Automata for Analyzing and Querying Compressed Documents

Barbara FILA, LIFO, Orléans (Fr.)
Siva ANANTHARAMAN, LIFO, Orléans (Fr.)

Rapport N° 2006-03
Automata for Analyzing and Querying
Compressed Documents

Barbara Fila, Siva Anantharaman
LIFO - Université d’Orléans (France),
e-mail: {fila, siva}@univ-orleans.fr

Abstract. In a first part of this work, tree/dag automata are defined as
extensions of (unranked) tree automata which can run indifferently on
trees or dags; they can thus serve as tools for analyzing or querying any
semi-structured document, whether or not given in a compressed format.
In a second part of the work, we present a method for evaluating posi-
tive unary queries, expressed in terms of Core XPath axes, on any dag
t representing an XML document possibly given in a compressed form;
the evaluation is done directly on t, without unfolding it into a tree. To
each Core XPath query of a certain basic type, we associate a word au-
tomaton; these automata run on the graph of dependency between the
non-terminals of the minimal straightline regular tree grammar associ-
ated to the given dag t, or along complete sibling chains in this grammar.
Any given positive Core XPath query can be decomposed into queries of
the basic type, and the answer to the query, on the dag t, can then be
expressed as a sub-dag of t whose nodes are suitably labeled under the
runs of such automata.

Keywords: Tree automata, Tree grammars, Dags, XML, Core XPath.

1 Introduction

Several algorithms have been optimized in the past, by using structures over
dags instead of over trees. Tree automata are widely used for querying XML
documents (e.g., [8, 9, 15, 16]); on the other hand, the notion of a compressed
XML document has been introduced in [2, 7, 12], and a possible advantage of
using dag structures for the manipulation of such documents has been brought
out in [12]. It is legitimate then to investigate the possibility of using automata
over dags instead of over trees, for querying compressed XML documents.

Dag automata (DA) were first introduced and studied in [5]; a DA was defined
there as a natural extension of tree automaton, i.e. as a bottom-up tree automa-
ton running on dags; and the language of a DA was defined as the set of dags
that get accepted under (bottom-up) runs, defined in the usual sense; the empti-
ness problem for DAs was shown there to be NP-complete, and the membership
problem proved to be in NP; but the problem of stability under complementation
of the class of dag automata –closely linked with that of determinization– was
left open. These two issues have since been settled negatively in [1]: the reason
is that the set of all terms (trees) represented by the set of dags accepted by a
non-deterministic DA is not necessarily a regular tree language; a consequence is
that the class of tree languages recognized by DAs (as sets of accepted dags) is a
strict superclass of the class of regular tree languages. It is well-known however,
that answers to MSO-definable queries on (semi-)structured trees form regular
tree languages ([18]); it is thus necessary to define the languages of DAs in a
manner different from that of [5, 1], if they are to serve as tools for analyzing
and querying a document, independently of whether it is given in a (partially or
fully) compressed format, or as a tree. Our first aim in this work is therefore to
redefine the notion of the language of a DA suitably, with such an objective.
For achieving that, we first present (in Section 2) the notion of a compressed
document as a \textit{tree/dag} (\textit{trdag}, for short), designating a directed acyclic
graph that may be partially or fully compressed. The terminology \textit{trdag} has been
chosen to distinguish it from that of \textit{tdag} employed in \cite{1}; this latter term will
be employed in this paper when referring to a fully compressed dag. A Tree/Dag
automaton (TDA, for short) is then defined as an automaton which runs on
trdags. The essential differences with the DAs of \cite{1} are the following: (i) our
TDAs can be unranked, and (ii) although the transition rules of a TDA look
quite like those of the DAs in \cite{1}, or those of TAs, a run of a TDA on any given
\textit{trdag} \(t\) will carry with it not only assignments of states to the nodes of \(t\), but
also to the edges of \(t\); runs will be so defined that a TDA accepts any given
\textit{trdag} \(t\) if and only if it accepts the tree \(\hat{t}\) obtained by uncompressing \(t\), as a tree
automaton running on the tree \(\hat{t}\), in the usual sense.

In the second part of the paper, we present an approach based on word
automata for evaluating queries on trdags that represent XML documents in
a partially or fully compressed format; the terms ‘\textit{trdag}’ and ‘\textit{document}’ will
therefore be considered synonymous in the sequel. Any given \textit{trdag} \(t\) is first seen
as equivalent to a minimal straightline regular tree grammar \(L_t\), that one can
naturally associate with \(t\), cf. e.g., \cite{3,4}. From the grammar \(L_t\), we construct
the graph of dependency \(D_t\) between its non-terminals, and also the \textit{chiblings}
(linear graphs formed of complete chains of sibling non-terminals) of \(L_t\). The
word automata that we build below will run on \(D_t\) or the chiblings of \(L_t\), rather
than on the document \(t\) itself.

We shall only consider \textit{positive} unary queries expressed in terms of Core
XPath axes. (The view we adopt allows us to define the various axes of Core
XPath on compressed documents, in a manner which does not modify their
semantics on trees.) For evaluating any such query on any document (\textit{tdag}) \(t\),
we proceed as follows. We first break up the given query into basic sub-queries
of the form \(Q = //*[\text{axis}::\sigma]\) where \text{axis} is a Core XPath axis of a certain
type. To each such basic query \(Q\), we associate a word automaton \(A_Q\). The
automaton \(A_Q\) runs on the graph \(D_t\) when \text{axis} is non-sibling, and on the
chiblings of \(L_t\) when \text{axis} is a sibling axis. An essential point in our method is
that the runs of \(A_Q\) are guided by some well-defined semantics for the nodes
traversed, indicating whether the current node answers \(Q\), or is on a path leading
to some other node answering \(Q\). The automaton, though not deterministic, is
made effectively unambiguous by defining a suitable priority relation between its
transitions, based on the semantics. A basic query \(Q\) can then be evaluated in
one single top-down pass of \(A_Q\), under such an unambiguous run. An arbitrary
positive unary \text{Core XPath} query can be evaluated on \(t\) by combining the answers
to its various basic sub-queries, and its answer set is expressed as a sub-\textit{trdag}
of \(t\), whose nodes get labeled in conformity with the semantics. It is important
to note that the evaluation is performed on the \textit{given} \textit{tdag} \(t\); as such, on two
different \textit{tdags} corresponding to two different compressions of a same XML tree,
the answers obtained may \textit{not} be the same, in general.

The paper is structured as follows: Section 2 presents the notions of \textit{tdags},
and of Tree/Dag automata. In Section 3, we construct from any \textit{tdag} \(t\) its
normalized straightline regular tree grammar \(L_t\), as well as the dependency graph
\(D_t\) and the chiblings of \(L_t\); these will be seen as rooted labeled acyclic graphs
(\textit{rlags}, for short); the basic notions of Core XPath are also recalled. Section 4
is devoted to the construction of the word automata for any basic \text{Core XPath}
query, based on the semantics, and an illustrative example. In Section 5 we prove
that the runs of these automata, uniquely and effectively determined under a
maximal priority condition, generate the answers to the queries. Section 6 shows
how a non basic (composite, or imbricated) \text{Core XPath} query can be evaluated
in a stepwise fashion. In Section 7, we show how to refine our approach, so as to derive, from the answer for any given Core Xpath query \( Q \) on a trdag \( t \), the answer set for the same query \( Q \) on the tree-equivalent \( t \) of \( t \). In the appendices, we show how to translate the ‘usual’ Core XPath queries into one in ‘standard’ form on which our approach is applicable (the translation is done in linear time on the size of the given query); we also present a polynomial time algorithm for constructing the maximal priority run, for any basic query automaton over any given document (trdag), with a complexity bound of \( O(n^2) \), where \( n \) is the number of nodes of the trdag; the bound reduces to \( O(n^3) \) on trees where the relation \( Parents \) is trivial; a complete illustrative example, on a composite imbricated query, is given in the last appendix.

2 Tree/Dag Automata

Definition 1 A tree/dag (trdag for short) over a not necessarily ranked alphabet \( \Sigma \) is a rooted dag (directed acyclic graph) \( t = (Nodes(t), Edges(t)) \), where, for any node \( u \in Nodes(t) \):

- \( u \) has a name \( name_1(u) = name(u) \in \Sigma \);
- the edges going out of any node are ordered;
- and if \( name(u) \) is ranked, then the number of outgoing edges at \( u \) is the rank of \( name(u) \).

Given any node \( u \) on a trdag \( t \), the notion of the sub-trdag of \( t \) rooted at \( u \) is defined as usual, and denoted as \( t|_u \). If \( v \) is any node, \( \gamma(v) = u_1 \ldots u_n \) will denote the string of all its not necessarily distinct children nodes; for every \( 1 \leq i \leq n \), the \( i \)-th outgoing edge from \( v \) to its \( i \)-th child node \( u_i \in \gamma(v) \) will be denoted as \( e(v,i) \); we shall also write then \( v \xrightarrow{i} u_i \); the set of all outgoing (resp. incoming) edges at any node \( v \) will be denoted as \( Out_v(t) \), or \( Out_v \) (resp. \( In_v(t) \), or \( In_v \)); and for any node \( u \), we set: \( Parents(u) = \{ v \in Nodes(t) \mid u \) is a child of \( v \} \).

A trdag \( t \) will be said to be a tree iff for every node \( u \) on \( t \) other than the root, \( Parents(u) \) is a singleton. For any trdag \( t \), we define the set \( Pos(t) \) as the set of all the positions \( posv(u) \) of all its nodes \( u \), these being defined recursively, as follows: if \( u \) is the root node on \( t \), then \( posv(u) = \epsilon \); otherwise, \( posv(u) = \{ \alpha.i \mid \alpha \in posv(v), v \) is a parent of \( u, u \) is an \( i \)-th child of \( v \} \). The set \( Pos(t) \) consists of (some of the) words over natural integers. To any edge \( e : u \xrightarrow{i} v \) on a trdag \( t \), is naturally associated the subset \( posv(e) = posv(u).i \) of \( Pos(t) \).

The function \( name_1 \) is extended naturally to the positions in \( Pos(t) \) as follows: for every \( u \in Nodes(t) \) and \( \alpha \in posv(u) \), we set \( name_1(\alpha) = name_1(u) \). Given a trdag \( t \), we define its tree-equivalent as a tree \( t \) such that: \( Pos(t) = Pos(t) \), and for every \( \alpha \in Pos(t) \) we have \( name_1(\alpha) = name_1(\alpha) \). It is immediate that \( t \) is uniquely determined, up to a tree isomorphism; it can actually be constructed canonically (cf. [7]), by taking for nodes the set \( Pos(t) \), and for directed edges the set \( \{ (\alpha.\alpha.i) \mid \alpha.\alpha.i \in Pos(t) \} \), each node \( \alpha \) being named with \( name_1(\alpha) \). There is then a natural, name preserving, surjective map from \( Nodes(t) \) onto \( Nodes(t) \); it will be referred to in the sequel as the compression map, and denoted as \( c \).

A trdag is said to be a tdag, or fully compressed, iff for any two different nodes \( u, u' \) on \( t \), the two sub-dags \( t|_u \) and \( t|_{u'} \) have non-isomorphic tree-equivalents; otherwise, the trdag is said to be partially compressed when it is not a tree. For example, the tree to the left of Figure 1 is the tree-equivalent of the partially compressed trdag to the right, and also to the fully compressed tdag to the middle.

We define now the notion of a Tree/Dag automaton, first over a ranked alphabet \( \Sigma \), to facilitate understanding. The definition is then easily extended to the unranked case.
Definition 2 A Tree/Dag automaton (TDA, for short) over a ranked alphabet \( \Sigma \) is a tuple \((\Sigma, Q, F, \Delta)\), where \( Q \) is a finite non-empty set of states, \( F \subseteq Q \) is the set of final (or accepting) states, and \( \Delta \) is a set of transition rules of the form: \( f(q_1, \ldots, q_k) \rightarrow q \), where \( f \in \Sigma \) is of rank \( k \), and \( q_1, \ldots, q_k, q \in Q \).

It will be convenient to write the transition rules of a TDA in a different (but equivalent) form: a transition of the form \( f(q_1, \ldots, q_k) \rightarrow q \) is also written as \((f, q_1, \ldots, q_k) \rightarrow q\), where \( q_1, \ldots, q_k \) is seen as a word in \( Q^* \), of length \( = \text{rank}(f) \) in the ranked case. The notion of a TDA is then extended easily to the unranked case. The notion of a TDA is then extended easily to the unranked case, i.e., where the signature symbols naming the nodes are not assumed to be of fixed rank: it suffices to define the transitions to be of the form \((f, \omega) \rightarrow q\), where \( \omega \in Q^* \); we may assume wlog that \( \omega \) is a \( \star \)-regular expression on \( Q \) not involving \( \star \), by replacing a rule \((f, \omega + \omega') \rightarrow q\), by the two rules \((f, \omega) \rightarrow q, (f, \omega') \rightarrow q\).

A TDA is said to be bottom-up deterministic if and only if whenever there are two transition rules of the form \((f, \omega) \rightarrow q, (f, \omega') \rightarrow q'\), with \( q \neq q' \), we have necessarily \( \omega \cap \omega' = \emptyset \); otherwise it is said to be non-deterministic. We also agree to denote the transitions of the form \((f, \emptyset) \rightarrow q\) simply as \( f \rightarrow q \), and refer to them as initial transitions.

For defining the notion of runs of TDAs on a trdag in a bottom-up style, we need some preliminaries. Let \( A \) be a TDA with state set \( Q \) and transition set \( \Delta \). Suppose \( t \) is a trdag and assume given a map \( M : \text{Edges}(t) \rightarrow Q \). If \( u \) is any node on \( t \) with \( u_1 \ldots u_n \) as the string of all its (not necessarily distinct) children, the string \( M(e(u, 1)) \ldots M(e(u, n)) \), formed of states assigned by \( M \) to the outgoing edges at \( u \), will be denoted as \( M(\text{Out}_u) \). We then define, recursively in a bottom-up style, a binary relation at \( u \) on the states of \( Q \), with respect to (w.r.t. or wrt, for short) the given map \( M \); this relation, denoted as \( \sigma_u^M = \sigma_u \), is defined as follows:

Definition 3 Let \( A, t, M \) be as above, and \( u \) any given node on the trdag \( t \).

- If \( u \) is a leaf with \( \text{name}(u) = a \), then \( q \triangleleft_u q' \) iff whenever \( a \rightarrow q \in \Delta \) we also have \( a \rightarrow q' \in \Delta \);
- otherwise \( q \triangleleft_u q' \) iff:
  (i) \( \text{name}(u), M(\text{Out}_u) \rightarrow q \) is an instance of a transition rule in \( \Delta \); i.e., \( \Delta \) has a rule \( \text{name}(u), \omega \rightarrow q \) such that \( M(\text{Out}_u) \) is in \( \omega \);
  (ii) there exists a map \( \sigma_q : Q \rightarrow Q \), such that:
    - \( \sigma_q(q) = q' \), and the rule \( \text{name}(u), \sigma_q(M(\text{Out}_u)) \rightarrow q' \) is also an instance of a transition rule in \( \Delta \);
    - for any edge \( e : u \rightarrow u' \in \text{Out}_u \), we have: \( M(e) \triangleleft_{u'} \sigma_{q'}(M(e)) \).

Definition 4 Let \( A = (\Sigma, Q, F, \Delta) \) be any given TDA, and \( t \) any given trdag. A run of \( A \) on \( t \) is a pair \((r, M)\), where \( r : \text{Nodes}(t) \rightarrow Q \) and \( M : \text{Edges}(t) \rightarrow Q \) are maps such that the following conditions hold, at any node \( u \) on \( t \):
(1) if \( \text{name}(u) = f \), then the rule \((f, M(\text{Out}_u)) \rightarrow r(u)\) is an instance of a transition rule in \( \Delta \);

(2) there is an incoming edge \( e \in \text{In}_u \) with \( M(e) = r(u) \); and for every \( e' \in \text{In}_u \) such that \( M(e') = q' \neq q = r(u) \), we have \( q \sqsubseteq^M u_q' \)

A run \((r, M)\) is accepting on a trdag \( t \) iff \( r(\varepsilon) \in F \), i.e. \( r \) maps the root-node of \( t \) to an accepting state. A trdag \( t \) is accepted by a TDA iff there is an accepting run on \( t \). The language of a TDA is the set of all trdags that it accepts.

Remark 1. i) Note that if \( t \) is a tree, then \( \text{In}_u \) is singleton at every non-root node \( u \) on \( t \), so a run \((r, M)\) of any TDA on \( t \) can be identified with its first component \( r \); we get then the usual notion of runs of tree automata on trees.

Example 1. Over the unranked signature \( \{a, f, g\} \) consider a TDA \( A \), with the following transitions:

\[
\begin{align*}
\text{a} & \rightarrow p, \quad \text{b} \rightarrow q', \quad \text{b} \rightarrow p, \quad \text{b} \rightarrow q, \\
(a, p) & \rightarrow q, \quad (a, q) \rightarrow p, \quad (a, q') \rightarrow q, \\
(g, qQ^*) & \rightarrow q, \quad (g, pq) \rightarrow p, \\
(f, qpq) & \rightarrow q_{\text{fin}}, \quad (f, pQ^*) \rightarrow q_{\text{fin}},
\end{align*}
\]

with \( Q = \{p, q, q', q_{\text{fin}}\} \), and \( q_{\text{fin}} \) as the unique accepting state. An accepting bottom-up run of \( A \) on a trdag is depicted on the left of Figure 2, and on its right, the “same” run as seen on the tree equivalent of the trdag.

A few comments on the above run may be of help: we start with assigning state \( q \) to the leaf node \( b \), under \( r \); the assignments of state \( q \) under \( M \) to all the incoming edges at this node \( b \) poses no problem; we can then assign state \( p \) to node \( a \), and subsequently also \( p \) to the node \( g \), under \( r \), via the transition rule \((g, pq) \rightarrow p \); we then assign \( p \) under \( M \) to the first incoming edge at \( g \); to assign state \( q \) under \( M \) to the second incoming edge at \( g \), we just need to check that:

- for a map \( \sigma : Q \rightarrow Q \) such that \( \sigma(p) = q, \sigma(q) = p \), the rule \((g, \sigma(pq)) \rightarrow q \) is an instance of a transition rule of the TDA;

- for the outgoing edge \( g \rightarrow a \), labeled with \( p \) by \( M \), we have \( p \sqsubseteq_a q = \sigma(p) \);

- for the outgoing edge \( g \rightarrow b \), labeled with \( q \) by \( M \), we do have \( q \sqsubseteq_b q = \sigma(q) \);

reaching \( q_{\text{fin}} \) at the root-node is trivial via the last transition rule. (Note that
we could have as well assigned \( p \) under \( M \) to the second incoming edge at \( g \), with no conditions to check, then reach \( q_{fin} \).

**Remark 1** (contd.). ii) Unlike the DAs of [5] or [1], the following bottom-up non-deterministic TDA: \( a \to q_1, a \to q_2, f(q_1, q_2) \to q_a \), with \( q_0, q_1, q_a \) as states where \( q_a \) is accepting, has a non-empty language: as a TDA it accepts \( f(a, a) \).

For a deterministic TDA, we have the following result (as expected):

**Proposition 1** Let \( A \) be a bottom-up deterministic TDA, and \( t \) any given trdag; then there is at most one run of \( A \) on \( t \).

**Proof.** Let \( Q \) be the set of states of \( A \), and \( M : \text{Edges}(t) \to Q \) any given map assigning states to the edges on \( t \). We shall show by induction that the hypothesis of determinism on \( A \) implies that, at any node \( u \) on \( t \), the binary relation \( \preceq_u^M = \preceq_u \) defined above (Definition 3), w.r.t. the map \( M \), is the identity relation on the set \( Q \). The proposition will then follow from conditions (1) and (2) on runs, cf. Definition 4; we will get, in particular, that for every incoming edge \( e \) at \( u \), \( M(e) \) must be the same as \( r(u) \); so the run can be identified with its first component \( r \) (as on a tree).

The induction will be on a non-negative integer \( d_u \), that we define at any node \( u \) on \( t \)– and refer to as its height on \( t \)– as the maximal number of arcs on \( t \) from \( u \) to the leaf nodes. If \( d_u = 0 \), then \( u \) is a leaf node; that \( \preceq_u \) is the identity relation on \( Q \) in this case is immediate, from the determinism of \( A \), and the definition of \( \preceq_u \). So, assume that \( d_u > 0 \), and let \( v_1 \ldots v_n \) be the string of all the children nodes of \( u \) on \( t \). By the inductive hypothesis, for every \( i, 1 \leq i \leq n \), the relation \( \preceq_i \) is the identity relation on \( Q \); it follows then, from the conditions (i) and (ii) on the relation \( \preceq_u \) (Definition 3), that this latter must also be the identity relation on \( Q \).

We may now formulate the principal result of the first part of this paper:

**Proposition 2** i) A TDA accepts a trdag \( t \) if and only if it accepts the tree equivalent of \( t \).

ii) The emptiness problem for a TDA is decidable in time \( P \) w.r.t. its number of states.

iii) The uniform membership problem for a TDA is decidable in time \( N P \) (resp. time \( P \)) w.r.t. its number of states, and the number of edges (resp. and the number of positions) on the given trdag.

**Proof.** Let \( \hat{t} \) be the tree equivalent of the trdag \( t \), and \( c \) the natural surjective compression map from \( \text{Nodes}(\hat{t}) \) onto \( \text{Nodes}(t) \).

Property i): For proving the ‘only if’ part, one uses the following reasoning, coupled with induction on the height function at the nodes of \( t \) (defined in the proof of the previous proposition): Let \((r, M)\) be an accepting run of the given TDA on the trdag \( t \); consider a node \( s \) on the tree equivalent \( \hat{t} \), of which the node \( u \) on \( t \) is the image under the compression map \( c \); let \( r(u) = q \) under the given run of the TDA on \( t \); then, for every state \( q' \) of the TDA such that \( q \preceq_u^M q' \), one can construct a partial run of the TDA –seen as a usual tree automaton– on the tree \( \hat{t} \), climbing up from a leaf below \( s \) on \( \hat{t} \) to the node \( s \), and assigning the state \( q' \) to this node (for an illustrative example, see the tree to the right of Figure 2).

Proving the ‘if’ part of Property i) is a little more complex. We start with a given accepting run \( \hat{r} \) of the given TDA, as a bottom-up tree automaton running in the usual sense on the tree \( \hat{t} \); from this run \( \hat{r} \), we shall construct a run \((r, M)\) of the TDA on the trdag \( t \), by an inductive, top-down traversal of the tdag \( t \); for this top-down traversal, we will be using an integer valued function defined at
any node \( u \) of \( t \) — and referred to as its depth on \( t \) — as the maximal number of arcs on \( t \) from the root node on \( t \) to the node \( u \). We shall also use the fact that the nodes of \( t \) are in natural bijection with the set \( Pos(t) \) of positions on \( t \). The top-down construction of the run \((r, M)\) is done by the following pseudo-algorithm, where \( d \) stands for the maximal depth on \( t \) at its leaf nodes.

BEGIN
/* define first \( r \) at the root node on \( t \),
and \( M \) on its outgoing edges */
\( r(\epsilon_i) = \hat{\rho}(\epsilon_j) \);  
For every outgoing edge \( e_j \), \( 1 \leq j \leq k \),
at \( \epsilon_i \), set \( M(e_j) = \hat{\rho}(\epsilon_j) \);
\( i = 1; \) /* Now go down */
while (\( i < d \)) do {
For every node \( u \) at depth \( i \) do {
choose \( e \in In_u(t) \), and \( \alpha \in pos_1(e) \)
such that \( M(e) = \hat{\rho}(\alpha) \);
set \( r(u) = M(e) \);
For every \( e_j \in Out_u(t) \), \( 1 \leq j \leq m \),
outgoing from \( u \), set \( M(e_j) = \hat{\rho}(\alpha_j) \);
}
i = i + 1; }
END.

It is not difficult to check then, that by construction, the pair of maps \((r, M)\) gives an accepting run of the TDA on the tdag \( t \). (The reasoning is illustrated below.)

Properties ii) and iii) follow, in the ranked case, from the proof of i) and the results of TATA ([6]), Chapter 1; in the unranked case, one can either employ a reasoning based on reduction to the ranked case as in [10], or appeal directly to the results of [13]. (Note: the number of positions on a tdag is the same as the size of its tree equivalent.)

We illustrate here the reasoning employed in the proof of the ‘if’ part of assertion i) of the above proposition, with the tdag \( t \) of Example 1. We start with the run \( \hat{\rho} \) on its tree-equivalent \( t' \), as depicted to the right of Figure 2. At start, to the root node on \( t \) (at depth 0) is assigned the state \( g_{fin} \), and to its three outgoing edges, are assigned the three states \( p, q, q \) respectively; at \( g \), which is the only node on \( t \) at depth 1, we choose the first incoming edge (of position 1, and labeled with \( p \) by \( M \)), and set \( r(u) = \hat{\rho}(1) = p; \) the two outgoing edges at \( g \) on \( t \) have as positions the sets \{11, 21\}, \{12, 22\} respectively; to these two outgoing edges at \( g \) on \( t \), we assign the states that \( \hat{\rho} \) assigns to the two sons of the node \( g \) at position 1 on \( t \), namely \( p, q \) respectively (this means in essence that we have ‘selected’ the positions 11 and 12 on the two outgoing edges at \( g \) on \( t \)); next, we go to depth 2 on \( t \), where \( a \) is the unique node, to which we then have to assign the state \( \hat{\rho}(11) \) that \( M \) has already assigned to its incoming edge; the rest of the reasoning is obvious, so left out.

**Remark 2.** Let \( t \neq t' \) be two given tdags such that \( Pos(t') = Pos(t) \), and there is a name preserving surjective map \( c' \) from \( Nodes(t') \) onto \( Nodes(t) \). We can then define \( t \) to be a compression, or compressed form, of \( t' \); and refer to \( t' \) as an uncompressed equivalent of \( t \), and to the surjective map \( c' \) on \( Nodes(t') \) as a compression map. It is easily checked that \( t \) and \( t' \) have then the same tree-equivalent; and it follows from Proposition 2 above that any given TDA \( A \) accepts \( t \) if and only if it accepts \( t' \). This means that it is legitimate to define the language of a TDA as the set of all tdags that it accepts (or trees that it accepts), or as the set of all tdags accepted, up to tree-equivalence.
3 Querying Compressed Documents: Preliminaries

Given a trdag $t$, one can naturally construct a regular tree grammar associated with $t$, which is straightline (cf. [4]), in the sense that there are no cycles on the dependency relations between its non-terminals, and each non-terminal produces exactly one sub-trdag of $t$. Such a grammar will be denoted as $L_t$, if it is normalized in the following sense:

(i) for every non-terminal $A_i$ of $L_t$, there is exactly one production of the form $A_i \rightarrow f(A_{j_1}, \ldots, A_{j_k})$, where $i < j_r$ for every $1 \leq r \leq k$; we shall then set $\text{Sons}(A_i) = \{A_{j_1}, \ldots, A_{j_k}\}$, and $\text{symb}_{L_t}(A_i) = f$;

(ii) the number of non-terminals is the number of nodes on $t$.

Such a normalized grammar $L_t$ is uniquely defined up to a renaming of the non-terminals. For instance, for the trdag $t$ to the left of Figure 3 we get the following normalized grammar:

$$A_1 \rightarrow f(A_2, A_3, A_4, A_5, A_2), \quad A_2 \rightarrow c, \quad A_3 \rightarrow a(A_5), \quad A_4 \rightarrow b, \quad A_5 \rightarrow b.$$

Such a grammar is easily constructed from $t$, for instance by using a standard algorithm which computes the ‘depth’ of any node (as the maximal distance from the root), to number the non-terminals so as to satisfy condition (i) above.

![Diagram](image)

**Fig. 3.** trdag $t$, associated rlag $D_t$, and chiblings of $L_t$

The dependency graph of the normalized grammar $L_t$ associated with $t$, and denoted as $D_t$, consists of nodes named with the non-terminals $A_i, 1 \leq i \leq n$, and one single directed arc from any node $A_i$ to a node $A_j$ whenever $A_j$ is a son of $A_i$. The root of $D_t$ is by definition the node named $A_1$. The notion of Sons of the nodes on $D_t$ is derived in the obvious way from that defined above on $L_t$.

Furthermore, to any production $A_i \rightarrow f(A_{j_1}, \ldots, A_{j_k})$ of $L_t$, we associate a rooted linear graph composed of $k$ nodes respectively named $A_{j_1}, \ldots, A_{j_k}$, with root at $A_{j_1}$ and such that for all $l \in \{2, \ldots, k\}$ the node named $A_{j_l}$ is the son of the node named $A_{j_{l-1}}$. This graph will be called the chibling of $L_t$ associated with the (unique) $A_i$-production; it is denoted as $F_i$. We also define a further chibling denoted $F_0$, as the linear graph with a single node named $A_1$, where $A_1$ is the axiom of $L_t$.

In the sequel, we designate by $G$ either $D_t$ or any of the chiblings $F$ of $L_t$. We complete any of these acyclic graphs $G$ into a rooted labeled acyclic graph (rlag, for short), by attaching to each node $u$ on $G$, with name$(u) = A_i$, a label denoted label$(u)$, and defined as label$(u) = (\text{symb}_{L_t}(A_i), -)$; cf. Figure 3.

3.1 Positive Core XPath Queries on trdags

In this paper we restrict our study to positive Core XPath queries on trdags. Recall that Core XPath is the navigational segment of XPath, and is based on the following axes of XPath (cf. [10, 19]): self, child, parent, ancestor, descendant, following-sibling, preceding-sibling. A location expression
is defined as a predicate of the form $\text{axis}::b$, where $\text{axis}$ is one of the above axes, and $b$ is a symbol of $\Sigma$. Given any trdag $t$ over $\Sigma$, a context node $u$ on $t$ and $b \in \Sigma$, the semantics for $\text{axis}$ is defined by evaluating this predicate at $u$. The semantics for the axes $\text{self}$, $\text{child}$, $\text{descendant}$ are easily defined, exactly as on trees (cf. [19]). For defining the semantics of the remaining axes, we first recall that $\text{Parents}(u) = \{v \in \text{Nodes}(t) \mid u \text{ is a child of } v\}$.

**Definition 5** Given a context node $u$ on a trdag $t$, and $b \in \Sigma$:

i) $[\text{parent}::b]$ evaluates to true at $u$, if and only if there exists a $b$-named node in $\text{Parents}(u)$;

ii) $[\text{ancestor}::b]$ evaluates to true at $u$, iff either $[\text{parent}::b]$ evaluates to true at $u$, or there exists a node $v \in \text{Parents}(u)$ such that $[\text{ancestor}::b]$ evaluates to true at $v$;

iii) $[\text{following-sibling}::b]$ evaluates to true at $u$, iff there exists a $b$-named node $u'$, and a node $v$ on $t$ such that $\gamma(v)$ is of the form $..u'..$;

iv) $[\text{preceding-sibling}::b]$ evaluates to true at $u$, iff there exists a $b$-named node $u'$, and a node $v$ on $t$ such that $\gamma(v)$ is of the form $...u'$..

For the ‘composite’ axes $\text{descendant-or-self}$ and $\text{ancestor-or-self}$, the semantics are then deduced in an obvious manner. We shall also need position predicates of the form $[\text{position}() = i]$; their semantics is that the expression $[\text{child}::b \ [\text{position}() = i]]$ evaluates to true at a context node $u$, iff: $[\text{child}::b]$ evaluates to true at $u$, and $u$ is an $i$-th child of some parent.

Positive Core XPath query expressions are usually defined in the literature (cf. e.g., [7]), as those generated by the following grammar:

$$A ::= \text{self} \mid \text{child} \mid \text{descendant} \mid \text{parent} \mid \text{ancestor} \mid \text{preceding-sibling} \mid \text{following-sibling}$$

$$S\text{can} ::= A::\sigma \mid \text{position}() = i \mid S\text{can} \text{ and } S\text{can} \mid S\text{can} \text{ or } S\text{can}$$

$$E\text{can} ::= A::[S\text{can}] \mid E\text{can}[E\text{can}]$$

$$Q\text{can} ::= /S\text{can} \mid /E\text{can} \mid Q\text{can}/Q\text{can}$$

We shall refer to the query expressions generated by this grammar as canonical; they can be shown to be of the type $//C_1/C_2/\ldots/C_n$, where each $C_i$ is of the form $A::\sigma[X\text{can}]$, or of the form $A::\sigma[X\text{can}] \text{ conn } A'::\sigma'[X'\text{can}]$, with $\text{conn} \in \{\text{and}, \text{or}\}$, and $X\text{can}, X'\text{can} \in \{S\text{can}, E\text{can}, \text{true}\}$; we agree here to identify $A::\sigma[\text{true}]$ with $A::\sigma$.

Any such positive Core XPath query expression can be translated into one that is in “standard form”, i.e., where the format of the sub-queries is of the type ‘$\text{axis}::b$’; we formalize this idea now. We shall refer to the axes $\text{self}$, $\text{child}$, $\text{descendant}$, $\text{parent}$, $\text{ancestor}$, $\text{preceding-sibling}$, $\text{following-sibling}$ as basic. A basic Core XPath query is a query of the form $//*[\text{axis}::\sigma]$, where $\text{axis}$ is a basic axis. More generally, the queries we propose to evaluate on trdags are defined formally as the expressions $Q\text{std}$ generated by the following grammar, where $\sigma$ stands for any node name on the documents, or for * (meaning ‘any’):

$$A ::= \text{self} \mid \text{child} \mid \text{descendant} \mid \text{parent} \mid \text{ancestor} \mid \text{preceding-sibling} \mid \text{following-sibling}$$

$$S ::= A::\sigma \mid \text{position}() = i \mid S \text{ and } S \mid S \text{ or } S \mid \text{Root}$$

$$E ::= A::*[S] \mid E[E]$$

$$Q\text{std} ::= ///[^*] \mid ///[^*]S \mid ///[^*]E$$

Core XPath queries $Q\text{std}$ of the format generated by this grammar are said to be in standard form; to be able to handle any positive Core XPath query with such a grammar, we have introduced a special predicate called $\text{Root}$, deemed true only at the root node of the trdag considered.

By the evaluation of a given query expression $Q$ on any trdag $t$, we mean the assignment: $t \mapsto$ the set of all context nodes on $t$ where the expression $Q$ evaluates to true (following the conventions of Definition 2); this latter set is also called the answer for $Q$ on $t$. Two given queries $Q_1, Q_2$ are said to be equivalent
iff, on any trdag $t$, the answer sets for $Q_1$ and $Q_2$ are the same. Any positive Core XPath query $Q_{can}$ can be translated into an equivalent one in standard form; e.g., $/c[following-sibling::g]/d$ is equivalent to $//*[self::d and parent::*:*[following-sibling::g]]$ in standard form.

An inductive procedure performing such a translation in the general case (of linear complexity w.r.t. the number of location steps in $Q_{can}$) is given in Appendix I. The following proposition results from Definition 5.

**Proposition 3** (1) For any set of nodes $X$ on a trdag $t$, and any axis $A$, we have: $A(X) = \bigcup_{x \in X, \alpha \in pos(x)} \{/child::*[position()=i_1]/.../child::*[position()=i_k]/A::*\}$

(2) For any trdag $t$, and any node with name $b$ on $t$, we have:

(i) $//*[preceding::b] = \bigcup_u \{\text{descendant-or-self}(\text{following-sibling}(u) ) \}$

(ii) $//*[following::b] = \bigcup_u \{\text{descendant-or-self}(\text{preceding-sibling}(u) ) \}$

Finally, following [2], for any set $S$ of nodes on $t$, the sets of nodes following($S$) and preceding($S$) can now be defined formally, as follows:

following($S$) = descendant-or-self(following-sibling(ancestor-or-self($S$)));
preceding($S$) = descendant-or-self(preceding-sibling(ancestor-or-self($S$))).

**Note:** Unlike on a tree, the ancestor, descendant, following, self and preceding axes do not partition the set of nodes on a trdag $t$, in general.

4 Automata for the Basic Core XPath Queries

4.1 The Semantics of the Approach

We first consider basic Core XPath queries. Composite or imbricated queries will subsequently be evaluated in a stepwise fashion; see Section 6.

To any basic query $Q = ///*[axis::*]$, we shall associate a word automaton (actually a transducer), referred to as $A_Q$. It will run top-down, on the rlag $D_t$ if $axis$ is non-sibling, and on each of the chiblings $F$ of $L_t$ otherwise. In either case, a run will attach, to any node traversed, a pair of the form $(l, x)$, where the component $l$ of the pair has the intended semantics of selection or not, by $Q$, of the corresponding node on $t$, and the component $x$ will be a 1 or 0, with the intended semantics that $x = 1$ iff the corresponding node on $t$ has a descendant answering $Q$. At the end of the run, label($u$), at any node $u$ of $D_t$, will be replaced by a new label derived from the ll-pairs attached to $u$ by the run.

To formalize these ideas, we introduce a set of new symbols $L = \{s, \eta, T, \top\}$ referred to as labels (the term ‘label’ is used so as to avoid confusion with the term label). We define ll-pairs as elements of the set $L \times \{0, 1\}$, and the states of $A_Q$ as elements of the set $\{\text{init}\} \cup (L \times \{0, 1\})$. For any $Q$, the automaton $A_Q$ is over the alphabet $\Sigma \cup \{s, \eta\}$, has $\text{init}$ as its initial state, and has no final state. The set $\Delta_Q$ of transitions of $A_Q$ will consist of rules of the form $(q, \tau) \rightarrow q'$ where $q \in \{\text{init}\} \cup (L \times \{0, 1\})$, $q' \in (L \times \{0, 1\})$, and $\tau \in \Sigma \cup \{s, \eta\}$.

For any rlag $G$, we define a function lab: Nodes($G$) $\rightarrow \Sigma \cup \{s, \eta\}$, by setting lab($u$) = $\pi_1$(label($u$)), the first component of label($u$). The automaton $A_Q$ associated to a basic query $Q = ///*[axis::*]$ will run top-down on the rlag $G$.
where $G$ is $D_t$ if \textbf{axis} is a basic non-sibling axis, and $G$ is any chibling $F$ of $L$; if \textbf{axis} is a basic sibling axis. A run of $A_Q$ on $G$ is a map $r : \text{Nodes}(G) \to L \times \{0, 1\}$, such that, for every $u \in \text{Nodes}(G)$, the following holds:

- if $u$ is root$_G$, then the rule $(\text{init}, llab(u)) \to r(u)$ is in $\Delta_Q$;
- otherwise, for every $v \in \gamma(u)$ the rules $(r(u), llab(v)) \to r(v)$ are all in $\Delta_Q$.

(Note: when \textbf{axis} is non-sibling, this amounts to requiring that, for any node $v$, the state $r(v)$ must be in conformity with the states $r(u)$ for every parent node $u$ of $v$, with respect to the rules in $\Delta_Q$.)

From the run of the automaton $A_Q$ and from the states it attaches to the nodes of $D_t$, we will deduce, at every node $u$ of $t$, a well-determined ll-pair as (a new) label at $u$, via the natural bijection between $\text{Nodes}(t)$ and $\text{Nodes}(D_t)$.

The ll-pairs thus attached to the nodes of $t$ will have the following semantics (where $x$ stands for the name of the node $u$ on $t$, corresponding to the ‘current’ node on $D_t$):

- $(\top', 1) : x = \sigma$, current node on $t$ is selected by (i.e., is an answer for) $Q$;
- $(\top, 1) : x = \sigma$, current node is not selected, but has a selected descendant;
- $(\top, 0) : x = \sigma$, current node is not selected, and has no selected descendant;
- $(s, 1) : x \neq \sigma$, current node is selected;
- $(\eta, 1) : x \neq \sigma$, current node is not selected, but has a selected descendant;
- $(\eta, 0) : x \neq \sigma$, current node is not selected, and has no selected descendant.

Only the nodes on $D_t$, to which the run of $A_Q$ associates the labels $(s, 1)$ or $(\top', 1)$, correspond to the nodes of $t$ that will get selected by the query $Q$.

The ll-pairs with boolean component 1 will label the nodes of $D_t$ corresponding to the nodes of $t$ which are on a path to an answer for the query $Q$; thus the automata $A_Q$ will have no transitions from any state with boolean component 0 to a state with boolean component 1. Moreover, with a view to define runs of such automata which are unique (or unambiguous in a sense that will be presently made clear), we define the following priority relations between the ll-pairs:

\[(\eta, 0) > (\eta, 1) > (s, 1), \quad \text{and} \quad (\top, 0) > (\top, 1) > (\top', 1).\]

A run of the automaton $A_Q$ will label any node $u$ on $G$ with an ll-pair either from the group \{$(\top, 0), (\top, 1), (\top', 1)$\} or from the group \{(s, 1), (s, 1)\}; and this group is determined by $llab(u)$.

For ease of presentation, we agree to set $\eta' := s$, and often denote either of the above two groups of ll-pairs under the uniform notation \{(l, 0), (l, 1), (l', 1)\}, where $l \in \{\eta, \top\}$, with the ordering $(l, 0) > (l, 1) > (l', 1)$.

We shall construct a run $r$ of $A_Q$ on $G$ that will be uniquely determined by the following maximal priority condition:

\[(\text{MP}) : \text{at any node } v \text{ on } G, r(v) \text{ is the maximal ll-pair } (l, x) \text{ for the ordering } > \text{ in the group } \{(l, 0), (l, 1), (l', 1)\}, \]
\[
\text{determined by } llab(v), \text{ such that } A_Q \text{ contains a transition rule of the form } (r(u), llab(v)) \to (l, x), \text{ for every parent } u \text{ of } v.
\]

Such a run will assign a label with boolean component 1 only to the nodes corresponding to those of the minimal sub-trdag $t$ containing the root of $t$ and all the answers to $Q$ on $t$.

### 4.2 Re-labeling of $D_t$ by the Runs of $A_Q$

We first consider a non-sibling basic query $Q$ on a given document $t$, and a given run $r$ of the automaton $A_Q$ on $D_t$; at the end of the run, the nodes on $D_t$ will get re-labeled with new ll-pairs, computed as below for every $u \in \text{Nodes}(D_t)$:

\[
\text{lab}_r(u) = (s, 1) \text{ iff } r(u) \in \{(s, 1), (\top', 1)\},
\]
\[
\text{lab}_r(u) = (\eta, 1) \text{ iff } r(u) \in \{(\eta, 1), (\top, 1)\},
\]
\[
\text{lab}_r(u) = (\eta, 0) \text{ iff } r(u) \in \{(\eta, 0), (\top, 0)\}.
\]
We first present the automata for the basic queries.

### 4.3 The Automata

We then derive, at each node of $D_t$ a unique ll-pair in conformity with the semantics of our approach, by using the following function:

\[
\lambda_t(u) = s \iff \ll_t(u) \cap \{(s,1), (\top',1)\} \neq \emptyset,
\]

\[
\lambda_t(u) = \eta \iff \ll_t(u) \cap \{(s,1), (\top',1)\} = \emptyset.
\]

From $D_t$ and this function $\lambda_t$, we next derive an rlag $\hat{r}_t(D_t)$ by re-labeling each node $u$ on $D_t$ with the pair $(\lambda_t(u), \quad)$. And finally we define $\hat{r}(D_t)$ as the rlag obtained from $\hat{r}_t(D_t)$, by running on it the automaton for the basic non-sibling query //*[self::s,], as indicated at the beginning of this subsection. In practical terms, such a run amounts in essence to setting, as the second component of $label(u)$ at any node $u$, the boolean 1 iff $u$ is on a path to some node with lab $s$, and 0 otherwise. All these details are illustrated with an example in the following subsection.

### The Automata

We first present the automata for the basic queries //*[self::s] and for //*[following-sibling::s], and give an illustrative example using the former for $s = s$, and the latter for $s = b$. The automata for the other basic queries are given after the example.

#### Automata: for //*[self::s] and for //*[following-sibling::s]

![Diagram](image)

Figure 4 below illustrates the evaluation of $Q = //*[following-sibling::b]$, on the trdag $t$ of Figure 3. We first use the automaton for the basic query //*[following-sibling::s] with $s = b$, and then the automaton for //*[self::s] with $s = s$. The sub-trdag of $t$, formed of nodes corresponding to those of $\hat{r}(D_t)$ with labels having boolean component 1, contains all the answers to $Q$ on $t$.
\( I_1 \) on \( F'_1 \):

\[
\begin{align*}
A_2(c,-) & \quad (s,1) \\
\downarrow & \\
A_3(a,-) & \quad (s,1) \\
\downarrow & \\
A_4(b,-) & \quad (\top',1) \\
\downarrow & \\
A_5(b,-) & \quad (\top,0) \\
\downarrow & \\
A_2(c,-) & \quad (\eta,0)
\end{align*}
\]

\( I_0 \) on \( F_0 \):

\[
\begin{align*}
A_1(f,-) & \quad (\eta,0) \\
\end{align*}
\]

\( I_3 \) on \( F_3 \):

\[
\begin{align*}
A_5(b,-) & \quad (\top,0) \\
\end{align*}
\]

\( I_2, I_2, I_3 \) on \( D_{\lambda} \):

\[
\begin{align*}
A_0(c,-) & \quad (s,1) \\
A_3(a,-) & \quad (s,1) \\
A_4(b,-) & \quad (\top',1) \\
\end{align*}
\]

\[
\begin{align*}
(\top,0) & \quad A_5(b,-)
\end{align*}
\]

**Run of the automaton**

for //*[self::s] on \( \lambda_{D_{\lambda}}(D) \):

\[
\begin{align*}
A_1(\eta,-) & \quad (\eta,1) \\
A_2(s,-) & \quad A_3(s,-) \\
A_4(s,-) & \quad (\top',1) \\
A_5(\eta,-) & \quad (\eta,0) \\
\end{align*}
\]

**Final re-labeled tag**

\( \preceq \top \) on \( D_{\lambda} \):

\[
\begin{align*}
A_1(\eta,1) & \\
A_2(s,1) & \quad A_3(s,1) \\
A_4(s,1) & \quad A_5(\eta,0)
\end{align*}
\]

**Fig. 4.**

- **Automaton for the query //*[parent::\sigma]**

- **Automaton for the query //*[ancestor::\sigma]**
A few words on some of the automata by way of explanation. First, the reason why the automaton for self does not have the states \((\top, 0), (\top, 1), (s, 1)\): for \((\top, 0), (\top, 1)\), by the semantics of subsection 4.1 we must have \(x = \sigma\), where \(x\) is the name of the current node on \(t\), but then the query \(//*[self::\sigma]\) should select the current node, so one cannot be at such a state; as for \((s, 1)\), the reasoning is just the opposite. Next, the reason why the automaton for descendant does not have the states \((\eta, 1), (\top, 1)\): if the semantics attribute one of these pairs to any node \(u\), that would mean the node \(u\) has a selected descendant \(u'\); which means that \(u'\) has some \(\sigma\)-descendant node, which would then be a \(\sigma\)-descendant for \(u\) too, so \(Q\) should select \(u\).
5 Maximal Priority Runs of Basic Query Automata

Note that the following properties, required by our semantics of subsection 4.1, hold on the automata $A_Q$ constructed above, for any basic Core XPath query $Q = \text{//*[axis::\sigma]}$:

i) There are no transitions from any state with boolean component 0 to a state with boolean component 1;

ii) The $\sigma$-transitions have all their target states in $\{(\top, 0), (\top, 1), (\top', 1)\}$; and for any $\gamma \neq \sigma$, the target states of $\gamma$-transitions are all in $\{(\eta, 0), (\eta, 1), (s, 1)\}$.

Theorem 1 Let $Q$ be any basic Core XPath query, $t$ any given trdag, and let $G$ denote either the rtag $D_t$, or any given chibling $F$ of $L_t$. Assume given a labeling function $L$ from $\text{Nodes}(G)$ into the set of ll-pairs, which is correct with respect to $Q$, i.e., in conformity with the semantics of subsection 4.1. Then there is a run $r$ of the automaton $A_Q$ on $G$, such that:

i) $r$ is compatible with $L$; i.e., $r(u) = L(u)$ for every node $u$ on $G$;

ii) $r$ satisfies the maximal priority condition (MP) of subsection 4.1.

Proof. We first construct, by induction, a ‘complete’ run (i.e., defined at all the nodes of $G$) satisfying property i). For that, we shall employ reasonings that will be specific to the axis of the basic query $Q$. We give here the details only for the axis parent; they are similar for the other axes.

$Q = \text{//*[parent::\sigma]}$: (The axis considered is non-sibling so $G = D_t$ here.) At the root $u$ node of $D_t$, we set $r(u) = L(u)$; we have to show that there is a transition rule in $A_Q$ of the form $\text{init}, \text{llab}(u)) \rightarrow L(u)$. Obviously, for the axis parent, the root node $u$ cannot correspond to a node on $t$ selected by $Q$, so the only ll-pairs possible for $L(u)$ are $(l, 0), (l, 1)$, with $l \in \{\eta, \top\}$; for each of these choices, we do have a transition rule of the needed form, on $A_Q$.

Consider then a node $v$ on $D_t$ such that, at each of its ancestor nodes on $D_t$, the part of the run $r$ of $A_Q$ has been constructed such that $r(u) = L(u)$; assume that the run cannot be extended at the node by setting $r(v) = L(v)$. This means that there exists a parent node $w$ of $v$, such that $(L(w), llab(v)) \rightarrow L(v)$ is not a transition rule of $A_Q$; we shall then derive a contradiction. We only have to consider the cases where the boolean component of $L(w)$ is greater than or equal to that of $L(v)$. The possible couples $L(w), L(v)$ are then respectively:

$L(u) : (\top, 0) \mid (\top, 1) \mid (\top', 1) \mid (\top', 1)$

$L(v) : (\eta, 0) \mid (\eta, 1) \mid (\eta', 1) \mid (\eta, 1)$

In all cases, we have $llab(w) = \sigma$ because of the semantics, so the node (on $t$ corresponding to the node) $v$ has a $\sigma$-parent, so must be selected; thus the above choices for $L(v)$ are not in conformity with the semantics; contradiction.

We now prove that the complete run $r$ thus constructed, satisfies property ii). For this part of the proof, the reasoning does not need to be specific for each $Q$; so, write $Q$ more generally, as $\text{//*[axis::\sigma]}$ for some given $\sigma$. Suppose the run $r$ does not satisfy the maximal priority condition at some node $v$ on $G$; assume, for instance, that the run $r$ made the choice, say of the ll-pair $(l, 1)$, although the maximal labeling of the node $v$, in a manner compatible with the ll-pairs of all its parents, was the ll-pair $(l, 0)$. Since $L$ is assumed correct, and $r$ is compatible with $L$, the maximal possible labeling $(l, 0)$ would mean that the node (on $t$ corresponding to the node) $v$ has no descendant selected by $Q$; whereas, the choice that $r$ is assumed to have made at $v$, namely the ll-pair $(l, 1)$, has the opposite semantics whether or not $llab(v) = \sigma$; in other words, the labeling $L$ would not be correct with respect to $Q$: contradiction. The other possibilities for the ‘bad’ labelings under $r$ also get eliminated in a similar manner. □

Theorem 2 Let $Q, t, D_t, F, G$ be as above. Let $r$ be a (complete) run of the automaton $A_Q$ on $G$, which satisfies the maximal priority condition (MP) of
subsection 4.1. Then the labeling function \( L \) on \( \text{Nodes}(G) \), defined as \( L(u) = r(u) \) for any node \( u \), is correct with respect to the semantics of subsection 4.1.

Proof: Let us suppose that the labeling \( L \) deduced from \( r \) is not correct with respect to \( Q \); we shall then derive a contradiction. The reasoning will be by case analysis, which will be specific to the axis of the basic query \( Q \) considered. We give the details here for \( Q = //*[\text{descendant}::s] \). The axis is non-sibling, so we have \( G = D_t \) here. The sets \( \text{Nodes}(t) \), \( \text{Nodes}(D_t) \) are in a natural bijection, so for any node \( u \) on \( D_t \) we shall also denote by \( u \) the corresponding node on \( t \), in our reasonings below.

We saw that the automaton \( A_Q \) for the descendant axis does not have the states \((\eta, 1), (\top, 1)\). Consider then a node \( u \) on \( D_t \) such that: for all ancestor nodes \( w \) of \( u \), the label \( r(w) \) is in conformity with the semantics, but the ll-pair \( r(u) \) is not in conformity. Now, \( A_Q \) has only 5 states: \((\text{init}), (\top', 1), (s, 1), (\top, 0), (\eta, 0)\), of which only the last four can ll-label the nodes. So the possible ‘bad’ choices that \( r \) is assumed to have made at our node \( u \), are as follows:

- \((a)\) \( r(u) = (\top', 1) \), but the node \( u \) is not an answer to the query \( Q \). Here \( \text{name} \) \( u \) must be \( \sigma \), so the choice of \( r \) ought to have been \( (\top, 0) \);
- \((b)\) \( r(u) = (s, 1) \), but the node \( u \) is not an answer to the query \( Q \). Here \( \text{name} \) \( u \) \neq \sigma, so the choice of \( r \) ought to have been \( (\eta, 0) \);
- \((c)\) \( r(u) = (\eta, 0) \), but the node \( u \) is an answer to the query \( Q \). Here \( \text{name} \) \( u \) \neq \sigma, so the choice of \( r \) ought to have been \( (s, 1) \);
- \((d)\) \( r(u) = (\top, 0) \), but the node \( u \) is an answer to the query \( Q \). Here \( \text{name} \) \( u \) must be \( \sigma \), so the choice of \( r \) ought to have been \( (\top', 1) \).

In all the four cases, we have to show:

- \((i)\) that the “ought-to-have-been” choice ll-pair is reachable from all the parent nodes of \( u \);
- \((ii)\) and that, with such a new and ‘correct’ choice made at \( u \), \( r \) can be completed from \( u \), into a run on the entire \( D_t \).

The reasoning will be similar for cases \((a)\), \((b)\), and for the cases \((c)\), \((d)\). Here are the details for case \((a)\): That \( u \) is not an answer to \( Q \) means that \( u \) has no \( \sigma \)-descendant node, so for all nodes \( v \) below \( u \) on \( D_t \), we have \( \text{llab}(v) \neq \sigma \). Therefore, assertions \((i)\) and \((ii)\) above follow from the following observations on the automaton for \( Q = //*[\text{descendant}::s] \):

- \((i)\) if \( r \) could reach the state \((\top', 1)\) at node \( u \) (via a \( \sigma \)-transition) from any parent node of \( u \), then \( (\top, 0) \) is also reachable thus at \( u \), from any of them;
- \((ii)\) if, from the state \((\top', 1)\), \( r \) could reach all the nodes on \( D_t \) below \( u \) (with state \((\eta, 0)\)), via transitions over \( \gamma \neq \sigma \), then it can do exactly the same now, with the ‘correct’ choice ll-pair \((\top, 0)\) at \( u \).

As for case \((c)\): Node \( u \) is an answer to \( Q \) here, so \( u \) has a \( \sigma \)-descendant; let \( v \) be a \( \sigma \)-node below \( u \) on \( D_t \); the ll-pair \( r(v) \) that \( r \) assigns to \( v \) must then be either \((\top', 1)\) or \((\top, 0)\); this implies that \( r \) passed from the state \((\eta, 0)\) – supposedly assigned by \( r \) to \( u \) – to \((\top', 1)\) or \((\top, 0)\) somewhere between \( u \) and \( v \); which is impossible, as is easily seen on the automaton \( A_Q \) for the axis descendant considered. The reasoning for case \((d)\) is even easier: from state \((\top, 0)\), no state with an outgoing \( \sigma \)-transition is reachable. \( \square \)

6 Evaluating Composite Queries

A composite query is a query in standard form, but is not basic. We propose to evaluate such a query incrementally. For this, it suffices to consider queries that are of the form //*[\text{A}::x \ \text{conn} \ \text{A}':::x']], where \text{conn} \in \{\text{and, or}\}, or of the form //*[\text{A}1::*\text{[A}2:::\text{s}]}. For those of the former type, we observe first that the components in a disjunction (resp. conjunction) under a ‘*’ can be evaluated separately. Indeed, the answer for \( Q = //*[\text{A}::x \ \text{conn} \ \text{A}':::x'] \) can
be obtained as union (resp. intersection) of the answers for the two “component”
queries ///*[A::x], and ///*[A’::x’], when conn is an or (resp. an and). We
apply the method described earlier, separately for \(Q_1 = ///*[A::x]\) and for \(Q_2 =
///*[A’::x’]\), thus getting two respective evaluating runs \(r_1, r_2\). Any node \(u\)
of the dag \(D_t\) will then be re-labeled, by the composite query \(Q\), with ll-pairs
computed by a function AND when conn = and (resp. OR when conn = or), in
conformity with the semantics presented in the Section 4.1:

\[
\begin{align*}
\text{AND}(u) & = (s, 1) \text{ iff } r_1(u) = (l’, 1) = r_2(u); \\
\text{AND}(u) & = (\eta, 0) \text{ iff } r_1(u) = (l, 0) \text{ or } r_2(u) = (l, 0); \\
\text{AND}(u) & = (\eta, 1) \text{ otherwise. } \\
\text{OR}(u) & = (s, 1) \text{ iff } r_1(u) = (l’, 1) \text{ or } r_2(u) = (l’, 1); \\
\text{OR}(u) & = (\eta, 0) \text{ iff } r_1(u) = (l, 0) = r_2(u); \\
\text{OR}(u) & = (\eta, 1) \text{ otherwise. }
\end{align*}
\]

Figure 5 below illustrates the above reasoning, for the evaluation of the com-
posite query \(Q = ///*[\text{self}::b \text{ and parent}::a]\), on the trdag \(t\) of Figure 3:

---

7 Deriving the Answer on the Tree-equivalent

Given a Core XPath query \(Q\) and its answer set on a trdag \(t\), we show here how
to derive the answer for the same query \(Q\) on the tree-equivalent \(\hat{t}\) of \(t\); this is
of importance, since the “standard model” for an XML document (even when
given in a compressed form) is generally considered as the tree representation
of the document.

We observe, to start with, that the answer set for \(Q\) on \(t\) is in general a
superset of the answer set for \(Q\) on the tree-equivalent \(\hat{t}\). This can be so for the
following two reasons:

(i) If a certain node \(u\) on \(t\) is selected by \(Q\), not all of the nodes \(u’\) on \(\hat{t}\),
that are ‘lifts’ of \(u\) under the compression map \(c\) on \(\text{Nodes}(t)\), may answer the

---

\[
\begin{array}{c}
///*[self::b] \\
A_1 (l,-) \\
A_2 (c,-) \\
A_3 (a,-) \\
A_4 (b,-) \\
(s,1) \\
A_5 (b,-) \\
A_6 (s,1)
\end{array}
\]

---

\[
\begin{array}{c}
///*[parent::a] \\
A_1 (l,-) \\
A_2 (c,-) \\
A_3 (a,-) \\
A_4 (b,-) \\
(s,1) \\
A_5 (b,-) \\
A_6 (s,1)
\end{array}
\]

---

\[
\begin{array}{c}
\text{and}(D_t) \\
A_1 (0,1) \\
A_2 (0,0) \\
A_3 (0,1) \\
A_4 (0,0)
\end{array}
\]

---

\[
\begin{array}{c}
/(*[self::b \text{ and parent}::a]) \\
A_1 (l,-) \\
A_2 (c,-) \\
A_3 (a,-) \\
A_4 (b,-) \\
(s,1) \\
A_5 (b,-) \\
A_6 (s,1)
\end{array}
\]
query $Q$ on the tree $\hat{t}$, even when $Q$ is a basic query. For instance, consider the basic query $\texttt{//*[@parent::a]}$ on the fully compressed tdag $f(a(c), b(c))$, the (unique) node named $c$ is an answer; it has two $c$-named nodes as lifts on the tree-equivalent $\hat{t}$, of which only one is an answer for the query.

(ii) A node $u$ on a trdag $t$ may answer a composite query $Q$, but none among the lifts of $u$ on $\hat{t}$ may answer the same query $Q$ on the tree $\hat{t}$. For instance, the unique $c$-named node on the compressed tdag $f(a(c), b(c))$ answers the query $\texttt{//*[@parent::a and parent::b]}$, but there is no node on the tree-equivalent answering this query.

Actually, such situations arise only for queries involving the upward axes parent, ancestor, which define relations that are less trivial on trdags than on trees. We can formulate this observation more precisely, as follows:

**Lemma 1.** Let $k$ be one of the axes self, child, descendant, $Q$ the basic query $\texttt{//*[k::x]}$, $t$ any given trdag, $\hat{t}$ its tree-equivalent, $u$ any given node on $t$, and $u' \in c^{-1}(u)$ any node lift of $u$ on $\hat{t}$. Then:

* wrt the maximal priority runs of the automaton for the axis $k$, respectively on $D_t$ and $D_{\hat{t}}$, the nodes $u$ on $t$, and $u'$ on $\hat{t}$, get labeled by the same ll-pair;
* in particular, the node $u$ answers $Q$ on $t$ if and only if the node $u'$ answers the same query $Q$ on the tree $\hat{t}$.

**Proof.** Follows by observing that the semantics of Section 4.1 have been defined in a manner which is top-down, and that the compression map $c : \text{Nodes}(\hat{t}) \to \text{Nodes}(t)$ maps the set $\text{Nodes}(\hat{t}|_{u'})$, of nodes below $u'$ on $\hat{t}$, onto the set of nodes of the sub-trdag $t|_{u'}$. $\blacksquare$

The above lemma is a first step towards the objective of this section. As a second step, we propose to distinguish, on the automata constructed above for the two queries $\texttt{//*[parent::σ]}$, $\texttt{//*[ancestor::σ]}$, the transitions that will never be fired on a tree; such as, e.g., the one from state $(\eta, 1)$ to state $(\top', 1)$. (Note: for this transition to be fireable, we have to reach a node corresponding to a $σ$-named node on the trdag, which must then also have as (unique) parent on the tree –i.e., the node from which the transition is to be fired– a $σ$-named node; this parent node cannot correspond then to a node labeled with $(\eta, 1)$.) Such transitions that are not fireable on a tree will be depicted with dotted arrows on the automaton; the transitions with full arrows are then the ones that are fireable both on trdags and on trees. The two automata thus revised are as follows:

- **Automaton for the query $\texttt{//*[parent::σ]}$ -revised**

![Diagram](image-url)
The next step towards our objective of this section consists in completing a maximal priority run $r$ of the automaton for any given basic query, by associating to a current node $u$ on $D_t$, a subset of $\text{Pos}_t(u)$ (remember: $\text{Nodes}(t)$ and $\text{Nodes}(D_t)$ are in natural bijection), denoted as $P_r(u)$, and defined as follows:

- **case $u$ selected** (i.e. the label of $u$ under $r$ is $(s, 1)$ or $(\top, 1)$; we set $P_r(u) = \bigcup_{\alpha \in J} \{ \alpha.i \mid \alpha.i \in \text{Pos}_t(u) \}$, the union being taken over the positions $\alpha$ of parent nodes $v$ on $t$ such that the transition from $v$ to $u$ is dotted;

- **case $u$ not selected** (i.e. the label of $u$ under $r$ is neither $(s, 1)$ nor $(\top, 1)$: we set here $P_r(u) = \text{Pos}_t(u)$.

A run $r$ completed in this manner will be denoted in boldface type as $r$, giving thus a map $r : \text{Nodes}(t) \rightarrow L \times \{0, 1\} \times \text{Pos}(t)$, defined by $u \mapsto (r(u), P_r(u))$.

In order to derive the answer to a composite query $Q$ on the tree-equivalent of $t$, from the answer for $Q$ on $t$, we need naturally to complete the functions $\text{AND}$ and $\text{OR}$ of Section 6, by adding component giving the selected positions for a query $Q$ which is a conjunction or disjunction of two sub-queries $Q_1, Q_2$. These completed functions, again denoted boldface as $\text{AND}$ and $\text{OR}$, or defined below in a rather obvious manner (the indices 1, 2 correspond to the runs wrt the two queries, and $l_1, l_2$ stand for $s$ or $\eta$; recall that $\eta'$ stands for the label $s$):

\[
\text{AND}(u) = ((s, 1), P_{r_1}(u) \cap P_{r_2}(u)) \text{ if } r_1(u) = ((l'_1, 1), P_{r_1}(u)) \text{ and } r_2(u) = ((l'_2, 1), P_{r_2}(u));
\]

\[
\text{AND}(u) = ((\eta, 0), P_{\text{Pos}}(u)) \text{ if } r_1(u) = (l_1, 0) \text{ and } r_2(u) = (l_2, 0);
\]

\[
\text{AND}(u) = ((\eta, 1), P_{\text{Pos}}(u)), \text{ otherwise.}
\]

\[
\text{OR}(u) = ((s, 1), P_{r_1}(u) \cup P_{r_2}(u)) \text{ if } r_1(u) = ((l'_1, 1), P_{r_1}(u)) \text{ and } r_2(u) = ((l'_2, 1), P_{r_2}(u));
\]

\[
= ((s, 1), P_{r_2}(u)) \text{ if } r_1(u) = (l'_1, 1) \text{ and } r_2(u) \neq (l'_2, 1);
\]

\[
= ((s, 1), P_{r_2}(u)) \text{ if } r_2(u) = (l'_2, 1) \text{ and } r_1(u) \neq (l'_1, 1);
\]

\[
\text{OR}(u) = ((\eta, 0), P_{\text{Pos}}(u)) \text{ if } r_1(u) = (l_1, 0) \text{ and } r_2(u) = (l_2, 0);
\]

\[
\text{OR}(u) = ((\eta, 1), P_{\text{Pos}}(u)), \text{ otherwise.}
\]

**Example 2.** We evaluate the query $Q = //*[\text{ancestor}:b \ [\text{parent}:c]]$ on the trdag $t$ presented to the left of Figure 6; the standard form of this query is $Q = //*[\text{ancestor}::*[\text{self}:b \ and \ \text{parent}:c]]$; and its answer consists of all the nodes having an ancestor $b$ with parent $c$. But we want here to obtain the ‘same’ answer for $Q$ on $t$ and on its tree-equivalent $\hat{t}$ presented to the right of Figure 6. To find such an answer, it is necessary to use the revised automata for the $\text{parent}$ and $\text{ancestor}$ axes. For ease of comprehension, we illustrate the evaluation of $Q$ directly on the trdag $t$ (and not on the rlag $D_t$) - it is possible because, for this document the trdag $t$ and its rlag $D_t$ are isomorphic. Note that each node $u$ of $t$ is represented by its name and the set of positions of the nodes on $t$ that are lifts of $u$.

First, look at Figure 7 where we have presented the evaluation of $Q$ using the non-revised automata. We obtain then an answer on $t$ selecting nodes $a$ and $g$ (which are the nodes of $t$ having an ancestor $b$ with parent $c$); but, if we unfold this answer, we obtain the tree with as selected nodes: $a$ at positions 11, 211, and
Fig. 6. tdag t and its tree-equivalent ē

g at positions 12, 212, 22; obviously this set of nodes is not the answer for Q on ē: indeed, only the nodes a at position 211 and g at position 212 have ancestor b (position 21) with parent c (position 2) on the tree ē.

Fig. 7. Evaluation of Q on t using the non-revised automata

The Figure 8 presents the evaluation of Q on t using the revised automata. As before, first we evaluate in parallel the inner queries */[self::b] and */[parent::c], but this time, to any node on t, the runs of the revised automata associate not only a state, but also a set of (edge-)positions, comprising exactly those corresponding to the productions of the revised automata that are represented by full edges. We find next the answer for conjunction using the revised function AND. This way we have obtained a relabeled rlag with the names of nodes in \{((η, −), (s, −))\}, and on this rlag we evaluate the outer query */[ancestor::s]. Using the revised automata, we thus obtain the answer for Q on t restricted to a set of positions; by unfolding this answer, we get exactly the answer for Q on the tree-equivalent ē of t.

Remark 3. The algorithm given in [7], for the evaluation of (Core) XPath queries on a compressed dag t, actually takes a given position on t as a parameter; this explains that their method works indifferently on the unfolded tree ē, or on
8 Conclusion

Our concern in this paper has been two-fold. The first part addressed the problem of running any bottom-up (unranked) tree automaton indifferently on a tree or on any of the dags obtained from the tree by full or partial compression; this gave rise to the notions of Tree/Dags (trdags) and of Tree/Dag automata. The second part of the paper addressed the issue of retrieving information from a trdag representing an XML document possibly given in a compressed form. (Note: Information retrieval from compressed structures, without having to uncompress them, is a field of active research; cf. e.g., [17, 11].) Limiting our concern here to the evaluation of queries formulated in terms of XPath axes, and more precisely to positive Core XPath queries, we have presented a method for evaluating them on any trdag $t$, without having to uncompress $t$, by breaking up the given query into sub-queries of a basic type; with each basic query, an automaton is associated such that an unambiguous maximal priority run of this automaton can evaluate the query. An algorithm constructing the maximal priority runs is given in Appendix II; it has just been implemented. It is of complexity $O(n^3)$ on any trdag $t$, where $n$ is the number of nodes of $t$; it reduces to $O(n^2)$ if $t$ is a tree, the relation Parents becoming trivial. (Note: our method of evaluation a priori gives the answers for the given query $Q$ on the given trdag $t$, but we have shown how one can derive the answer set for $Q$ on the tree-equivalent of $t$.)

An advantage of the approach presented in this paper seems to be that the basic sub-queries “composing” a given query can be evaluated in parallel, in several cases; a detailed analysis of this issue could be a direction for future work. We also expect to be able to extend our approach to the evaluation of more general XPath queries, such as those involving the data values, by adapting its underlying mechanism based on labeling.
References

8. G. Gottlob, C. Koch, Monadic Queries over Tree-Structured Data, In Proc. of LICS’02, IEEE.
15. F. Neven, Automata Theory for XML Researchers, In SIGMOD Record 31(3), September 2002.
Appendix I: From Canonical Forms to Standard Forms

We stick to the notations of Section 3.1. Given any canonical XPath expression $Q_{can}$, we compute, inductively, an equivalent standard XPath expression denoted as $Std(Q_{can})$; as earlier, conn stands for either of the boolean connectives and, or.

To start with, we define:

$$Std([true]) = self::*$$

$$Std([S_{can}]) = S_{can}$$

$$Std([A::σ[S_{can}]]) = A::*[self::σ and Std([S_{can}])]$$

$$Std([A::σ[A_1::σ_1[...A_k::σ_k]...]]) = A::*[self::σ and A_1::*[self::σ_1 and... A_{k-1}::*[self::σ_{k-1} and A_k::σ_k] ...]]$$

We also define, for every basic axis relation, an inverse relation, as follows:

$$self^{-1} = self$$

$$child^{-1} = parent$$

$$parent^{-1} = child$$

$$ancestor^{-1} = descendant$$

$$descendant^{-1} = ancestor$$

$$following-sibling^{-1} = preceding-sibling$$

$$preceding-sibling^{-1} = following-sibling$$

For any query $Q = // * [X]$ in standard form, we set $exp(Q) = X$. For any canonical XPath query $Q = /C_1/C_2/.../C_n$, the standard form $Std(Q)$ of $Q$ is then generated by the following recursive construction:

**Case of length 1: $Q = /C_1$**

$$Std(/child::σ[X_{can}]) = // *[self::σ and Std([X_{can}])]$$

$$Std(/child::*[X_{can}]) = // *[self::σ and Std([X_{can}])]$$

$$Std(/descendant::σ[X_{can}]) = // *[self::σ and Std([X_{can}])]$$

$$Std(/descendant::*[X_{can}]) = // *[self::σ and Std([X_{can}])]$$

$$Std/axis::σ[X_{can}] conn axis':::σ'[X_{can}'] =$$

$$// *[exp(Std(/axis::σ[X_{can}])) conn exp(Std(/axis':::σ'[X_{can}']))]$$

**Case of length $n>1$: $Q = /C_1/C_2/.../C_n$**

$$Std(/C_1/.../C_{n-1}/A::σ[X_{can}]) =$$

$$// *[self::σ and Std([X_{can}])] and A^{-1}::*[exp(Std(/C_1/.../C_{n-1}))]$$

$$Std(/C_1/.../C_{n-1}/A::σ[X_{can}] conn A':::σ'[X_{can}']) =$$

$$// *[self::σ and Std([X_{can}']) conn self::σ and Std([X_{can}'])]$$

$$and A^{-1}::*[exp(Std(/C_1/.../C_{n-1}))]$$

This translation procedure is of complexity linear with respect to the total number of location steps (i.e. of the form $axis::σ$) that appear in $Q$. 

---

24
Appendix II: Constructing the Maximal Priority Run

Given a tredag $t$, and a basic non-sibling query $Q$ we give here the algorithm constructing the maximal priority run of the automaton $A_Q$ on $D_t$. (It is trivial for the sibling queries.) Let $n$ be the number of nodes on $D_t$; The idea is to construct a directed acyclic graph $G = (V, E)$, representing all possible (complete) runs of $A_Q$ on $D_t$, and at the same time choose the one of maximal priority.

The set $V$ of vertices of $G$ are elements of the form $A_{i,(l,x)}$, $1 \leq i \leq n$, where $A_i$ is a non-terminal of $L_t$, and $(l, x)$ is an il-pair such that $(l, x)$ is in $States(A_Q)$ such that there exists a (complete) maximal priority run $r$ of $A_Q$ on $D_t$ with $r(A_i) = (l, x)$. The algorithm has four steps as described below (where $lab(A_i)$ is the first component of the current label at the node on $D_t$ named $A_i$).

Step i): For every non-terminal $A_i$ of $L_t$, $1 \leq i \leq n$ compute the set $Parents(A_i)$ of all its parent non-terminals. To render the presentation uniform, we shall introduce a ‘fictional’ symbol $A_0$, and set $Parents(A_1) = \{A_0\}$.

The cost of this step is $O(n^3)$: For every $A_i$ we have to check if $A_i$ is in $Sons(A_j)$ for $j \in \{1, \ldots, i - 1\}$. The maximal size of $Sons(A_j)$ can be $(n - j)$ thus, for $A_i$ we have $\sum_{j=1}^{n}(n - j) = (i - 1)n - \sum_{j=1}^{i-1} j \in \mathcal{O}(i(n - i))$. In total we get: $\sum_{i=1}^{n} i(n - i) \leq n \sum_{i=1}^{n} i \in \mathcal{O}(n^3)$.

Step ii): Construct the set of vertices and the arcs of the graph $G = (V, E)$.

For every $i \in \{1, \ldots, n\}$ we have at most three different $(l, x)$ determined by $llab(A_i)$, so constructing the vertices costs $\mathcal{O}(3n) = \mathcal{O}(n)$.

Next, a pair $(A_{i,(l,x')}, A_{i,(l,x)})$ will be an arc in $E$ iff $A_{j,(l,x')}, A_{j,(l,x)}$ are in $V$, and the rule $((l,x'), llab(A_{j})) \rightarrow (l,x)$ is a transition of $A_Q$. So, for every given vertex $A_{j,(l,x')} \in V$, we construct the set $Target(A_{j,(l,x')})$ of indices of all non-terminals $A_i$, reachable (with some label), from $A_{j,(l,x)}$ on $G$. The cost of this construction is in $O(n^2)$; indeed, for every vertex $A_{j,(l,x')}$ and for every $A_{j,(l,x)} \in V$ such that $A_j \in Parents(A_i)$, we have to check if the former is reachable from the latter; that gives us $3\#(Parents(A_i))$ transitions of $A_Q$ to check, so, on the whole, we get: $3 \sum_{i=1}^{n} \#(Parents(A_i)) = 9 \sum_{i=1}^{n} (i - 1) \in \mathcal{O}(n^2)$.

Step iii): Eliminate the incomplete runs.

Consider any two given $i, j \in \{1, \ldots, n\}$, and suppose $A_{j,(l,x')}$ is a vertex such that $j \in Parents(A_i)$ but $i \notin Target(A_{j,(l,x')})$; this means $A_i$ is not reachable from the former vertex if we associate the label $(l, x')$ to $A_j$; and this implies that the vertex $A_{j,(l,x')}$ is incorrect; we remove from $G$ all incorrect vertices, and all the incoming arcs at such vertices (by a bottom-up traversal of $G$).

Finding all the incoming arcs at a given $A_{i,(l,x)}$, costs $3\#Parents(A_i) = 3(i - 1)$ checks; so, the total cost of this step amounts to $3 \sum_{i=1}^{n} 3(i - 1) \in \mathcal{O}(n^2)$.

Step iv): Choose the maximal priority run.

Begin by setting $r(A_0) = init$; then, for every $A_i$, $i \in \{1, \ldots, n\}$, associate a pair $(l, x)$ such that: $A_{i,(l,x)} \in V$ and $(l, x)$ is of maximal priority between all possible pairs $(l, x')$ such that $A_{i,(l,x')} \in V$. The cost of this last step is in $\sum_{i=1}^{n} 3(i - 1) \in \mathcal{O}(n^2)$.

The total cost of the algorithm is therefore $\mathcal{O}(n^3)$.
Given:

The dependency dag $D_i$, $n$ = its number of nodes;
$A_i$, $1 \leq i \leq n$: the non-terminals of $L_i$ naming the nodes of $D_i$;
$A_Q$ the automaton for the basic query $Q$,
$States$ = the set of states of $A_Q$, $\Delta$ = the set of transition rules of $A_Q$.

BEGIN:
/* Step i): Construct the relation Parents */
Parents($A_1$) := \{ $A_0$ \}; for all $i \in \{2, \ldots, n\}$ Parents($A_i$) := $\emptyset$;
for $i \in \{2, \ldots, n\}$ do
for $j \in \{1, \ldots, i-1\}$ do
if $A_i \in Sons(A_j)$ Parents($A_i$) := Parents($A_i$) $\cup$ $\{A_j\}$;
end; end;
/* Step ii): Construct the vertices $V$, the arcs $E$, and the Target sets */
$V := \{ A_{0,init} \} \cup \{ A_{i,(l,x)} \mid 1 \leq i \leq n, l \text{ defined by } llab(A_i), (l,x) \in States \}$
$E := \emptyset$; for all $A_{i,(l,x)} \in V$ Target($A_{i,(l,x)}$) := $\emptyset$;
for all $A_{i,(l,x)} \in V$, with $i \geq 1$
for all $A_j \in Parents(A_i)$
if $A_j,(l',x') \in V$ and $((l',x'),llab(A_j)) \rightarrow (l,x) \in \Delta$ {
$E := E \cup \{(A_j,(l',x'), A_{i,(l,x)})\}$;
Target($A_{j,(l',x')} := Target(A_{j,(l',x')}) \cup \{i\}$;
}
/* Step iii): Eliminate the incomplete runs, via a bottom-up search */
i := $n$;
for all $A_{i,(l,x)}, A_{i,(l',x')} \in V$
if Target($A_{i,(l,x)}$) $\not\subseteq$ Target($A_{i,(l',x')}$) {
$V := V \setminus \{A_{i,(l,x)}\}$;
for all $A_{k,(l,x')} \in V$ such that $(A_{k,(l,x')}, A_{i,(l,x)}) \in E$ {
$E := E \setminus \{(A_{k,(l,x')}, A_{i,(l,x)})\}$;
if $E \cap \{(A_{k,(l,x')}, A_{i,(l,y)}) \mid (l,y) \text{ defined by } llab(A_i) = \emptyset$
Target($A_{k,(l,x')} := Target(A_{k,(l,x')}) \setminus \{i\}$;
}
i := $i - 1$;
if $i \geq 1$ GOTO (#);
/* Step iv): Construct now the maximal priority run, top-down */
r($A_0$) := init; $i := 1$;
($\$) r($A_i$) := max$\{r(A_j) \mid A_j,(l,x) \in V\}$;
if $i < n$, $\{i := i + 1; GOTO (\$)\}$;
else RETURN $\{r(A_1), \ldots, r(A_n)\}$;
END.
Appendix III: A Complete Example

1) We evaluate the query $Q = /{\text{descendant}}::*[{\text{descendant}}::b \, [{\text{parent}}::a]]$ on the partially compressed document $t$, given to the left of Figure 6. Note that we want to select every node having some descendant $b$ with parent $a$. To start with, we first translate $Q$ into standard form, as:

$$Q = //[{\text{descendant}}::*[{\text{self}}::b \, \text{and} \, {\text{parent}}::a]]$$

Fig. 9. Document $t$, its normalized Grammar $L_t$, and the Dependency rlag $D_t$

Figure 7 represents, to the left, the rlag $D_t$ labeled by the automaton for the query $//[{\text{self}}::b]$; to the middle, the same rlag labeled by the automaton for the query $//[{\text{parent}}::a]$; and to the right, the rlag $D'_t$ re-labeled for the conjunction, as explained in Section 6.

Fig. 10.

Figure 8 shows, to the left, the rlag obtained by re-labeling $D'_t$ with the run of the automaton for the query $//[{\text{descendant}}::s]$; and to the right, the (minimal) sub-rlag of $D_t$ formed of nodes marked now by ll-pairs with boolean component 1, and the corresponding answer for the query $Q$ on the document $t$. This final sub-rlag of $D_t$ is obtained by cutting out the nodes where the ll-pairs attached have boolean component 0.

2) On the same document $t$ as above, we consider now the following standard form query:  $Q' = //[{\text{child}}::b \, \text{or} \, {\text{following-sibling}}::b]$  
To the left of Figure 9 is the rlag $D_t$ labeled by the run of the automaton for $//[{\text{child}}::b]$; and to the right, the labeling of the 3 chiblings of $D_t$, by the run of the automaton for $//[{\text{following-sibling}}::b]$.

The two rlags $D'_t, D''_t$ of Figure 10 below, are then obtained by applying the re-labeling functions respectively $lab_r$ and $\lambda_r$ (of subsection 4.2) for these
Fig. 11.

//*(descendant::s)

```
A1 (s, 1)  
  |      |      |
  |      |      |  A5 (T, 0)  
  |      |      |
A2 (q, 0) A3 (s, 1) A4 (q, 0)  
```

//*(child::b)

```
A1 (s, 1)  
  |      |
  |      |
A2 (q, 0) A3 (s, 1) A4 (T, 0)  
```

//*(following-sibling::b)

```
A1 (s, 1)  
  |      |
  |      |
A2 (c, -) A3 (s, 1) A4 (T, 0)  
```

Fig. 12.

```
F1: 
A2 (c, -) (s, 1)  
  |      |      |  A5 (T, 0)  
  |      |      |
A3 (a, -) (s, 1) A4 (b, -) (T, 1)  
```

```
F0: 
A1 (f, -) (q, 0)  
  |      |      |  A6 (b, -) (T, 0)  
  |      |      |      |  A5 (T, 0)  
  |      |      |      |      |  A4 (T, 0)  
```

```
F3: 
A6 (b, -) (T, 0)  
  |      |      |  A5 (T, 0)  
  |      |      |
A2 (c, -) (q, 0)  
```

Fig. 13.

respective runs. The tag $D_t''$ to the right is obtained by the run of automaton for //∗ [self::s] on $D_t'$. 

```
D_t'  
A1 (s, 1)  
  |      |
  |      |
A2 (q, 0) A3 (s, 1) A4 (q, 0)  
```

```
D_t''  
A1 (q, 1)  
  |      |
  |      |
A2 (s, -) A3 (s, 1) A4 (s, -)  
```

```
D_t'''  
A1 (q, 1)  
  |      |
  |      |
A2 (q, 1) A3 (q, 1) A4 (q, 1)  
```

Fig. 13.
The Figure 11 presents the final answer for our query obtained by applying the function \( OR \) from section 6.

\[
\begin{align*}
D'_t & \quad \text{OR} \quad D''_t' & \quad \text{answer} \\
A_1 (s, \Delta) & \quad \text{OR} \quad A_1 (t, \Delta) & \quad A_3 (s, \Delta) \\
A_2 (t, \Delta) & \quad A_4 (s, \Delta) & \quad A_3 (t, \Delta) \\
A_5 (t, \Delta) & \quad A_5 (s, \Delta) & \quad A_4 (s, \Delta) \\
A_5 (t, \Delta) & \quad A_5 (s, \Delta) & \quad A_4 (s, \Delta) \\
\end{align*}
\]

Fig. 14.