CSC 2538
Topics in DB: Foundations of XML
Lecture 4

Prof. Leonid Libkin
notes by Solmaz Kolahi

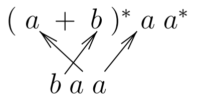
October 21, 2005

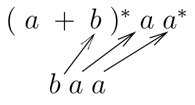
1 1-Unambiguous Regular Expressions

Elements in XML DTDs are defined using regular expressions. As a W3C standard, only 1-unambiguous regular expressions are allowed in DTDs. As we will see later, there is an efficient membership algorithm for this class of regular expressions, which facilitates the process of validating XML documents with respect to DTDs.

A regular expression is 1-unambiguous if every symbol in any input string can be uniquely matched to an occurrence of the symbol in the regular expression, without looking ahead in the string. That is, every string can be parsed with a single-symbol lookahead.

As an example, consider regular expression $e = (a + b)^*aa^*$ and string $s = baa$ in the language of e. String s can be parsed in two different ways as shown below. Regular expression e is therefore not 1-unambiguous.





To define this concept more formally, we mark every occurrence of symbols in the regular expression with subscripts, such that different occurrences of the same symbol have different subscripts. For instance, a *marking* of regular expression $e = (a + b)^*aa^*$ is $e_m = (a_1 + b_1)^*a_2a_3^*$. Given an alphabet Σ, we denote the subscripted set of symbols in Σ by Σ_m. For every symbol $x \in \Sigma_m$, x° denotes the corresponding unmarked symbol in Σ and is obtained by simply removing the subscript of x.

Let e be a regular expression over Σ and e_m be a marking of e over Σ_m. Then:

Definition 1 *Regular expression e is 1-unambiguous if for every three strings $u, v, w \in \Sigma_m^*$ and every two symbols $x, y \in \Sigma_m$ such that $uxv, uyw \in L(e_m)$, $x \neq y$ implies $x^\circ \neq y^\circ$.*

A regular language is 1-unambiguous if it is denoted by a 1-unambiguous regular expression. Note that the language denoted by regular expression $e = (a + b)^*aa^*$ is 1-unambiguous, because there is a 1-unambiguous regular expression $e' = b^*a(b^*a)^*$ such that $L(e) = L(e')$.

Given a regular expression e, one can construct a finite automaton $\mathcal{A}_e$ that accepts $L(e)$ using Glushkov construction. Then:

Theorem 1 *Regular expression e is 1-unambiguous if and only if $\mathcal{A}_e$ is deterministic.*

1.1 Glushkov Construction

Let L be a regular language over alphabet Σ. We define the following sets, as we need them later in the construction of Glushkov automaton.

- $first(L) = \{a \in \Sigma \mid \exists\, u \in \Sigma^*\ au \in L\}$.
- $last(L) = \{a \in \Sigma \mid \exists\, u \in \Sigma^*\ ua \in L\}$.
- $follow(L, a) = \{b \in \Sigma \mid \exists u, w \in \Sigma^*\ uabw \in L\}$, for every $a \in \Sigma$.

Let e be a regular expression and e_m be a marking of e. By $Sym(e_m)$ we denote the set of all symbols in e_m. For example, if $e = b^*a(b^*a)^*$ and $e_m = b_1^*a_1(b_2^*a_2)^*$, then $Sym(e_m) = \{a_1, a_2, b_1, b_2\}$.

Now the Glushkov automaton for e is defined as $\mathcal{A}_e = (Q, q_0, F, \delta)$, where

- $Q = Sym(e_m) \cup \{q_0\}$;

- $\delta(q_0, a) = \{x \mid x \in first(L(e_m)), x^\circ = a\}$, for every $a \in \Sigma$;
- $\delta(y, a) = \{x \mid x \in follow(L(e_m), y), x^\circ = a\}$, for every $y \in Sym(e_m)$, $a \in \Sigma$;
- $F = \begin{cases} last(L(e_m)) & if\ \epsilon \notin L(e), \\ last(L(e_m)) \cup \{q_0\} & if\ \epsilon \in L(e). \end{cases}$

The Glushkov automaton for regular expressions $e = (a + b)^*a$ is given below. Note that the automaton is not deterministic sine e is not 1-unambiguous.

$e_m = (a_1 + b_1)^* a_2$
$\mathcal{A}_e = (Q, q_0, F, \delta)$
$Q = \{q_0, a_1, b_1, a_2\}$
$F = \{a_2\}$

$\delta(q_0, a) = \{a_1, a_2\}$	$\delta(a_1, a) = \{a_1, a_2\}$	$\delta(b_1, a) = \{a_1, a_2\}$	$\delta(a_2, a) = \{\}$
$\delta(q_0, b) = \{b_1\}$	$\delta(a_1, b) = \{b_1\}$	$\delta(b_1, b) = \{b_1\}$	$\delta(a_2, b) = \{\}$

The Glushkov automaton for regular expressions $e' = b^*a(b^*a)^*$ is given below. Note that the automaton is deterministic sine e is 1-unambiguous.

$e'_m = b_1^* a_1 (b_2^* a_2)^*$
$\mathcal{A}_e = (Q, q_0, F, \delta)$
$Q = \{q_0, a_1, b_1, b_2, a_2\}$
$F = \{a_1, a_2\}$

$\delta(q_0, a) = \{a_1\}$	$\delta(a_1, a) = \{a_2\}$	$\delta(b_1, a) = \{a_1\}$	$\delta(b_2, a) = \{a_2\}$	$\delta(a_2, a) = \{a_2\}$
$\delta(q_0, b) = \{b_1\}$	$\delta(a_1, b) = \{b_2\}$	$\delta(b_1, b) = \{b_1\}$	$\delta(b_2, b) = \{b_2\}$	$\delta(a_2, b) = \{b_2\}$

2 Unary (Node-Selecting) Queries over Unranked Trees

In this lecture, by a *unary query* we refer to a function that maps every tree to a set of its nodes that satisfy some property. These queries are important in the context of XML documents since we are often looking for subdocuments (subtrees) that satisfy a certain pattern.

Each query $\mathcal{Q}$ can be defined using a formula φ in FO or MSO. Recall that a tree T is defined as a pair (D, λ), where D is a tree domain and λ is a labeling function over the nodes in D. Then given a tree $T = (D, \lambda)$,

$$\mathcal{Q}(T) = \{s \mid s \in D,\ T \models \varphi(s)\}.$$

Now we want to know how tree automata can be used to compute such queries.

2.1 Nondeterministic Unranked Query Automata

One way to define query automata for unranked trees is to add a *selecting set* to nondeterministic tree automata. A *nondeterministic unranked query automaton* over alphabet Σ is defined as

$$\mathcal{A} = (Q, F, \delta, S),$$

where (Q, F, δ) is a nondeterministic unranked tree automata, and $S \subseteq Q \times \Sigma$ is a selecting set.

Such an automata defines two unary queries $\mathcal{Q}^{\exists}_{\mathcal{A}}$ and $\mathcal{Q}^{\forall}_{\mathcal{A}}$. Given an unranked tree $T = (D, \lambda)$, the semantics of these queries on T are defined as follows:

- For every $s \in D$, $s \in \mathcal{Q}^{\exists}_{\mathcal{A}}(T) \iff$ there exists an accepting run $\rho_{\mathcal{A}} : D \to Q$ on T such that $(\rho_{\mathcal{A}}(s), \lambda(s)) \in S$.

- For every $s \in D$, $s \in \mathcal{Q}^{\forall}_{\mathcal{A}}(T) \iff$ for every accepting run $\rho_{\mathcal{A}} : D \to Q$ on T, $(\rho_{\mathcal{A}}(s), \lambda(s)) \in S$.

Theorem 2 *For a unary query $\mathcal{Q}$ on unranked trees, the following are equivalent:*

1. *$\mathcal{Q}$ is definable in MSO.*
2. *$\mathcal{Q}$ is of the form $\mathcal{Q}^{\exists}_{\mathcal{A}}$ for some query automaton $\mathcal{A}$.*
3. *$\mathcal{Q}$ is of the form $\mathcal{Q}^{\forall}_{\mathcal{A}}$ for some query automaton $\mathcal{A}$.*

2.2 Deterministic Query Automata

To define deterministic ranked query automata, we extend the definition of *two-way deterministic tree automata* over ranked trees by adding a selecting set.

2.2.1 Two-Way Deterministic Tree Automata over Binary Trees

A two-way deterministic tree automaton (2DTA) over the alphabet Σ is defined as

$$\mathcal{A} = (Q, q_0, F, \delta_{\downarrow}, \delta_{\uparrow}, \delta_{root}, \delta_{leaf}),$$

where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $q_0 \in Q$ is the initial state. There are two disjoint subsets D and U of $Q \times \Sigma$ such that:

- $\delta_{\downarrow} : D \to Q \times Q$ is the transition function for down transitions.
- $\delta_{\uparrow} : U \times U \to Q$ is the transition function for up transitions.
- $\delta_{root} : Q \times \Sigma \to Q$ is the transition function for the root.
- $\delta_{leaf} : Q \times \Sigma \to Q$ is the transition function for the leaves.

Note that U and D are disjoint to ensure that for a given node of the tree at a particular state, there is only one transition. To define the behavior of this automaton, we need to define some notations. A *cut* in a tree $T = (D, \lambda)$ is a subset C of D such that C contains exactly one node of each path from the root to a leaf. A *configuration* of automaton $\mathcal{A}$ on T is a mapping $c : C \to Q$.

The automaton $\mathcal{A}$ operating on T makes a *transition* between configurations $c_1 : C_1 \to Q$ and $c_2 : C_2 \to Q$, written as $c_1 \to c_2$, if it makes one of the following transitions:

1. *Down transition*: if there is a node $s \in D$ such that
   - s is in the cut C_1,
   - $C_2 = C_1 - \{s\} \cup \{s0, s1\}$,
   - $(c_2(s0), c_2(s1)) = \delta_{\downarrow}(c_1(s), \lambda(s))$, and c_2 is identical to c_1 on $C_1 \cap C_2$.
2. *Up transition*: if there is a node $s \in D$ such that
   - both children of s are in the cut C_1 $(s0, s1 \in C_1)$,
   - $C_2 = C_1 - \{s0, s1\} \cup \{s\}$,
   - $c_2(s) = \delta_{\uparrow}((q_1, \lambda(s0)), (q_2, \lambda(s1)))$, where $q_1 = c_1(s0)$ and $q_2 = c_1(s1)$, and c_2 is identical to c_1 on $C_1 \cap C_2$.
3. *Root transition*: if

- $C_1 = \{root\}$; cut only contains the root of T,
- $C_2 = C_1$,
- $C_2(root) = \delta_{root}(c_1(root), \lambda(root))$, and c_2 is identical to c_1 on $C_1 - \{root\}$.

4. *Leaf transition*: if there is a leaf node $s \in D$ such that
   - s is in the cut C_1,
   - $C_2 = C_1$,
   - $c_2(s) = \delta_{leaf}(c_1(s), \lambda(s))$, and c_2 is identical to c_1 on $C_1 - \{s\}$.

The initial configuration is $c : \{root\} \to q_0$. A configuration is *accepting* if it is of the form $c : \{root\} \to q$ such that $q \in F$. A *run* is a sequence of configurations $c_1, \ldots, c_m$, $m \geq 1$, such that c_1 is the initial configuration, and $c_i \to c_{i+1}$ for $i \in [1, m-1]$. A run is *accepting* if c_m is accepting and there is no configuration c such that $c_m \to c$.

There can be more than one run for a given tree. However, for every node, the sequence of states in which the node is visited is the same in all these runs. This is because a node labeled with a certain state cannot make an up transition in one run and a down transition in another run. Therefore, the behavior of the automaton is considered as deterministic, and as of now, we can use the term *the run* of $\mathcal{A}$ on a tree.

A 2DTA $\mathcal{A}$ *accepts* a tree T if the run of $\mathcal{A}$ on T is accepting. For $\mathcal{A}$ to accept a tree T it should start at the root and return there. Note that $\mathcal{A}$ can run for ever on a tree T. In this case the run is not finite and therefore not accepting. It is however decidable to determine whether a 2DTA halts on an input tree. We only consider automata that always terminate on every input tree.

2.2.2 Query Automata

A deterministic ranked query automaton is defined as

$$\mathcal{A} = (Q, q_0, F, \delta_\downarrow, \delta_\uparrow, \delta_{root}, \delta_{leaf}, S \subseteq Q \times \Sigma),$$

where $(Q, q_0, F, \delta_\downarrow, \delta_\uparrow, \delta_{root}, \delta_{leaf})$ is a two-way deterministic ranked tree automata, and S is a selecting set.

Given a tree $T = (D, \lambda)$, we say $\mathcal{A}$ *selects* a node $s \in D$ if the run $c_1, \ldots, c_m$ of $\mathcal{A}$ on T is accepting, and there is a configuration $c_i : C \to Q$, $i \in [1, m]$, such that $s \in C$ and $(c(s), \lambda(s)) \in S$. The *query computed by* $\mathcal{A}$ is defined as

$$\mathcal{A}(T) = \{s \mid s \in D,\ \mathcal{A} \text{ selects } s\}.$$

Theorem 3 *A query is computable by a ranked query automaton if and only if it is definable in MSO.*

To define deterministic query automaton for unranked trees, we need to add *stay transitions* in order to obtain the equivalence with MSO. You can refer to [1] for more details.

Given an MSO query φ, the size of the query automaton that computes φ is not elementary. In the next lecture, we will talk about logics equivalent to MSO in expressive power, for which query computation can be done more efficiently.

References

[1] Frank Neven and Thomas Schwentick, Query automata over finite trees, *Theor. Comput. Sci.*, 275(1-2): 633–674, 2002.