

Perfect Hashing with Pseudo-minimal Bottom-up Deterministic Tree Automata

Jan Daciuk¹ and Rafael C. Carrasco²

¹ Knowledge Engineering Department, Gdańsk University of Technology, Poland,
(jandac@eti.pg.gda.pl)

² Dep. de Lenguajes y Sistemas Informáticos, Universidad de Alicante, E-03071
Alicante, Spain (carrasco@dlsi.ua.es).

Abstract

We describe a technique that maps unranked trees to their hash codes using a bottom-up deterministic tree automaton (DTA). In contrast to techniques implemented with minimal tree automata, our procedure builds a pseudo-minimal DTA. Pseudo-minimal automata are larger than the minimal ones but in turn the mapping can be arbitrary, so it can be determined prior to the automaton construction. We also provide procedures to build incrementally the pseudo-minimal DTA and the associated hash codes.

Keywords: minimal perfect hashing, deterministic tree automata, pseudo-minimal automata

1 Introduction

In many applications, there is a need to associate every piece of data with an integer which is later used as an identifier to access data-related information stored in some other structures. This number or *hash code* is computed from a part of the data (often called *key*) that holds enough information to identify the whole piece. As *hashing* (Czech *et al.*, 1997) transforms the key into an integer in a limited range, the domain of the key is usually orders of magnitude larger 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 (Russell, 1993), 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 0 to $n - 1$), it is called a *minimal perfect hash* function.

In particular, if keys are strings, acyclic deterministic finite-state automata (or acyclic DFA for short) can be used to implement minimal perfect hashing (Lucchiesi and Kowaltowski, 1993; Revuz, 1991). Using minimal DFA for this purpose provides both a very compact representation for the keys, and a very fast access to the data; however, hash codes cannot be modified arbitrarily.

In contrast, *pseudo-minimal automata* —first introduced by Dominique Revuz (Revuz, 1991) as a side-effect of a failed attempt at incremental construction of

minimal acyclic DFAs— can implement an arbitrary mapping from keys to integers (not limited, for instance, to that implied by lexicographical order of keys). This potential was discovered by Denis Maurel (Maurel, 2000) and has been applied to *dynamic perfect hashing* (Daciuk *et al.*, 2005), where new keys may be continually added (and old ones deleted) but the original mapping must be preserved, i.e., renumbering of keys is not allowed.

Trees are ubiquitous in computer science theory and applications. For instance, they are often used in the implementations of indexes in databases; they appear when structured information (as that contained in XML documents and in configuration files) is processed; and tree structures are also obtained as the result of parsing code. In natural language processing, trees are used to store annotated corpora and to represent syntactic constraints or connectivity of words in grammars such as XTAG (Doran *et al.*, 1994).

Tree automata recognize tree languages and deterministic tree automata (DTA) can process trees without the need for backtracking. 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 languages (Comon *et al.*, 2002) 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 hash codes. In section 2 previous work on pseudo-minimal DFAs is recalled. Pseudo-minimal DTA are introduced in section 3 and the procedure to build them incrementally is described in section 4. Section 5 shows how proper transitions can be identified in a pseudo-minimal DTA and how they can be used to assign a unique hash code for every tree in the language. Finally, conclusions are presented.

2 Preliminaries: pseudo-minimal DFA

As customary, a *deterministic finite-state automaton* (or DFA for short) is defined as $A = (Q, \Sigma, q_0, \delta, F)$ and consists of a finite set of states Q , a finite set of symbols Σ (called the *alphabet*), a start state $q_0 \in Q$ (also called the *initial state*), a transition function $\delta : Q \times \Sigma \mapsto Q$, and a set of final states $F \subseteq Q$ (also called *accepting states*).

For every state $q \in Q$ in a DFA, its right language is defined as

$$\vec{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta(q, w) \in F\} \quad (1)$$

while its left language is

$$\overleftarrow{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta(q_0, w) = q\} \quad (2)$$

The state $q \in Q$ is *useless* if either $\vec{\mathcal{L}}(q)$ or $\overleftarrow{\mathcal{L}}(q)$ are empty.

A DFA $A = (Q, \Sigma, q_0, \delta, F)$ is *proper* iff Q contains no useless states and for all $q \in Q$:

$$\left| \vec{\mathcal{L}}(q) \right| > 1 \Rightarrow \left| \overleftarrow{\mathcal{L}}(q) \right| = 1 \quad (3)$$

that is, if the right language of state q contains more than one string, then its left language contains a single string (and vice-versa). Clearly, a proper DFA must be acyclic.

A proper DFA A has the useful property that every string w accepted by A can be associated with a *proper element* for w . A proper element for w is a state or transition whose removal from F or δ respectively has a single effect on $L(A)$: w is removed from the language accepted by A . If an end-of-string marker is appended to every string then the proper DFA accepting that language is modified so that it has a single final state with all its incoming transitions labeled with the end-of-string marker, every $w \in L(A)$ has a *proper transition* and that transition can be used to store string-related information, e.g. an arbitrary number to implement perfect hashing.

A proper DFA A is *pseudo-minimal* iff it is smaller than any other proper DFA equivalent to A , that is, all proper automata B accepting $L(B) = L(A)$ satisfy $|B| > |A|$. A pseudo-minimal automaton is unique (up to isomorphisms).

The pseudo-minimal DFA equivalent to the proper DFA A can be easily obtained by merging states in the same class of a refinement of the partition computed by the standard minimization algorithms (Hopcroft and Ullman, 1979). In this refinement, the classes $[q]$ containing states with identical right languages are split as follows:

- class $[q]$ is refined into $|\overrightarrow{\mathcal{L}}(q)|$ classes, each one containing a single state, if $|\overrightarrow{\mathcal{L}}(q)| > 1$;
- class $[q]$ remains intact otherwise.

Recall that all states in $[q]$ have identical right languages and merging states just modifies their left languages.

3 Pseudo-minimal DTA

In contrast to finite-state automata, which accept strings, tree automata accept trees. For a given alphabet Σ , the language of *unranked ordered trees* T_Σ is defined as follows:

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

Any subset of T_Σ is called a *tree language*. In particular, the language of subtrees $\text{sub}(t)$ of a given tree $t \in T_\Sigma$ is defined as:

$$\text{sub}(t) = \begin{cases} \{\sigma\} & \text{if } t = \sigma \in \Sigma \\ \{t\} \cup \bigcup_{k=1}^m \text{sub}(t_k) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma \end{cases} \quad (4)$$

Every subtree of the first type (i.e., a symbol in Σ) is called a *leaf* of t .

A *finite-state tree automaton* (Comon *et al.*, 2002) is defined as $A = (Q, \Sigma, \Delta, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the *alphabet*, $\Delta \subset \bigcup_{i=0}^m \Sigma \times Q^{m+1}$ is a finite set of transitions, and $F \subseteq Q$ is a set of *final*

states. This definition differs from that of automata operating on strings in two remarkable aspects: 1) there is no start state, and 2) 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) iff for each $(\sigma, q_1, \dots, q_m) \in \Sigma \times Q^m$, there is at most one $q \in Q$ such that $(\sigma, q_1, \dots, q_m, q) \in \Delta$. For every DTA, one can define a collection of transition functions $\delta_m : \Sigma \times Q^m \mapsto Q$ as follows:

$$\delta_m(\sigma, q_1, \dots, q_m) = \begin{cases} q & \text{if } q \in Q \text{ is such that } (\sigma, q_1, \dots, q_m, q) \in \Delta \\ \perp & \text{if no such } q \in Q \text{ exists} \end{cases} \quad (5)$$

where \perp is the special *absorption state*, a state in $Q - F$ which is not used in Δ . For every transition $\tau = (\sigma, q_1, \dots, q_m, q)$, the states q_1, \dots, q_m are *source* or *input states* of τ and state q its *target* or *output*.

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

$$A(t) = \begin{cases} \delta_0(\sigma) & \text{if } t = \sigma \in \Sigma \\ \delta_m(\sigma, A(t_1), \dots, A(t_m)) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma \end{cases} \quad (6)$$

The language accepted at state q in an automaton A is the set of trees such that $A(t)$ returns q ,

$$L_A(q) = \{t \in T_\Sigma : A(t) = q\}, \quad (7)$$

and the language accepted by the whole automaton A is the sum of the languages accepted by all its final states:

$$L(A) = \bigcup_{q \in F} L_A(q). \quad (8)$$

A state q such that $L_A(q) = \emptyset$ is *unreachable*. Unreachable states can be safely removed from the automaton and, in the following, DTA will be assumed to have no unreachable states and therefore $|L_A(q)| > 0$ for all $q \in Q$.

In a DTA, the languages $L_A(q)$ play the role of the left languages $\overleftarrow{\mathcal{L}}(q)$ in a DFA: 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 accepted languages are disjoint. It is also possible to define the analogous to the right languages, denoted here with $R_A(q)$, as follows:

$$R_A(q) = \{t \in T_\Sigma^\# : A(t\#s) \in F \text{ for all } s \in L_A(q)\} \quad (9)$$

where $T_\Sigma^\#$ is the language of trees over $\Sigma \cup \{\#\}$ such that they have exactly one node with label $\#$, this node is a leaf and $t\#s$ represents the tree in T_Σ that is obtained after replacing the node labeled $\#$ with the subtree s .¹ As happens with the right languages $\overrightarrow{\mathcal{L}}$ in a DFA, equivalent states in a DTA have identical languages $R_A(q)$ (Sakakibara, 1992). A state q such that $R_A(q) = \emptyset$ is *useless*.

¹The replacement of the ‘‘pointed border node’’ (Nivat and Podelski, 1997) can be seen as a generalization for trees of string concatenation.

A DTA A is *acyclic* iff $A(t) = A(s)$ with $s \in \text{sub}(t)$ implies $s = t$ or $A(t) = \perp$. We also define the *fanout* of a state $q \in Q$ as

$$\text{fanout}(q) = \{(\sigma, i_1, \dots, i_m, j, n) \in \Delta \times \mathbb{N} : n \leq m \wedge i_n = q\} \quad (10)$$

and its *fanin* as

$$\text{fanin}(q) = \{(\sigma, i_1, \dots, i_m, j) \in \Delta : j = q\}. \quad (11)$$

The *minimal* automaton A_{\min} is the smallest among all DTA (with no unreachable states) accepting the same language:

$$A_{\min} = \text{argmin}_{B:L(B)=L(A)}(|B|) \quad (12)$$

There is only one minimal (up to isomorphisms) DTA accepting the language $L(A)$ and it can be obtained by merging pairs of equivalent states in A (Carrasco *et al.*, 2007). Two states p and q are *equivalent* ($p \equiv q$) iff $R_A(p) = R_A(q)$. This equivalence relation is a *congruence* (Brainerd, 1968), that is, all pairs of equivalent states (p, q) are both final or both non-final, and they are freely interchangeable as source states in any transition:

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

$$\delta_m(\sigma, i_1, \dots, i_{k-1}, p, i_{k+1}, \dots, i_m) \equiv \delta_m(\sigma, i_1, \dots, i_{k-1}, q, i_{k+1}, \dots, i_m) \quad (13)$$

In a minimal DTA, each equivalence class contains a single state, i.e., all pairs of states are inequivalent. Furthermore, $|R_A(q)| > 0$ for all states $q \neq \perp$.

A DTA A *proper* iff for no $q \in Q$ both $R_A(q)$ and $L_A(q)$ contain more than one tree. If A is proper, the minimal proper DTA equivalent to A can be obtained by merging pairs of pseudo-equivalent states. Two states p and q in a proper DTA are *pseudo-equivalent* ($p \simeq q$) iff they are equivalent and $R_A(p)$ contains at most one tree, that is,

$$p \simeq q \iff R_A(p) = R_A(q) \wedge |R_A(q)| \leq 1. \quad (14)$$

Pseudo-equivalence is also an equivalence relation, so it partitions Q into equivalence classes. In a pseudo-minimal automaton, there is only one state in each class. Of course, pseudo-minimal DTA are usually larger than the equivalent minimal DTA.

For instance, let $A = (Q, \Sigma, \Delta, F)$ such that $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Delta = \{(a, q_1), (b, q_2), (a, q_1, q_1, q_3), (a, q_1, q_2, q_3), (a, q_2, q_1, q_3), (a, q_2, q_2, q_3)\}$ and $F = \{q_3\}$. This DTA is proper, because the languages $L_A(q_1) = \{a\}$, $L_A(q_2) = \{b\}$ and $R_A(q_3) = \{\#\}$ contain a single tree. However, the minimal equivalent DTA $A_{\min} = (Q_{\min}, \Sigma, \Delta_{\min}, F_{\min})$ 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 $L_{A_{\min}}(q_1) = \{a, b\}$ and $R_{A_{\min}}(q_1) = \{a(\#a), a(\#b), a(a\#), a(b\#)\}$ contain more than one tree. Indeed, q_1 and q_2 are not pseudo-equivalent in A as $R_A(q_1) = R_A(q_2) = \{a(\#a), a(\#b), a(a\#), a(b\#)\}$ contains more than one tree.

4 Incremental Construction of Pseudo-minimal DTA

A traditional approach to build a minimal automaton recognizing a set of elements is to use a simple method for constructing 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. To alleviate this problem, 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 whole machine, the minimization can be local, acting only on those parts that have changed. The incremental construction of minimal DFAs has been addressed in a number of papers (Daciuk *et al.*, 2000; Aoe *et al.*, 1992; Revuz, 2000; Ciura and Deorowicz, 2001), and also some useful extensions (Carrasco and Forcada, 2002; Daciuk, 2004). The incremental construction of pseudo-minimal DFAs has been also addressed before (Daciuk *et al.*, 2006).

For minimal DTAs, incremental construction has been described in (Carrasco *et al.*, 2008) and is succinctly presented below. Figure 1 shows the main procedure to obtain a minimal DTA A' which accepts a tree t in addition to the language accepted by the input DTA A . In this procedure, Θ represents the set of states that are sources of transitions whose target is a new state, i.e. a state that belongs to $Q^\dagger - Q$, and the register R contains some of the states which remain mutually inequivalent after the modifications in the DTA.

```

1:  function AddTree( $A, t$ )
2:     $(A^\dagger, \Theta) \leftarrow \text{split}(A, t)$ ;
3:     $F \leftarrow F \cup \{A^\dagger(t)\}$ ;  $\Theta \leftarrow \Theta \cup \{A^\dagger(t)\}$ ;
4:     $R \leftarrow Q^\dagger - \{A^\dagger(s) : s \in \text{sub}(t)\}$ ;
5:    return  $A' \leftarrow \text{minim}(A^\dagger, R, \Theta)$ ;

```

FIGURE 1: Main procedure for the addition of a tree t to an minimal DTA A resulting in a new minimal automaton A' with $L(A') = L(A) \cup \{t\}$.

The procedure starts with a call to function `split`, shown in figure 2, which creates the product DTA (Carrasco *et al.*, 2008). When this recursive function is called with input A (a minimal DTA) and t (a tree), it returns a DTA A^\dagger such that $L(A^\dagger) = L(A)$ and $L_{A^\dagger}(A^\dagger(t)) = \{t\}$. After the recursive calls with the subtrees as input are finished, if there is already a single transition leading to $A^\dagger(t)$ then, the tree is simply traversed. Otherwise, if $q = A^\dagger(t)$ is the absorption state a new target state is created and, if $q \neq \perp$, then q is cloned. Cloning the state q means that a new state n is created and suitable transitions with source n are added to Δ^\dagger so that n is equivalent to q . Finally, source states of modified transitions are stored in Θ .

Note that, if A is a proper DTA, the DTA A^\dagger obtained after these steps remains proper. On the one hand, all states $n = A^\dagger(s)$ which are the output when A^\dagger operates on $s \in \text{sub}(t)$ satisfy $|L_A(n)| = 1$; on the other hand, the only relevant effect on the unvisited states q comes from the modification by `split` of the output

```

1:   function split( $A, t = \sigma(t_1, \dots, t_m)$ );
2:      $\Theta \leftarrow \emptyset$ ;
3:     for  $k \leftarrow 1 \dots m$  do
4:        $(A, \theta) \leftarrow \text{split}(A, t_k)$ ;
5:        $q_k \leftarrow A(t_k)$ ;  $\Theta \leftarrow \Theta \cup \theta$ ;
6:      $q \leftarrow \delta_m(\sigma, q_1, \dots, q_m)$ ;
7:     If  $|\text{fanin}(q)| \neq 1$  then
8:       Add a new state  $n$  to  $Q$ ;
9:       If  $|\text{fanin}(q)| = 0$  then
10:         $\Delta \leftarrow \Delta \cup \{(\sigma, q_1, \dots, q_m, n)\}$ ;
11:       else
12:         $A \leftarrow \text{clone}(A, q, n)$ ;
13:         $\Delta \leftarrow \Delta - \{(\sigma, q_1, \dots, q_m, q)\} \cup \{(\sigma, q_1, \dots, q_m, n)\}$ ;
14:         $\Theta \leftarrow \Theta \cup \{q_1, \dots, q_m\}$ ;
15:     return  $(A, \Theta)$ ;

```

FIGURE 2: Function `split` returning a new automaton A^\dagger with $L(A^\dagger) = L(A) \cup \{t\}$ and an auxiliary set Θ (see text for details).

of some transitions (as cloning only modifies $R_A(n)$ for new states n) which cannot increase $|L_A(q)|$ or $|R_A(q)|$.

```

1:   function minim( $A, R, \Theta$ );
2:      $\Omega \leftarrow \text{TopologicalSort}(Q - R)$ ;
3:     while  $\Omega \neq \emptyset$  do
4:        $n \leftarrow \text{pop}(\Omega)$ ;  $\tau \leftarrow (\sigma, q_1, \dots, q_m, n)$ ;
5:        $q \leftarrow \text{findEquiv}(A, R, \Theta, n)$ ;
6:       If  $q \neq n$  then
7:          $\Delta \leftarrow \Delta - (\sigma, q_1, \dots, q_m, n) \cup (\sigma, q_1, \dots, q_m, q)$ ;
8:         Remove  $n$  from  $Q, F, \delta$ , and  $\Theta$ ;
9:          $\Theta \leftarrow \Theta \cup \{q_1, \dots, q_m\}$ ;
10:      else
11:         $R \leftarrow R \cup \{n\}$ ;
12:     return  $A$ 

```

FIGURE 3: Function `minim` performing local minimization. States in the already minimized part of A are in R . Θ holds states that have changed their suite of outgoing transitions.

After `split`, function `minim` (figure 3) performs the local minimization. States are considered in reverse order of their processing by function `split` by taking states from a stack Ω which, in a practical implementation, can be computed inside function `split`. For every state n popped from Ω , function `findEquiv` returns a state $q \in R$ equivalent to n (or n , if such state does not exist). In the first case, q replaces n and in the second one n is added to the register R .

Function `findEquiv` uses the register R and the set Θ for efficiency: details on its implementation can be found in (Carrasco *et al.*, 2007).

According to equation (14), in order to implement the construction of pseudo-minimal instead of minimal DTAs, we need to replace line 5 in function `minim` with

5: **If** $|R_A(q)| \leq 1$ **then** $q \leftarrow \text{findEquiv}(A, R, \Theta, n)$ **else** $q \leftarrow n$;

and this predicate can be computed in acyclic DTAs as follows

$$|R_A(q)| \leq 1 \equiv \begin{cases} \text{true} & \text{if } \text{fanout}(q) = \emptyset \\ \text{false} & \text{if } |\text{fanout}(q)| > 1 \\ \text{false} & \text{if } |\text{fanout}(q)| = 1 \wedge q \in F \\ |R_A(j)| \leq 1 \wedge \forall_{k \neq n} |L_A(i_k)| \leq 1 & \text{otherwise, with } \text{fanout}(q) = \\ & \{(\sigma, i_1, \dots, i_m, j, n)\} \end{cases} \quad (15)$$

where $|L_A(q)| \leq 1$ can be computed for $q \neq \perp$ as² as

$$|L_A(q)| \leq 1 \equiv \begin{cases} \text{false} & \text{if } |\text{fanin}(q)| > 1 \\ \forall_k |L_A(i_k)| \leq 1 & \text{if } |\text{fanin}(q)| = 1 \text{ with } (\sigma, i_1, \dots, i_m, q) \in \Delta \end{cases} \quad (16)$$

With this additional computation, the DTA A' obtained using the incremental construction algorithm accepts $L(A') = L(A) \cup \{t\}$, remains proper and pseudo-minimal. The method be easily modified Carrasco *et al.* (2008) to deal with the removal of trees.

5 Identification of Proper Transitions

The procedure outlined in the previous section leads to a pseudo-minimal automaton. However, in order to implement arbitrary perfect hashing, proper transitions need to be identified. For the sake of simplicity we will assume that all trees are descendants of a super-root node labeled λ (the equivalent of the end-of-string marker in DFAs), the DTA contains a single accepting state Λ and Δ includes a transition (λ, q, Λ) for all $q \in F$. Clearly, $R_A(\Lambda) = \{\#\}$ and $A(\lambda(t)) = \Lambda$ iff $t \in L(A)$.

With this convention, we can select a proper transition for every $t \in L(A)$. Indeed, it is not difficult to show that for every $t \in L(A)$ there is at least one transition $\tau = (\sigma, i_1, \dots, i_m, j) \in \Delta$ such that τ is used when A operates on t , $|R_A(j)| = 1$ and either $m = 0$ or there is $k \leq m$ such that $|R_A(i_k)| > 1$. If we select such a transition as the proper transition for t , when new trees are added, this transition (or a cloned one) will remain proper for t in the resulting DTA.

Note that such a transition always exists. As $|R_A(\Lambda)| = 1$, there is either a subtree $s = \sigma(s_1 \dots s_m) \in \text{sub}(t)$ with $m > 0$ such that $\tau = (\sigma, A(s_1), \dots, A(s_m), A(s))$ satisfies $|R_A(A(s))| = 1$ and $|R_A(A(s_k))| > 1$ for some $k < m$ or there is a leaf $(\sigma, A(\sigma))$ with $|R_A(A(\sigma))| = 1$. Therefore, whenever a new tree t is added through the incremental algorithm, it is always possible to find such a transition for t . Note

²In the absence of unreachable states, $|\text{fanin}(q)| > 0$.

also that a proper transition in A for a tree $s \neq t$ will either remain proper in A' (if states are unvisited) or will be cloned (and the cloned transition is less used, so it is suitable as proper transition for s).

Finally, local pseudo-minimization only considers those states q such that $|R_A(q)| = 1$ and thus, a transition $(\sigma, i_1, \dots, i_m, j)$ of the type selected, with $|R_A(i_k)| > 1$, will not be merged.

Function *split* will create state Λ for the first tree added to an automaton recognizing the empty language, and the hash code will be put on the transition labeled with λ . For every subsequent tree t , function *split* will create a clone Λ' of Λ , which will then be merged with Λ . Pseudo-minimization may result in more mergers. As transitions leading from the last merged state q are not proper, the hash code for t will be put on a transition $\tau = (\sigma, p_1, \dots, p_m, q), \forall_{0 \leq i \leq m} p_i \in \text{sub}(t)$. If now $|\text{fanin}(q)| = 2$, transitions above q will no longer be proper, and a hash code written on one of them has to be moved onto the other transition leading to q . If $|\text{fanin}(q)| > 2$, transitions above q were not proper before the merger.

6 Conclusions

We have presented a method to implement perfect hashing on trees using bottom-up deterministic tree automata. Instead of using minimal DTAs —as in (Daciuk, 2007)—, this method uses pseudo-minimal automata which, at the cost of a size increase, can implement arbitrary hashing. The algorithm presented in (Carrasco *et al.*, 2008) has been adapted for the incremental construction of pseudo-minimal DTAs and, from this construction, a procedure to identify the proper elements has been deduced. We plan to compare the performance of this method with that of methods using unminimized DTA.

Acknowledgments

This research was partially supported by the Spanish CICYT through grants TIN2006-15071-C03-01 and TIC2003-08681-C02-01.

References

- J. AOE, K. MORIMOTO, and M. HASE (1992), An Algorithm for Compressing Common Suffixes Used in Trie Structures, *Trans. IEICE*, J75-D-II:770–779.
- Walter S. BRAINERD (1968), The Minimalization of Tree Automata, *Information and Control*, 13(5):484–491.
- Rafael C. CARRASCO, Jan DACIUK, and Mikel L. FORCADA (2007), An Implementation of Deterministic Tree Automata Minimization, in Jan HOLUB and Jan ŽD'ÁREK, editors, *Implementation and Application of Automata. 12th International Conference, CIAA 2007, Prague, Czech republic, July 2007*, volume 4783 of *LNCS*, pp. 122–129, Springer.
- Rafael C. CARRASCO, Jan DACIUK, and Mikel L. FORCADA (2008), Incremental Construction of Minimal Tree Automata, *Algorithmica*, to appear. Available at <http://dx.doi.org/10.1007/s00453-008-9172-4>.
- Rafael C. CARRASCO and Mikel L. FORCADA (2002), Incremental Construction and Maintenance of Minimal Finite-State Automata, *Computational Linguistics*, 28(2).

- Marcin CIURA and Sebastian DEOROWICZ (2001), How to Squeeze a Lexicon, *Software – Practice and Experience*, 31(11):1077–1090.
- Hubert COMON, Max DAUCHET, Rémi GILLERON, Florent JACQUEMARD, Denis LUGIER, Sophie TISON, and Marc TOMMASI (2002), Tree Automata Techniques and Applications, draft book, available at <http://www.grappa.univ-lille3.fr/tata>.
- Zbigniew J. CZECH, George HAVAS, and Bohdan S. MAJEWSKI (1997), Perfect hashing, *Theoretical Computer Science*, 182:1–143.
- Jan DACIUK (2004), Comments on Incremental Construction and Maintenance of Minimal Finite-State Automata by Rafael C. Carrasco and Mikel Forcada, *Computational Linguistics*, 30(2):227–235.
- Jan DACIUK (2007), Perfect Hashing Tree Automata, in *Proceedings of Finite-State Methods in Natural Language Processing*, Potsdam, September 14–16.
- Jan DACIUK, Denis MAUREL, and Agata SAVARY (2005), Dynamic Perfect Hashing with Finite-State Automata, in Mieczysław A. KŁOPOTEK, Sławomir WIERZCHOŃ, and Krzysztof TROJANOWSKI, editors, *Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM'05 Conference held in Gdańsk, Poland, June 13-16, 2005*, volume 31 of *Advances in Soft Computing*, pp. 169–178, Springer.
- Jan DACIUK, Denis MAUREL, and Agata SAVARY (2006), Incremental and Semi-incremental Construction of Pseudo-Minimal Automata, in Jacques FARRE, Igor LIPOVSKY, and Sylvain SCHMITZ, editors, *Implementation and Application of Automata: 10th International Conference, CIAA 2005*, volume 3845 of *LNCS*, pp. 341–342, Springer.
- Jan DACIUK, Stoyan MIHOV, Bruce WATSON, and Richard WATSON (2000), Incremental Construction of Minimal Acyclic Finite State Automata, *Computational Linguistics*, 26(1):3–16.
- Christy DORAN, Dania EGEDI, Beth Ann HOCKEY, B. SRINIVAS, and Martin ZAIDEL (1994), XTAG System – A Wide Coverage Grammar for English, in *Proceedings of the 15th International Conference on Computational Linguistics (COLING 94)*, volume II, pp. 922–928, Kyoto, Japan.
- J. HOPCROFT and J. D. ULLMAN (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, MA.
- Claudio LUCCHIESI and Tomasz KOWALTOWSKI (1993), Applications of Finite Automata Representing Large Vocabularies, *Software Practice and Experience*, 23(1):15–30.
- Denis MAUREL (2000), Pseudo-minimal transducer, *Theoretical Computer Science*, 231(1):129–139.
- Maurice NIVAT and Andreas PODELSKI (1997), Minimal Ascending and Descending Tree Automata, *SIAM J. Comput.*, 26(1):39–58.
- Dominique REVUZ (1991), *Dictionnaires et lexiques: méthodes et algorithmes*, Ph.D. thesis, Institut Blaise Pascal, Paris, France, IITP 91.44.
- Dominique REVUZ (2000), Dynamic Acyclic Minimal Automaton, in *CIAA 2000, Fifth International Conference on Implementation and Application of Automata*, pp. 226–232, London, Canada.
- Alexander RUSSELL (1993), Necessary and Sufficient Conditions For Collision-Free Hashing, in *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pp. 433–441, Springer-Verlag, London, UK, ISBN 3-540-57340-2.
- Yasubumi SAKAKIBARA (1992), Efficient Learning of Context-Free Grammars from Positive Structural Examples, *Information and Computation*, 97(1):23–60.