

UNIVERSITY OF BAYREUTH

BACHELOR SEMINAR TREE AUTOMATA

Introduction to Ranked Tree Automata

Author:
Martin BRAUN

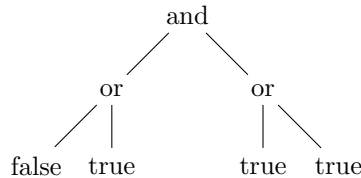
Supervisor:
Prof. Dr. Wim MARTENS

Introduction to Tree Languages

Regular Tree Languages are a powerful tool when it comes to parsing data given in a textual form. However, they lack in the context of parsing hierarchical data. Using Tree Languages to define your data structure can help with this shortcoming. A good example for a tree language is the one consisting of all binary boolean expressions evaluating to true, for which an instance - if formatted in the right way - could look like this:

$and(or(false, true), or(true, true))$

To simplify, the elements of the language are often represented as a tree in a graphical way:



Just like for regular word languages, it is of interest to know whether a given word (in this case a tree) is part of the (tree-)language. In order to describe an automaton that recognizes tree-languages we have to define what **ranked tree alphabets**, Σ -trees and (regular) **tree-languages** are, first.

Definition 1. *ranked tree alphabet [2][5]*

A **ranked tree alphabet** Σ of **arity** n is a refinement of an ordinary alphabet such that each symbol $\sigma \in \Sigma$ either has zero or exactly n arguments. Symbols with zero arguments are called **constants**.

*Note: There also exists a definition for tree alphabets that don't have the restriction of arity. These are called **unranked tree alphabets**, but in this paper we will only take a look at the ranked case.*

alle Beispiele muessen diese Art der Notation dann auch benutzen

Definition 2. Σ -tree [1]

The set of Σ -trees T_Σ over the ranked tree alphabet Σ of arity n is inductively defined as follows:

1. every $\sigma \in \Sigma$ is a Σ -tree
2. $\sigma \in \Sigma$ and $t_1, \dots, t_n \in T_\Sigma \iff \sigma(t_1, \dots, t_n) \in T_\Sigma$

Definition 3. *tree-language [1]*

A tree language $L_{t\Sigma}$ over the alphabet Σ is defined as a subset of T_Σ :

$$L_{t\Sigma} \subseteq T_\Sigma$$

From that definition, we can see that T_Σ is already a tree-language. Next, we have to declare some terminology in the context of Σ – *trees*.

We can now define (Non-Deterministic) Finite Tree Automata (NFTA) for tree languages. One can get a good grasp of how they work if you consider them to be NFAs with the possibility to have multiple states in their transition rules.

Definition 4. *NFTA [2]*

A (Non-Deterministic) Finite Tree Automaton (NFTA) over the alphabet Σ of arity n is a tuple $A = (Q, \Sigma, Q_f, \Delta)$ where Q is a **finite set of states**, $Q_f \subseteq Q$ is a **finite set of final states**, and Δ is a **finite set of transition rules** of the type:

$$f(q_1, \dots, q_n) \rightarrow q_x$$

where $f \in \Sigma, q_x, q_1, \dots, q_n \in Q$

For constants, we write:

$$a \rightarrow q$$

where $a \in \Sigma, q \in Q$

Note: These rules transition into the initial states of a NFTA
(that's why we call them initial rules rather informally)

An automaton starts at the leaves and moves upward, inductively associating along a **run** each subterm to a state while reducing the tree via the transition rules.

For a tree $t' \in T_{\Sigma \cup Q}$ that is the result of applying a transition rule on a tree $t \in T_{\Sigma \cup Q}$ we write:

$$t \rightarrow_A t'$$

If zero or more transition rules are applied, we write:

$$t \rightarrow_A^* t'$$

If a given input tree t_{input} can be reduced to only a single state $q \in Q_f$, then it is accepted by A . The set of all input trees accepted by A is called the **Language of A** , which is denoted by L_A .

reflexive and transitive closure -> Language of an automaton

Our binary-boolean-expression NFTA can now be written as:

Example 1. binary-boolean-statement NFTA

$A = (Q, \Sigma, Q_f, \Delta)$

$\Sigma = \{or, and, not, true, false\}$

$Q = \{q_f, q_t\}$

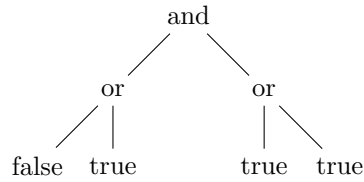
$Q_f = \{q_t\}$

$\Delta = \{false \rightarrow q_f, true \rightarrow q_t,$
 $and(q_t, q_t) \rightarrow q_t, and(q_t, q_f) \rightarrow q_f, and(q_f, q_t) \rightarrow q_f, and(q_f, q_f) \rightarrow q_f,$
 $or(q_t, q_t) \rightarrow q_t, or(q_t, q_f) \rightarrow q_t, or(q_f, q_t) \rightarrow q_t, or(q_f, q_f) \rightarrow q_f,$
 $not(q_f) \rightarrow q_t, not(q_t) \rightarrow q_f\}$

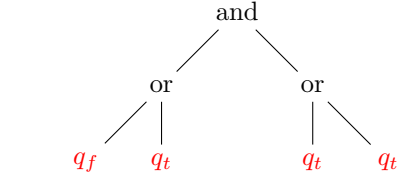
Running a tree automaton can be represented by a series of stages the tree is in. We will now show this by running the above automaton on the tree from the beginning of this chapter.

\rightarrow_A vll genauer definieren, dann hier: erst die Formale Version und dann das Beispiel (abgeschnitten auch)

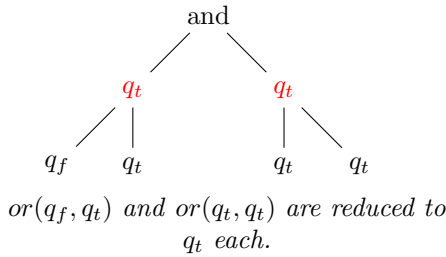
Example 2. running a NFTA



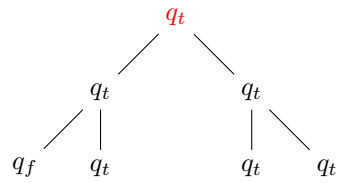
The start.



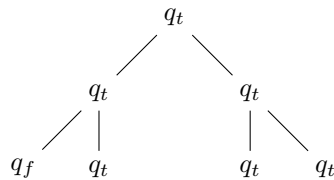
false and true, according to our initial rules, are reduced to q_f and q_t .



$or(q_f, q_t)$ and $or(q_t, q_t)$ are reduced to q_t each.



$and(q_t, q_t)$ is reduced to q_t .



The final stage.

In this graphical representation we kept the states after reducing them. However, this is not needed for running an automaton like the formal version of this run shows:

$$\begin{aligned} & \text{and}(\text{or}(\text{false}, \text{true}), \text{or}(\text{true}, \text{true})) \rightarrow_A^* \text{and}(\text{or}(q_f, q_t), \text{or}(q_t, q_t)) \\ & \rightarrow_A^* \text{and}(q_t, q_t) \rightarrow_A^* q_t \end{aligned}$$

Since the tree could be reduced to the accepting state $q_t \in Q_f$, we can say that A accepts w and therefore the tree w is in the language L_A recognized by the automaton.

Determinization

Non Deterministic Finite Tree Automata (NFTA) can be determinized just like Non Deterministic Automata (NFA) in the word case. By knowing that there exists a DFTA for every NFTA, definitions, proofs and algorithms become much easier, since we don't have to take special care of the properties of NFTAs. We will now take a look at how this is done. But first we have to define formally, what being deterministic means in the context of FTAs.

Definition 5. *Deterministic Finite Tree Automaton*

A tree automaton with no two rules of the type:

$$\begin{aligned} f(q_1, \dots, q_n) &\rightarrow q_x \\ f(q_1, \dots, q_n) &\rightarrow q_y \\ &\text{with } q_x \neq q_y \end{aligned}$$

or

$$\begin{aligned} \epsilon(q_1, \dots, q_n) &\rightarrow q_x \\ &\text{(state changes, even though no actual symbol is read)} \end{aligned}$$

with $n \geq 0, q_x, q_y, q_1, \dots, q_n \in Q, q_x \neq q_y, f \in \Sigma$ is called a **Deterministic Finite Tree Automaton (DFTA)**.

Similar to the algorithm for Determinization in the word case, there exists a power set construction algorithm for determinizing Tree Automata.

[powerset benutzen hier](#)

Definition 6. *Algorithm DET for Tree Automata [2]*

Note: statesOf(x) returns the set of states that contributed to the creation of the state x, while state(X) returns a state representing all states in the set X.

Data: NFTA $A = (Q, \Sigma, Q_f, \Delta)$

$Q_d := \emptyset$

$\Delta_d := \emptyset$

while Δ_d grew last cycle **do**

$f(q_1, \dots, q_n) \in \Delta$

$s_1, \dots, s_n \in Q_d$

/* meta-state representing the set of reachable states */

$s := \text{state}(\{q \in Q \mid q_1 \in \text{statesOf}(s_1), \dots, q_n \in$

$\text{statesOf}(s_n), f(q_1, \dots, q_n) \rightarrow q \in \Delta\})$

$Q_d := Q_d \cup \{s\}$

$\Delta_d := \Delta_d \cup f(s_1, \dots, s_n) \rightarrow s$

end

$Q_{fd} := \{s \in Q_d \mid \{s\} \cap Q_f \neq \emptyset\}$

Result: DFTA $A_d = (Q_d, \Sigma, Q_{fd}, \Delta_d)$

It is easy to see that the algorithm produces a deterministic automaton A_d as we are automatically constructing meta-states for all reachable states and therefore eliminating all possible non-deterministic behaviour. However, we still have to prove $L(A) = L(A_d)$. For this, we have to show that the meta-states $s \in Q_d$ are "built correctly", or in formal terms:

$$\text{For any tree } t : t \rightarrow_{A_d}^* s \iff s = \text{state}(\{q \in Q \mid t \rightarrow_A^* q\})$$

Proof. $L(A) = L(A_d)$ (Correctness of DET) [2]

This proof is done via an induction over the structure of the symbols in Σ .

- **Base case:** For any tree $t = a \in \Sigma$ we take a look at the rule $a \rightarrow q(a)$. Because of the way we defined s as the meta-state representing the set of all reachable states in a given situation this is inherently correct.
- **induction step:** $t = f(q_1, \dots, q_n)$

- 1.: $t \rightarrow_{A_d}^* s \Rightarrow (s = \text{state}(\{q \in Q \mid t \rightarrow_A^* q\}))$

Supposing $t \rightarrow_{A_d}^* f(s_1, \dots, s_n) \rightarrow_{A_d} s$, by induction hypothesis, for each $i \in 1, \dots, n$, we can see $s_i = \text{state}(\{q \in Q \mid q_i \rightarrow_A^* q\})$.

Because states $s_i \in Q_d$, rules $f(s_1, \dots, s_n) \rightarrow s \in \Delta_d$ are added by the determinization algorithm and $s := \text{state}(\{q \in Q \mid q_1 \in \text{statesOf}(s_1), \dots, q_n \in \text{statesOf}(s_n), f(q_1, \dots, q_n) \rightarrow q \in \Delta\})$, we learn $s = \text{state}(\{q \in Q \mid t \rightarrow_A^* q\})$.

- 2.: $s = \text{state}(\{q \in Q \mid t \rightarrow_A^* q\}) \Rightarrow t \rightarrow_{A_d}^* s$

Considering $s = \text{state}(\{q \in Q \mid f(q_1, \dots, q_n) \rightarrow_A^* q\})$ with state sets S_i defined as $S_i := \{q \in Q \mid q_i \rightarrow_A^* q\}$, by induction hypothesis for each $i \in \{1, \dots, n\}$ we know $q_i \rightarrow_{A_d}^* s_i, s_i = \text{state}(S_i)$. Thus $s = \text{state}(\{q \in Q \mid q_1 \in S_1, \dots, q_n \in S_n, f(q_1, \dots, q_n) \rightarrow q \in \Delta\})$.

By the definition of Δ_d in the determinization algorithm, $f(s_1, \dots, s_n) \in \Delta_d$ and thus $t \rightarrow_{A_d}^* s$.

□

Following is an example of how a NFTA can be determinized with this algorithm.

Beispiel ohne epsilon

Example 3. Running the DET algorithm consider a non deterministic FTA given like this:

$$\begin{aligned}
A &= (Q, \Sigma, Q_f, \Delta) \\
\Sigma &= \{ul, li, text, empty\} \\
Q &= \{q_{ul}, q_{li1}, q_{li2}, q_{text}, q_{empty}\} \\
Q_f &= \{q_{ul}\} \\
\Delta &= \{ul(q_{li1}, q_{li2}) \rightarrow q_{ul}, ul(q_{li2}, q_{li1}) \rightarrow q_{ul}, \\
&\quad li(q_{text}) \rightarrow q_{li1}, li(q_{text}) \rightarrow q_{li2}, \\
&\quad text \rightarrow q_{text}, empty \rightarrow q_{empty}, \\
&\quad \epsilon(q_{empty}) \rightarrow q_{text}\}
\end{aligned}$$

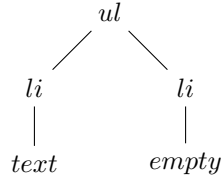
This recognizes all trees that represent unordered lists (ul) in HTML notation, which contain 2 list items (li):

```

<ul>
  <li>text</li>
  <li>empty</li>
</ul>

```

Or as a tree input:



If we start determinizing with the rules containing no state and then go "up in the hierarchy" and generate all the states on-the-fly, we get these new rules:

$$\begin{aligned}
text &\rightarrow state(\{q_{text}\}) \\
empty &\rightarrow state(\{q_{text}, q_{empty}\}) \\
li(state(\{q_{text}\})) &\rightarrow state(\{q_{li1}, q_{li2}\}) \\
li(state(\{q_{text}, q_{empty}\})) &\rightarrow state(\{q_{li1}, q_{li2}\}) \\
ul(state(\{q_{li1}, q_{li2}\}), state(\{q_{li1}, q_{li2}\})) &\rightarrow state(\{q_{ul}\})
\end{aligned}$$

And the set of final states is $Q_{fd} = \{state(\{q_{ul}\})\}$.

As we can see, there is no ϵ -rule left and we don't have to choose which rule to apply when reading

Minimization

Now that we can obtain a DFTA for each NFTA, we can take a look at how we can minimize these newly determinized automata.

Just like in the word case there exists a Myhill-Nerode theorem for Finite Tree Automata. But before we can use it, we have to define **Contexts**, **Congruence** and \equiv_L .

For the definition of a **Context** it is convenient to define a **Slot** first.

Definition 7. Slot *(is this definition sufficient?)*

A **Slot** $s \in S, S \cap \Sigma = \emptyset$ is a special token, that, if found in a tree $t_1 \in T_{\Sigma \cup S}$, can be replaced by any tree $t_2 \in T_{\Sigma \cup S}$ (t_2 can contains **slots** as well).

As an abstract representation, a tree with a slot is often drawn as a triangle with a marker for every slot:

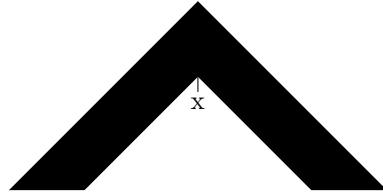


Fig. 1: Tree with the slot x [3]

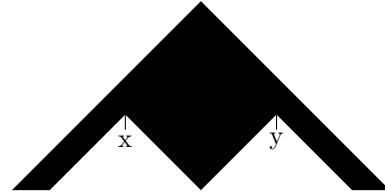


Fig. 2: Tree with the slots x and y [3]

Defining a Context is straightforward now.

Definition 8. Context [2][3]

A tree with slots is called a **Context**. Furthermore, if C is a context with slots $s_1, \dots, s_n \in S$, then $C[t_1, \dots, t_n], t_1, \dots, t_n \in T_\Omega$ is known as a **context application**, with the slots s_i being replaced by (sub-)trees $t_i \in T_\Omega, T_\Omega \supseteq T_{\Sigma \cup S}$.

Note: T_Ω **can** contain new slots.



Fig. 3: Context application [3]

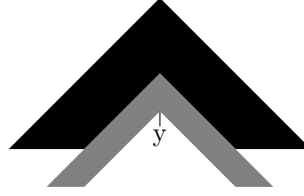


Fig. 4: Context application with a new context [3]

Definition 9. *Congruence [2]*

An equivalence relation \equiv on T_Σ is a **congruence** on T_Σ if for every $f \in \Sigma$ with n arguments applies:

$$T_\Sigma \ni u_i \equiv w_i \in T_\Sigma, 1 \leq i \leq n \Rightarrow f(u_1, \dots, u_n) \equiv f(w_1, \dots, w_n)$$

of \equiv -classes is finite $\Rightarrow \equiv$ is of **finite index**.

Additionally a congruence is an equivalence relation closed under context. This means that for any $C \in T_{\Sigma \cup V}$, if $u \equiv w \Rightarrow C[u] \equiv C[w]$.

Definition 10. \equiv_L [2]

For any given tree language $L \in T_\Sigma$, we define the congruence \equiv_L on T_Σ by: $T_\Sigma \ni u \equiv_L w \in T_\Sigma$, if for all Contexts $C \in T_{\Sigma \cup V}$ applies:

$$C[u] \in L \iff C[v] \in L$$

For the sake of easier proofs, we consider all following DFTAs to be **complete** and **reduced**.

Definition 11. *Completeness and reduction [5]*

A FTA A is **complete** if there is at least one transition rule available for every possible symbol-states combination. A state q is **accessible** if there exists a tree t such that $t \rightarrow_A^* q$. A NFTA is **reduced** if all its states are accessible.

Note: All examples for Finite Tree Automata given in this paper are supposed to be complete and reduced. However, we do not add a capturing state for all impossible symbol-state combinations for the sake of simplicity. Let $q_c \in Q$ be the capturing state, then every transition rule that contained it state on the left side look like $f(\dots, q_c, \dots) \rightarrow q_c$. This means, that once the capturing state is reached, there is no way of getting to a different state anymore.

We can now give the Myhill-Nerode theorem.

Theorem 1. *Myhill-Nerode [2]*

These statements are equivalent:

- (i) L is a regular tree language
- (ii) L is the union of some congruence classes of finite index
- (iii) the relation \equiv_L is a congruence of finite index

Proof.

- (i) \Rightarrow (ii): Assume that the tree language L is recognized by some complete DFTA $A = (Q, \Sigma, Q_f, \delta)$ with δ being a transition function (i). Let us consider the relation \equiv_A defined on T_Σ by: $T_\Sigma \ni u \equiv v \in T_\Sigma$, if $\delta(u) = \delta(v)$. Since we know that Q only has a finite amount of states in it and the number of equivalence classes may at most be equal to the size of Q , we can deduce that \equiv_A is a congruence of finite index (ii).
- (ii) \Rightarrow (iii): By denoting the congruence of finite index as \cong and assuming that $T_\Sigma \ni u \cong v \in T_\Sigma$, it can be proven that $C[u] \cong C[v]$ for all contexts $C \in T_{\Sigma \cup V}$ by an easy induction on the structure of terms. Since L is the union of some equivalence classes of the congruence of finite index \cong (ii), we have $C[u] \in L \iff C[v] \in L$. Therefore we know that $u \equiv_L v$ and that \equiv_L contains the equivalence class of u in \cong . Furthermore we now know that $index(\equiv_L) \leq index(\cong) \Rightarrow index(\equiv_L)$ is finite (iii).
- (iii) \Rightarrow (i): By representing the set of equivalence classes of \equiv_L (iii) as the finite set of states Q_{min} with $|Q_{min}| = |\equiv_L|$, we know that every equivalence class has its own state. By denoting the equivalence class of a term $u \in T_\Sigma$ as $[u]$ we define the transition function δ_{min} for every $f \in \Sigma$ with n arguments as:

$$\delta_{min}(f, [u_1], \dots, [u_n]) = [f(u_1, \dots, u_n)]$$

The definition of δ is consistent because \equiv_L is a congruence. With $Q_{min_f} := \{[u] | u \in L\}$ the resulting DFTA $A_{min} := (Q_{min}, \Sigma, Q_{min_f}, \delta_{min})$ recognizes the tree language L (i). \square

As a consequence of this theorem we can deduce the following:

Corollary 1. [2]

The minimum DFTA recognizing a tree language L is unique up to renaming the states and is given by A_{min} in the proof of the Myhill-Nerode Theorem.

This means that we can minimize a tree automaton by computing the congruence classes of the language it recognizes. But before we can put this to use, we have to prove the corollary first.

Proof. [2]

Assume that L is recognized by some DFTA $A = (Q, \Sigma, Q_f, \delta)$. Then the relation \equiv_A is a refinement of \equiv_L with $index(\equiv_A) \geq index(\equiv_L)$, thus $|Q| \geq |Q_{min}|$. We know that A is reduced (all states are accesible), because otherwise a state could be removed contradicting to the definition of \equiv_A . Let $q \in Q$ and $u \in T_\Sigma$, such that $\delta(u) = q$. Then the state q can be consistently identified with the state $\delta_{min}(u)$, since δ is a refinement of δ_{min} and we can see that every state $q \in Q$ has a corresponding state $q_{min} \in Q_{min}$. \square

By using this Corollary and the construction given in the Myhill-Nerode theorem we can deduce an algorithm to minimize Deterministic Finite Tree Automata:

Definition 12. *Algorithm MIN for Tree Automata* [4]

Data: complete and reduced DFTA $A = (Q, \Sigma, Q_f, \Delta)$

Set $P = \{(q_f, q) \mid q_f \in Q_f, q \in Q \setminus Q_f\}$

Set $P' = P$

while $P' \neq P$ **do**

$P = P'$

$\forall p_1, p_2, p_3 \in Q, p_1 \neq p_2, p_1 \neq p_3, p_2 \neq p_3,$

define $p_1 P' p_2 \iff$

/* could distinguish in the last cycle */

1. $p_1 P p_2$ or

/* can distinguish p_1 from p_2 , with: */

2. $\exists f \in \Sigma$ with n arguments, $\exists q_1, \dots, q_{i-1}, q_{i+1}, \dots, q_n \in Q,$

$r_1 P r_2, r_1, r_2 \in Q,$ where:

$f(q_1, \dots, q_{i-1}, p_1, q_{i+1}, \dots, q_n) \rightarrow r_1$ and

$f(q_1, \dots, q_{i-1}, p_2, q_{i+1}, \dots, q_n) \rightarrow r_2$

(Note: this works for multiple occurences of p_1 and p_2 as well,
see the example on the next page)

end

Q_{min} = set of equivalence classes of P

$\Delta_{min} = \{f([q_1], \dots, [q_n]) \rightarrow [q] \mid f(q_1, \dots, q_n) \rightarrow q \in \Delta\}$

$Q_{f_{min}} = \{[q] \mid q \in Q_f\}$

Result: complete, reduced and minimal DFTA

$A_{min} = (Q_{min}, \Sigma, Q_{f_{min}}, \Delta_{min})$

While we are not giving a complete proof for this algorithm, we can at least go over why it is correct for a given language $L = L(A)$ and the automaton $A = (Q, \Sigma, Q_f, \Delta)$ [2]:

- In the while loop we are marking all tuples p_1, p_2 as distinguishable when they are added to the relation
- To get the equivalence classes for Q_{min} , we are merging all pairs of indistinguishable states to a new one representing both. After having done this incrementally for all not marked combinations, these "artificial" states have to be distinguishable to all other states and Q_{min} must therefore be minimal. This can easily be proven correct by contradiction.
- It isn't hard to see that the construction of $Q_{f_{min}}$ and Δ_{min} is correct.

Example 4. Running the MIN algorithm

After cleaning up the Automaton of our previous unordered list example, one might already see that it isn't minimal yet:

$$\begin{aligned}
A &= (Q, \Sigma, Q_f, \Delta) \quad \Sigma = \{ul, li, text, empty\} \\
Q &= \{q_{ul}, \mathbf{q}_{text}, \mathbf{q}_{text2}, q_{li}\} \\
Q_f &= \{q_{ul}\} \\
\Delta &= \{text \rightarrow q_{text}, \\
&\quad empty \rightarrow q_{text2}, \\
&\quad \mathbf{li}(\mathbf{q}_{text}) \rightarrow \mathbf{q}_{li}, \\
&\quad \mathbf{li}(\mathbf{q}_{text2}) \rightarrow \mathbf{q}_{li}, \\
&\quad ul(q_{li}, q_{li}) \rightarrow q_{ul}\}
\end{aligned}$$

While we should be able to fix this by hand, we are now using the MIN algorithm to minimize this automaton. In the following table, we are marking all tuples p_1, p_2 that are distinguishable in that cycle by the index of that cycle, so we can see the process in action.

	q_{text}	q_{text2}	q_{li}	q_{ul}
q_{text}	-	-	-	-
q_{text2}	(merge)	-	-	-
q_{li}	1	1	-	-
q_{ul}	0	0	0	-

As predicted q_{text} and q_{text2} have to be merged in order to minimize A . The resulting automaton is:

$$\begin{aligned}
A &= (Q, \Sigma, Q_f, \Delta) \quad \Sigma = \{ul, li, text, empty\} \\
Q &= \{q_{ul}, \mathbf{q_{text_{1\&2}}}, q_{li}\} \\
Q_f &= \{q_{ul}\} \\
\Delta &= \{text \rightarrow \mathbf{q_{text_{1\&2}}}, \\
&\quad empty \rightarrow \mathbf{q_{text_{1\&2}}}, \\
&\quad \mathbf{li(q_{text_{1\&2}})} \rightarrow \mathbf{qli}, \\
&\quad ul(q_{li}, q_{li}) \rightarrow q_{ul}\}
\end{aligned}$$

References

1. Automata theory for XML researchers, Frank Neven, University of Limburg, frank.neven@ucl.ac.be, <http://homepages.inf.ed.ac.uk/libkin/dbtheory/frank.pdf>, 03/11/2015
2. Tree Automata and Techniques, Hubert Comon et. al, Pages 19-39
3. Automata and Logic on Trees, Wim Martens, Stijn Vansummen, <http://lrb.cs.uni-dortmund.de/~martens/data/essli07/lecture01.pdf>, 03/17/2015
4. Automata and Logic on Trees, Wim Martens, Stijn Vansummen, <http://lrb.cs.uni-dortmund.de/~martens/data/essli07/lecture02.pdf>, 03/21/2015
5. http://en.wikipedia.org/wiki/Tree_automaton, 03/20/2015