedi, B.A. Hockey, B. Srinivas and M. Zaidel, XTAG system – a wide coverage grammar for english. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING 94)*, Vol. II, Kyoto, Japan (1994) 922–928.
[17] C. Lucchiesi and T. Kowaltowski, Applications of finite automata representing large vocabularies. *Software – Practice and Experience* **23** (1993) 15–30.
[18] M.P. Marcus, B. Santorini and M. Marcinkiewicz, Building a large annotated corpus of english: the Penn Treebank. *Computational Linguistics* **19** (1993) 313–330.
[19] D. Maurel, Pseudo-minimal transducer. *Theoretical Computer Science* **231** (2000) 129–139.
[20] M. Nivat and A. Podelski, Minimal ascending and descending tree automata. *SIAM J. Comput.* **26** (1997) 39–58.
[21] D. Revuz, *Dictionnaires et lexiques: méthodes et algorithmes*. Ph.D. thesis, Institut Blaise Pascal, Paris, France. LITP 91.44 (1991).
[22] D. Revuz, Dynamic acyclic minimal automaton, edited by S. Yu and A. Paun, *CIAA 2000, Fifth International Conference on Implementation and Application of Automata. Lect. Notes Comput. Sci.* **2088** (2000) 226–232.
[23] G. Rozenberg and A. Salomaa, *Handbook of Formal Languages*. Springer-Verlag, New York, Inc., Secaucus, NJ, USA (1997).
[24] A. Russell, Necessary and sufficient conditions for collision-free hashing, in *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, London, UK. Springer-Verlag (1993) 433–441.
[25] K. Sgarbas, N. Fakotakis and G. Kokkinakis, Two algorithms for incremental construction of directed acyclic word graphs. *International Journal on Artificial Intelligence Tools* **4** (1995) 369–381.

Communicated by C. Choffrut.
Received January 12, 2009. Accepted September 3, 2009.