Perfect Hashing Tree Automata

Jan Daciuk

Department of Knowledge Engineering, Gdańsk University of Technology, Poland

Abstract. We present an algorithm that computes a function that assigns consecutive integers to trees recognized by a deterministic, acyclic, finite-state, bottom-up tree automaton. Such function is called minimal perfect hashing. It can be used to identify trees recognized by the automaton. Its value may be seen as an index in some other data structures. We also present an algorithm for inverted hashing.

1 Introduction

Hashing [1] is a technique where a key is transformed into an integer in a limited range with a *hash function*. Usually, there are far more possible keys than integers in the range, so conflicts where different keys are mapped into the same integers are unavoidable. However, in certain contexts, it is possible to map n keys without any conflicts. A function that implements it is called a *perfect hash function*. If it maps n keys into a consecutive range of n integers, it is called a *minimal perfect hash function*.

The nature of a hash function and its application depends closely on the hash key. Minimal, deterministic, acyclic, finite state automata (FSAs) provide a minimal perfect hash function on strings [2], [3]. This allows for fast and compact representation of dictionaries storing arbitrary information associated with words. An insight from perfect hashing with FSAs is useful in developing perfect hashing with deterministic, acyclic, bottom-up tree automata (DTAs), although the latter case is more complex. DTAs store a finite set of finite trees. Trees are ubiquitous in both computer science and natural language processing. They are used e.g. for storing the result of parsing a program or a sentence. A language that is best suited for parsing with DTAs is XML. It is widely used both in computer science and in natural language processing. For example, in natural language processing, it is used for annotating corpora. Perfect hashing with tree automata implements a mapping from trees to integers. It can be used for identification of trees, which allows for e.g. retrieval of arbitrary information associated with the given tree, like all locations in a corpus where the given parse tree occurs. The inverse mapping has an even greater potential, as a tree automaton can be used as a compact representation for a forest of trees. A mapping from an integer to a tree makes it possible to retrieve fast a given tree. A dictionary associating words with trees can be implemented as a perfect hash FSA associating words with numbers, a vector associating word numbers with tree numbers, and a perfect hash DTA.

The rest of the paper is structured as follows: Section 2 provides basic definitions, while Section 3 discusses issues related to counting trees in a tree automaton. An implementation of a minimal perfect hash function and its inverse is given in Section 4, while their complexity is investigated in Section 5. The paper ends with conclusions given in Section 7.

2 Basic Definitions

A *finite-state, bottom-up tree automaton* [4] 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 final set of *transitions*, and $F \subseteq Q$ is a set of *final states*. Another name for bottom-up is *frontier-to-root*. Another name for final states is *accepting states*. This definition is similar to the definition of finite-state automata, except for two differences: there is no initial (start) state, and a transition is a relation between an alphabet symbol and an arbitrary number of states (and not necessarily two states, as in case of finite-state automata). In a *deterministic*, finite-state, bottom-up tree automaton (or a DTA for short), for each $(\sigma, q_1, \ldots, q_m) \in \Sigma \times Q^m$, there is at most one $q \in Q$ such that $(\sigma, q_1, \ldots, q_m, q) \in \Delta$. In that case, we can define a function δ:

$$\delta_m(\sigma, q_1, ..., q_m) = \begin{cases} q & \text{if } q \in Q \text{ is such that } (\sigma, q_1, ..., q_m, q) \in \Delta \\ \bot & \text{if no such } q \in Q \text{ exists} \end{cases} \tag{1}$$

States $q_1, \ldots, q_m$ are *source states*, while q is a *target state*. Finite-state automata accept strings. Tree automata accept trees. Trees are defined as follows:

1. Each symbol $\sigma \in \Sigma$ is a tree.
2. For each $t = \sigma(t_1, \ldots, t_m)$, where $\sigma \in \Sigma$, and $t_1, \ldots, t_m$, $m \geq 0$ are trees, t is a tree.

Any subset of all trees defined over an alphabet Σ is called a *tree language* T_Σ. We can define an extended transition function on trees:

$$\delta_A(t) = \begin{cases} \delta_0(\sigma) & \text{if } t = \sigma \in \Sigma \\ \delta_m(\sigma, \delta_A(t_1), \ldots, \delta_A(t_m)) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma - \Sigma \end{cases} \tag{2}$$

A language of a state q in an automaton A is a set of trees such that the extended transition function returns q for each of them:

$$L_A(q) = \{t \in T_\Sigma : \delta_A(t) = q\} \tag{3}$$

A language of the whole automaton A is the union of the languages of all its final states:

$$L(A) = \bigcup_{q \in F} L_A(q). \tag{4}$$

A language of a transition $\tau = (\sigma, q_1, \ldots, q_m, q) \in \Delta$ is the subset of $L_A(q)$ recognized by following τ:

$$L_A(\tau) = \begin{cases} \sigma & \tau = (\sigma, q) \\ \bigcup_{(t_1,\ldots,t_m)\in L(q_1)\times\ldots\times L(q_m)} \sigma(t_1, \ldots, t_m) & \tau = (\sigma, q_1, \ldots, q_m, q) \end{cases} \tag{5}$$

A DTA A is *acyclic*, when $L(A)$ is a finite set of finite trees. From this moment on in the paper, when we refer to automata, we mean deterministic, acyclic finite-state, bottom-up tree automata without useless states, unless otherwise specifically stated.

3 Numbering Trees in a DTA

A tree t has number i (counting from 0 to $n-1$, where $n = |L(A)|$ is the number of trees recognized by the automaton) if there are i trees that precede it in an order imposed by the automaton. To compute i, we have to count the trees that precede t. The first step is to compute the number of trees that precede the current tree t in the language of the state $\delta_A(t)$:

$$\iota_A(t) = |\{t' : \delta_A(t') = \delta_A(t) \wedge t' \prec_A t\}| \tag{6}$$

This can be done recursively. Let $t = \sigma(t_1, \ldots, t_m)$, $\tau = (\sigma, \delta_A(t_1), \ldots, \delta_A(t_m), q)$, and $q = \delta_A(t)$. Then $\iota_A(t)$ is the sum of the number of trees that precede t but use the same transition τ, and the number of trees recognized while following transitions preceding τ:

$$\iota_A(t) = \rho_A(t) + \sum\nolimits_{\tau'=(\sigma',q'_1,\ldots,q'_{m'},q)\prec_A\tau} |L_A(\tau')| \tag{7}$$

The language of a transition (see Equation (5)) can also be defined recursively:

$$L_A(\tau) = \begin{cases} \sigma & \text{if } \tau = (\sigma, q) \\ \bigcup_{(t'_1,\ldots,t'_m)\in L_A(q_1)\times\ldots\times L_A(q_m)} \sigma(t'_1, \ldots, t'_m) & \text{if } \tau = (\sigma, q_1, \ldots, q_m, q) \end{cases} \tag{8}$$

Its cardinality is:

$$|L_A(\tau)| = \begin{cases} 1 & \text{if } \tau = (\sigma, q) \\ \prod_{i=1}^{m} |L_A(q_i)| & \text{if } \tau = (\sigma, q_1, \ldots, q_m, q), m > 0 \end{cases} \tag{9}$$

The key component in (9) is $|L_A(q)|$. We rewrite definition (3) recursively:

$$L_A(q) = \bigcup\nolimits_{\tau=(\sigma,q_1,\ldots,q_m,q)\in\Delta, m\geq 0} |L_A(\tau)| \tag{10}$$

so that its cardinality can easily be computed as:

$$|L_A(q)| = \sum\nolimits_{\tau=(\sigma,q_1,\ldots,q_m,q)\in\Delta, m\geq 0} |L_A(\tau)| \tag{11}$$

To compute $\rho_A(t)$, i.e. the number of trees that reach $q = \delta_A(t)$ by the same transition τ and precede t, we need to introduce some order of trees in the language $L_A(\tau)$. Let $\tau = (\sigma, q_1, \ldots, q_m, q)$, and let $\mathrm{next}(t_i)$ be the next subtree in $L_A(q_i)$. Then $\forall_{1 \leq j < k \leq m} \sigma(t'_1, \ldots, \mathrm{next}(t'_j), \ldots, t'_k, \ldots, t'_m) \prec_A \sigma(t'_1, \ldots, t'_j, \ldots, \mathrm{next}(t'_k), \ldots, t'_m)$, where $t'_i \in L_A(q_i)$. So

$$\rho_A(t) = \begin{cases} 1 & \text{if } t \in \Sigma \\ \sum_{i=1}^{m} \iota_A(t_i) \cdot \prod_{j=i+1}^{m} |L_A(\delta_A(t_j))| & \text{if } t = \sigma(t_1, \ldots, t_m) \in T_\Sigma - \Sigma \end{cases} \tag{12}$$

In practice, we would use $\rho_A{}^i$ defined as:

$$\rho_A{}^i(t) = \begin{cases} 1 & \text{if } i = 0 \\ \iota_A{}^i(t) = \rho_A{}^{i-1}(t) \cdot |L(\delta_A(t_i))| + \iota_A(t_i) & \text{if } 1 \leq i \leq m \end{cases} \tag{13}$$

Thus, $\rho_A(t) = \rho_A{}^m(t)$.

A tree t is recognized if $\delta_A(t) \in F$. However, there may be more than one final state, and languages of final states are disjoint. We assume that final states $f_i \in F$ are ordered: $f_1 \prec_A \ldots \prec_A f_{|F|}$. A tree number for a tree t is thus $\iota_A(t)$ plus $\sum_{F \ni f' \prec_A \delta_A(t)} |L_A(f')|$.

4 Perfect Hash Function

The perfect hash function is given on Figure 1. The argument is a tree t, for which we want to obtain the hash value. A call to function *rh* in line 2 returns a pair $(\delta_A(t), \iota_A(t))$. The loop in lines 4–6 adds the number of trees belonging to languages of those final states that precede $\delta_A(t)$. If $t \notin L(A)$, i.e. either $\delta_A(t) \notin Q$ or $\delta_A(t) \notin F$, $h_A(t)$ returns -1. In function *rh*, the loop in line 13 finds numbers associated with subtrees t_i of t, and the loop in lines 14–20 computes $\rho_A{}^i(t)$. In lines 22–25, $\sum_{\tau'(\sigma', q'_1, \ldots, q'_{m'}, q) \prec_A \tau} |L_A(\tau')|$ is added to that value.

Inverse perfect hash function is given on Figure 2. The parameter n is the tree number. First, we process the final states f_i one by one, keeping the number of trees recognized at all preceding final states in the variable h. If $h \leq n < h + |L_A(f_i)|$, then the tree number n belongs to the language of f_i, and it is $(n - h)$-th tree in that language. Function rh^{-1} takes two parameters: a state q being the root of a subtree, and a tree number among all trees in $L_A(q)$. All transitions reaching q are tried in order, and the number of trees recognized while following them is added to variable h. The process continues until for the current transition τ_i, $h \leq n < h + |L_A(\tau_i)|$. Then the subtree we are looking for belongs to $L_A(\tau_i)$. To calculate the subtree number among $L_A(\tau_i)$, the subtree t is decomposed into individual subtrees $t_1, \ldots, t_{m_i}$ with roots being the states $q_1, \ldots, q_{m_i}$. The loop in lines 17–20 builds the tree t from its subtrees. Basically, it calculates the inverse of $\rho_A{}^i(t)$.

Note that the values $|L_A(q)|$ for all states, and $|L_A(\tau)|$ for all transitions can be calculated in advance and stored in appropriate states and transitions.

```
1:  **function** $h_A(t)$
2:      $(q, v) \leftarrow \mathrm{rh}(t)$;
3:      **if** $q \in F \wedge v \geq 0$ **then**
4:          **foreach** $p \in F : p \prec_A q$
5:              $v \leftarrow v + |L_A(p)|$;
6:          **end foreach**;
7:          **return** $v$;
8:      **else**
9:          **return** -1;
10:     **end if**;
11: **end function**;

12: **function** $\mathrm{rh}(t = \sigma(t_1, \ldots, t_m))$
13:     $h \leftarrow 0$; **for** $i \leftarrow 1$ **to** $m$ **do** $(q_i, v_i) \leftarrow \mathrm{rh}(t_i)$ **end for**;
14:     **for** $i \leftarrow 1$ **to** $m$
15:         **if** $q_i = \perp \vee v_i = -1$ **then**
16:             **return** -1;
17:         **else**
18:             $h \leftarrow h \cdot |L_A(q_i)| + v_i$;
19:         **end if**;
20:     **end for**;
21:     $q \leftarrow \delta(\sigma, q_1, \ldots, q_m)$;
        Let $N = |\{(\sigma, q_1, \ldots, q_m, q) \in \Delta\}|$;
        Let $\tau_1 \prec_A \ldots \prec_A \tau_N$, $\tau_i = (\sigma_i, q_{1_i}, \ldots, q_{m_i}, q)$;
22:     $i \leftarrow 1$;
23:     **while** $\tau_i < (\sigma, q_1, \ldots, q_m, q)$ **do**
24:         $h \leftarrow h + |L_A(\tau_i)|$; $i \leftarrow i + 1$;
25:     **end while**;
26:     **return** $(q, h)$;
27: **end function**;
```

Fig. 1. Perfect hash function

5 Computational Complexity

The time complexity of the perfect hash function given on Figure 1 is $\mathcal{O}(|t| + |F| + |\Delta|)$, where $|t|$ is the number of tree nodes – defined below in (14), $|F|$ is the number of final states, and $|\Delta| = |\{\tau : \tau \in \Delta\}|$ is the number of transitions in the automaton (this could be replaced with $|\Delta| = \sum_{\tau \in \Delta} |\tau|$, where $|\tau| = m + 1$, in case we were not to store $|L_A(\tau)|$ in transitions).

$$|t| = \begin{cases} 1 & \text{if } t = \sigma \in \Sigma \\ 1 + |t_1| + \ldots + |t_m| & \text{if } t = \sigma(t_1, \ldots, t_m) \in T_\Sigma - \Sigma \end{cases} \tag{14}$$

In function $h_A(t)$, we have one loop that executes at most $|F|$ times, and consists of adding a constant to a variable (a constant-time operation), as well as a call to function *rh*. In function *rh*, there is a loop in lines 23–25 that adds a constant to

```
1:  function $h_A{}^{-1}(n)$
2:      $h \leftarrow 0$;
3:      for $i \in 1, \ldots, |F| : f_1 \prec_A \ldots \prec_A f_{|F|}$ do
4:          if $h + |L_A(f_i)| > n$ then
5:              return $rh^{-1}(f_i, n - h)$;
6:          else
7:              $h \leftarrow h + |L_A(f_i)|$;
8:          end if;
9:      end for;
10: end function;

11: function $rh^{-1}(q, n)$
        Let $N = |\{(\sigma, q_1, \ldots, q_m, q) \in \Delta\}|$;
        Let $\tau_1 \prec_A \ldots \prec_A \tau_N$, $\tau_i = (\sigma_i, q_{1_i}, \ldots, q_{m_i}, q)$;
12:     $i \leftarrow 1$; $h \leftarrow 0$;
13:     while $h + |L_A(\tau_i)| \leq n$ do
14:         $h \leftarrow h + |L_A(\tau_i)|$; $i \leftarrow i + 1$;
15:     end while;
16:     $h \leftarrow n - h$; $th \leftarrow |L_A(\tau_i)|$;
17:     for $j = 1, \ldots, m_i$ do
18:         $th \leftarrow th / |L(q_{j,i})|$;
19:         $t_j \leftarrow rh^{-1}(q_{j,i}, h/th)$; $h \leftarrow h - (h/th)$;
20:     end for;
21:     return $\sigma_i(t_1, \ldots, t_{m_i})$;
22: end function;
```

Fig. 2. Inverse perfect hash function

a variable and increments a variable – also constant-time operations. As only $|\Delta|$ transitions can precede the current one across all calls to *rh*, this loop contributes an $\mathcal{O}(|\Delta|)$ component to the time complexity. Another loop in the same function in lines 14–21 contains constant-time operations and one recursive call to *rh*. Since there is one call to *rh* per every node of the tree, this contributes $|t|$ to the time complexity.

There is one important trick that eliminates the $\mathcal{O}(|\Delta|)$ component, and also reduces the size of the automaton as we are no longer forced to keep back-transitions. Instead of storing $|L_A(\tau)|$ in transitions, we store $\sum_{\tau' \in \Delta : \tau' \prec_A \tau} |L_A(\tau')|$ there. The computation in lines 24–26 is then no longer needed. The same can be done with final states, i.e. they can hold the number of trees recognized in those final states that precede the current one. This eliminates the $\mathcal{O}(|F|)$ component, giving us $\mathcal{O}(|t|)$ time complexity.

For the inverse perfect hashing, the time complexity is $\mathcal{O}(|t| + |F| + |\Delta|)$, regardless of the use of the trick described above. We also have to keep back transitions, as we need to find a transition (or a final state) with the appropriate value. The loop in function $h_A{}^{-1}(t)$ executes at most $|F|$ times, with all but one run containing constant-time operations. The loop is finished with a single call to $rh^{-1}(q,n)$. Inside that function, the loop in lines 13–15 counts transitions – at most $|\Delta|$ across all calls, and it calls itself – once per tree node, giving the $|t|$ component.

6 Example

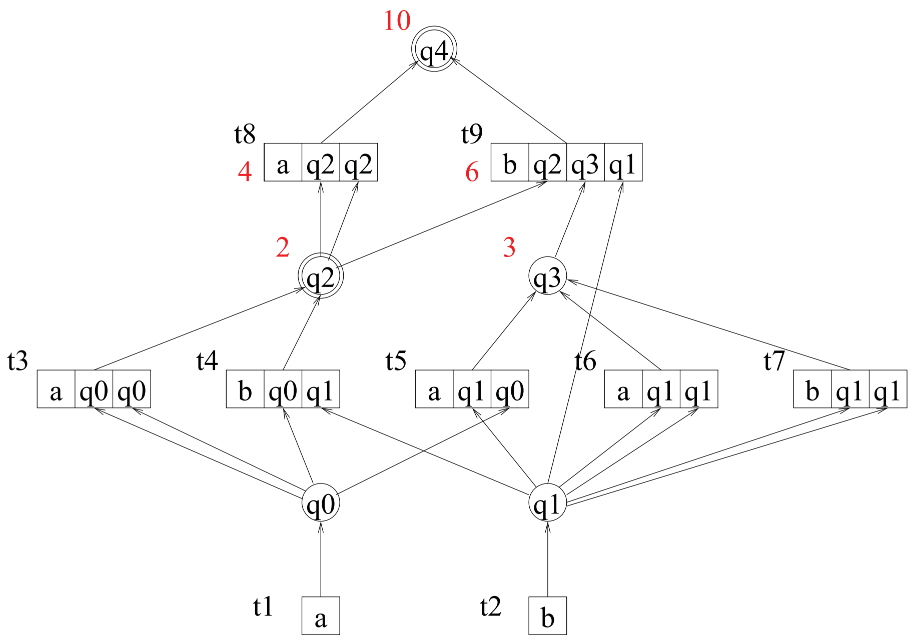

Fig. 3. A DTA $A = (\{q_0, \ldots, q_4\}, \{a, b\}, \Delta = \{t_1, \ldots, t_9\}, F = \{q_2, q_4\})$. Numbers by circles representing states, or by boxes representing transitions, give the cardinalities of the languages of states and transitions, respectively. If the number is not given, it is 1.

A DTA recognizing trees:

0. $a(a, a)$,
1. $b(a, b)$,
2. $a(a(a, a), a(a, a))$,
3. $a(b(a, b), a(a, a))$,
4. $a(a(a, a), b(a, b))$,
5. $a(b(a, b), b(a, b))$,
6. $b(a(a, a), a(b, a), b)$,
7. $b(b(a, b), a(b, a), b)$,
8. $b(a, (a, a), a(b, b), b)$,

9. $b(b(a, b), a(b, b), b)$,
10. $b(a(a, a), b(b, b), b)$,
11. $b(b(a, b), b(b, b), b)$

is given on Figure 3. Let us find the number $h_A(t)$ associated with $t = b(b(a, b), a(b, b), b)$. First, $rh(t)$ is called. It is called on t and recursively on each of its subtrees. We call $rh(b(b(a, b), a(b, b), b))$, take the first subtree and call $rh(b(a, b))$, then take the first subtree and call $rh(a)$. The last call returns $(q_0, 0)$, since q is set in line 22, and h, which is set in line 13, cannot be modified in loops in lines 14–21 and 24–26 as they are not executed. Similarly, $rh(b)$ returns $(q_1, 0)$. In $rh(b(a, b))$, the loop in lines 14–21 runs twice, but leaving $h = 0$. The loop in lines 24–26 increases h by $|L_A(t_3)|$ (the number of trees in languages of preceding transitions), making $rh(b(a, b))$ return $(q_2, 1)$. Next, $rh(a(b, b))$ is called, which in turn calls $rh(b)$ (twice), which returns $(q_1, 0)$ (twice) as described above. In $rh(a(b, b))$, the loop in lines 14–21 leaves $h = 0$, as no trees precede $a(b, b)$ in the language of t_6, but the loop in lines 24–26 increases that value by $|L_A(t_5)|$, i.e. the sun of cardinalities of languages of transitions preceding t_6, making the function return $(q_3, 1)$. Back in $rh(b(b(a, b), a(b, b), b))$, the value returned by $rh(b(a, b))$ (i.e. 1) is multiplied by $|L_A(q_3)| = 3$ before adding 1 returned by $rh(a(b, b))$. The result is 4. Since $rh(b)$ returned $(q_1, 0)$, and $|L_A(q_1)| = 1$, 4 is multiplied by 1, and then 0 is added. Afer having added $|L_A(t_8)|$, the value returned by $rh(b(b(a, b), a(b, b), b))$ is then $(q_4, 8)$. Since $q_2 \in F$ precedes q_4, and $|L_A(q_2)| = 2$, $h_A(t)$ returns $8 + 2 = 10$.

Now, let us find which tree has number 10. The process is illustrated in Table 1.

7 Conclusions

We have presented an efficient implementation for minimal perfect hashing with finite-state, deterministic, acyclic, bottom-up tree automata. It can be used for computing an index for trees that can further be used to access additional data structures associated with the trees. We have also shown how to compute the inverse perfect hash function, which can help to retrieve trees stored in a compact way in a tree automaton. Our implementation for minimal automata does not preserve the order of trees at input. However, when the automata are to be used in a static way, the order imposed by the automaton can easily be found.

The author wishes to thank Rafael Carrasco and Mikel Forcada for discussions about tree automata during the author's short visits to Alicante supported by the Spanish CICyT through grants TIN2006-15071-C03-01 and TIC2003-08681-C02-01, and for jointly developing a program that helped in testing ideas presented in this paper. Comments from reviewers helped in improving the paper.

References

1. Czech, Z.J., Havas, G., Majewski, B.S.: Fundamental study perfect hashing. Theoretical Computer Science **182** (1997) 1–143

$\boxed{h_A{}^{-1}(10)}$			
$i \leftarrow 1$; $p \leftarrow q_2$;			
$h \leftarrow 2$;			
$i \leftarrow 2$; $p \leftarrow q_4$;			
$rh^{-1}(q_4, 10-2)$;	$\boxed{rh^{-1}(q_4, 8)}$		
	$i \leftarrow 1$; $\tau_i = t_8$;		
	$h \leftarrow 4$; $i \leftarrow 2$;		
	$\tau_i \leftarrow t_9$;		
	$h \leftarrow 4$; $th \leftarrow 6$;		
	$j \leftarrow 1$; $th \leftarrow 3$;		
	$rh^{-1}(q_2, 1)$;	$\boxed{rh^{-1}(q_2, 1)}$	
		$i \leftarrow 1$; $\tau_i = t_3$;	
		$h \leftarrow 1$; $i \leftarrow 2$;	
		$\tau_i = t_4$;	
		$h \leftarrow 0$; $th \leftarrow 1$;	
		$j \leftarrow 1$; $th \leftarrow 1$;	
		$rh^{-1}(q_0, 0)$;	$\boxed{rh^{-1}(q_0, 0)}$
		$t_1 \leftarrow a$; $h \leftarrow 0$;	$\hookleftarrow a$;
		$j \leftarrow 2$; $th \leftarrow 1$;	
		$rh^{-1}(q_1, 0)$;	$\boxed{rh^{-1}(q_1, 0)}$
		$t_2 \leftarrow b$; $h \leftarrow 0$;	$\hookleftarrow b$;
	$t_1 \leftarrow b(a, b)$;	$\hookleftarrow b(a, b)$;	
	$h \leftarrow 1$;		
	$j \leftarrow 2$; $th \leftarrow 1$;		
	$rh^{-1}(q_3, 1)$;	$\boxed{rh^{-1}(q_3, 1)}$	
		$i \leftarrow i$; $\tau_i \leftarrow t_5$;	
		$h \leftarrow 1$; $i \leftarrow 2$;	
		$\tau_i \leftarrow t_6$;	
		$h \leftarrow 0$; $th \leftarrow 1$;	
		$j \leftarrow 1$; $th \leftarrow 1$;	
		$rh^{-1}(q_1, 0)$;	$\boxed{rh^{-1}(q_1, 0)}$
		$t_1 \leftarrow b$; $h \leftarrow 0$;	$\hookleftarrow b$;
		$j \leftarrow 2$; $th \leftarrow 1$;	
		$rh^{-1}(q_1, 0)$	$\boxed{rh^{-1}(q_1, 0)}$
		$t_2 \leftarrow b$; $h \leftarrow 0$;	$\hookleftarrow b$;
	$t_2 \leftarrow a(b, b)$;	$\hookleftarrow a(b, b)$;	
	$h \leftarrow 0$;		
	$j \leftarrow 3$; $th \leftarrow 1$;		
	$rh^{-1}(q_1, 0)$;	$\boxed{rh^{-1}(q_1, 0)}$	
	$t_3 \leftarrow b$; $h \leftarrow 0$;	$\hookleftarrow b$;	
$\hookleftarrow$	$\hookleftarrow$		
$b(b(a, b), a(b, b), b)$	$b(b(a, b), a(b, b), b)$		

Table 1. Call sequence in $h_A{}^{-1}(10)$.

2. Lucchiesi, C., Kowaltowski, T.: Applications of finite automata representing large vocabularies. Software Practice and Experience **23**(1) (Jan. 1993) 15–30
3. Revuz, D.: Dictionnaires et lexiques: méthodes et algorithmes. PhD thesis, Institut Blaise Pascal, Paris, France (1991) LITP 91.44.
4. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugier, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Draft book, available at http://www.grappa.univ-lille3.fr/tata (September 2002)