Twig Pattern Matching on Tree Signatures

Federica Mandreoli∗ Pavel Zezula† Riccardo Martoglia ‡

Abstract

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conform to the labelled-tree data model. In this paper we deal with the three problems of pattern matching (path, ordered and unordered twig matching) by exploiting the tree signature approach [ZADR03], which has originally been proposed for the ordered tree matching. In particular, we propose pattern matching algorithms which use auxiliary structures in main memory to store partial results. In order to avoid slowdowns or even overflow, such structures should be maintained as compact as possible by putting only the useful nodes and deleting the ones which are no longer necessary. We thus first introduce a formal framework defining the reduction policies for the management of such auxiliary structures. Finally an experimental analysis of the proposed algorithms, based on real and synthetic data, is also provided.

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conform to the labelled-tree data model. The idea behind evaluating tree pattern queries, sometimes called the twig queries, is to find all existing ways of embedding the pattern in the data. From the formal point of view, three main types of pattern matching exist: One involving paths and two involving trees. XML data objects can be seen as ordered labelled trees, so the problem can be characterized as the ordered tree pattern matching, of which the path pattern matching can be seen as a particular case. Though there are certainly situations where the ordered tree pattern matching perfectly reflects the information needs

∗DII, Univ. di Modena e Reggio Emilia, Italy
†Masaryk University, Brno, Czech Republic
‡DII, Univ. di Modena e Reggio Emilia, Italy
of users, there are many other that would prefer to consider query trees as unordered. For example, when searching for a twig of the element person with the subelements first name and last name (possibly with specific values), ordered matching would not consider the case where the order of the first name and the last name is reversed. However, this could exactly be the person we are searching for. The way to solve this problem is to consider the query twig as an unordered tree in which each node has a label and where only the ancestor-descendant relationships are important – the preceding-following relationships are unimportant. This is called unordered tree pattern matching.

In general, since XML data collections can be very large, efficient evaluation techniques for all types of tree pattern matching are needed. A naive approach to solve the problem is to first decompose complex relationships into binary structural relationships (parent-child or ancestor descendent) between pairs of nodes, then match each of the binary relationships against the XML database, and finally complete together results from those basic matches.

The main disadvantage of such decomposition based approach is that intermediate result sizes can become very large, even for quite small search results. Another disadvantage is that users must wait long to get (even partial) results. In order to avoid such problems, the holistic twig join approach was proposed [BKS02]. In order to compactly represent partial results of individual query root-to-leaf paths, a chain of linked stacks is used as a structure. Then the algorithm merges the sorted lists of participating element sets together and in this way avoids creating large intermediate results. Such approach was further improved by using additional index structures on element sets to quickly locate the first match for a sub-twig pattern, see for example [CVZ02, JWLY03].

Another interesting way to support tree data processing tries to make the relational system aware of tree-structured data [GKT03, GKT04]. With local changes to the traditional (relational) database kernel, the system is easily able to identify a subtree size, intersection of paths, inclusion or disjointness of subtrees, and improves the performance of XPaths, in general.

Most of the advanced solutions share two characteristics: (i) they are based on a coding scheme, which retains structural relationships between tree elements; (ii) they process the supporting data structures in a sequential way, skipping areas of obviously no query answers, whenever possible. The generic trends aiming at efficiency of matching call for compact structures, stored possibly in main memory, and at evaluation algorithms able to provide (partial) solutions as soon as possible. The obvious key to success is to skip
as much of the underlying data as possible, and at the same time never return back in the processed sequence. This is fundamental for the important segment of XML applications processing data streams [KSSS04]. However, some data must be retained in special structures to make the reasoning possible. In order to build correct and efficient twig-matching algorithms, strong theoretical bases must be established so that the skipped area can be maximized with the minimum amount of retained data.

In this paper we deal with the three problems of pattern matching (path, ordered and unordered twig matching) by exploiting the tree signature approach [ZADR03], which has originally been proposed for the ordered tree matching. By taking advantage of the tree signature properties (Section 2), and in particular of the involved pre/post order coding scheme and of its sequential nature, we first characterize the pattern matching problems from a pre- and post-order point of view (Section 2) and then we show:

- that the pre/post-order ranks are sufficient to define a complete set of conditions under which a data node accessed at a given step is no longer necessary for the subsequent generation of a matching solution (Section 3);
- the properties of the twig matching solutions, generated at each step of the scanning (Section 3);
- how such conditions can be used to write pattern matching algorithms that are correct and which, from a numbering scheme point of view, cannot be further improved (Section 4).

Finally, we verify the theoretical results by extensive experimental evaluation performed on real and synthetic data (Section 5) before concluding the paper (Section 6).

2 Tree Signatures

The idea of tree signatures proposed in [ZADR03] is to maintain a small but sufficient representation of the tree structures able to decide the ordered tree inclusion problem for the XML data processing. As a coding schema, the preorder and postorder ranks [Die82] are used. In this way, tree structures are linearized, and extended string processing algorithms are applied to identify the tree inclusion.

An ordered tree $T$ is a rooted tree in which the children of each node are ordered. If a node $v \in T$ has $k$ children then the children are uniquely
A labelled tree $T$ associates a label (name) $t_v \in \Sigma$ (the domain of tree node labels) with each node $v \in T$. If the path from the root to $v$ has length $n$, we say that the node $v$ is on the level $n$, i.e. $\text{level}(v) = n$. Finally, $\text{size}(v)$ denotes the number of nodes rooted at $v$ – the size of any leaf node is zero. In this section, we consider ordered labelled trees.

The preorder and postorder sequences are ordered lists of all nodes of a given tree $T$. In a preorder sequence, a tree node $v$ is traversed and assigned its (increasing) preorder rank, $\text{pre}(v)$, before its children are recursively traversed from left to right. In the postorder sequence, a tree node $v$ is traversed and assigned its (increasing) postorder rank, $\text{post}(v)$, after its children are recursively traversed from left to right. For illustration, see the preorder and postorder sequences of our sample tree in Fig. 1 – the node’s position in the sequence is its preorder/postorder rank, respectively.

Given a node $v \in T$ with $\text{pre}(v)$ and $\text{post}(v)$ ranks, the following properties are important towards our objectives:

- all nodes $x$ with $\text{pre}(x) < \text{pre}(v)$ are the ancestors or preceding nodes of $v$;
- all nodes $x$ with $\text{pre}(x) > \text{pre}(v)$ are the descendants or following nodes of $v$;
- all nodes $x$ with $\text{post}(x) < \text{post}(v)$ are the descendants or preceding nodes of $v$;
- all nodes $x$ with $\text{post}(x) > \text{post}(v)$ are the ancestors or following nodes of $v$;

Figure 1: Preorder and postorder sequences of a tree
Figure 2: Properties of the preorder and postorder ranks.

- For any \( v \in T \), we have \( \text{pre}(v) - \text{post}(v) + \text{size}(v) = \text{level}(v) \);

- If \( \text{pre}(v) = 1 \), \( v \) is the root, if \( \text{pre}(v) = n \), \( v \) is a leaf. For all the other neighboring nodes \( v_i \) and \( v_{i+1} \) in the preorder sequence, if \( \text{post}(v_{i+1}) > \text{post}(v_i) \), \( v_i \) is a leaf.

As proposed in [Gru], such properties can be summarized in a two dimensional diagram. See Fig. 2 for illustration, where the ancestor (A), descendant (D), preceding (P), and following (F) nodes of \( v \) are strictly located in the proper regions. Notice that in the preorder sequence all descendant nodes (if they exist) form a continuous sequence, which is constrained on the left by the reference node and on the right by the first following node (or the end of the sequence). The parent node of the reference is the ancestor with the highest preorder rank, i.e. the closest ancestor of the reference.

2.1 The Signature

The tree signature is a list of entries for all nodes in acceding preorder. Except the node name, each entry also contains the node’s position in the postorder sequence.

**Definition 1** Let \( T \) be an ordered labelled tree. The signature of \( T \) is a sequence, \( \text{sig}(T) = \langle t_1, \text{post}(t_1); t_2, \text{post}(t_2); \ldots t_n, \text{post}(t_n) \rangle \), of \( n = |T| \) entries, where \( t_i \) is a name of the node with \( \text{pre}(t_i) = i \). The \( \text{post}(t_i) \) value is the postorder value of the node named \( t_i \) and the preorder value \( i \).

Observe that the index in the signature sequence is the node’s preorder, so the value serves actually two purposes. In the following, we use the term preorder if we mean the rank of the node, when we consider the position
of the node’s entry in the signature sequence, we use the term index. For example, \( \langle a, 10; b, 5; c, 3; d, 1; e, 2; g, 4; f, 9; h, 8; o, 6; p, 7 \rangle \) is the signature of the tree from Fig. 1. The first signature element \( a \) is the tree root. Leaf nodes in signatures are all nodes with postorder smaller than the postorder of the following node in the signature sequence, that is nodes \( d, e, g, o \) – the last node, in our example it is node \( p \), is always a leaf. We can also determine the level of leaf nodes, because the \( \text{size}(i) = 0 \) for all leaves \( i \), thus \( \text{level}(i) = i - \text{post}(i) \).

### 2.1.1 Extended Signatures

By extending entries of tree signatures with two preorder numbers representing pointers to the first following, \( ff \), and the first ancestor, \( fa \), nodes, the extended signatures are defined in [ZADR03]. The generic entry of the \( i \)-th extended signature entry is \( \langle t_i, \text{post}(t_i), ff_i, fa_i \rangle \). Such version of the tree signatures makes possible to compute levels for any node as: \( \text{level}(i) = ff_i - \text{post}(i) - 1 \). The cardinality of the descendant node set can also be computed: \( \text{size}(i) = ff_i - i - 1 \). For the tree from Fig. 1, the extended signature is: \( \text{sig}(T) = \langle a, 10, 11; b, 5, 7; c, 3, 6; d, 1, 5, 3; e, 2, 6, 3; g, 4, 7, 2; f, 9, 11, 1; h, 8, 11, 7; o, 6, 10, 8; p, 7, 11, 8 \rangle \).

### 2.1.2 Sub-Signatures

A sub-signature \( \text{sub}_s i g s(T) \) is a specialized (restricted) view of \( T \) through signatures, which retains the original hierarchical relationships of elements in \( T \). Specifically, \( \text{sub}_s i g s(T) = \langle t_{s_1}, \text{post}(t_{s_1}); t_{s_2}, \text{post}(t_{s_2}); \ldots t_{s_k}, \text{post}(t_{s_k}) \rangle \) is a sub-sequence of \( \text{sig}(T) \), defined by the ordered set \( S = (s_1, s_2, \ldots s_k) \) of indexes (preorder values) in \( \text{sig}(T) \), such that \( 1 \leq s_1 < s_2 < \ldots < s_k \leq n \). Naturally, the set operations of the union and the intersection can be applied on sub-signatures provided the sub-signatures are derived from the same signatures and the results are kept sorted. For example consider two sub-signatures of the signature representing the tree from Fig. 1, defined by ordered sets \( S_1 = (2, 3, 4) \) and \( S_2 = (2, 3, 5, 6) \). The union of \( S_1 \) and \( S_2 \) is the set \( (2, 3, 4, 5, 6) \), that is the sub-signature representing the subtree rooted at the node \( b \) of our sample tree.

### 2.2 Twig pattern inclusion evaluation

The problem of twig pattern inclusion evaluation on tree signature can be seen as a problem of finding all sub-signatures of a given data signature matching with the twig pattern at node name level and satisfying some of
the relationships of parent-child (ancestor-descendant) and sibling between the nodes.

In the following we will consider three kind of twig pattern inclusion evaluation: ordered tree inclusion, path inclusion, and unordered tree inclusion.

2.2.1 Ordered tree inclusion evaluation

Let $D$ and $Q$ be ordered labelled trees. The tree $Q$ is included in $D$, if $D$ contains all elements (nodes) of $Q$ and when the sibling and ancestor relationships of the nodes in $D$ are the same as in $Q$. Using the concept of signatures, we can formally define the ordered tree inclusion problem as follows. Suppose the data tree $D$ and the query tree $Q$ specified by signatures

$$\text{sig}(D) = \langle d_1, \text{post}(d_1); d_2, \text{post}(d_2); \ldots; d_m, \text{post}(d_m) \rangle,$$

$$\text{sig}(Q) = \langle q_1, \text{post}(q_1); q_2, \text{post}(q_2); \ldots; q_n, \text{post}(q_n) \rangle.$$

Let $\text{sub}_{\text{sig}}(D)$ be the sub-signature (i.e. a subsequence) of $\text{sig}(D)$ induced by a name sequence-inclusion of $\text{sig}(Q)$ in $\text{sig}(D)$ – a specific query signature can determine zero or more data sub-signatures. Regarding the node names, any $\text{sub}_{\text{sig}}(D) \equiv \text{sig}(Q)$, because $q_i = d_{s_i}$ for all $i$, but the corresponding entries can have different postorder values. The following lemma defines the necessary constrains for qualification.

**Lemma 1** The query tree $Q$ is included in the data tree $D$ in an ordered fashion if the following two conditions are satisfied: (1) on the level of node names, $\text{sig}(Q)$ is sequence-included in $\text{sig}(D)$ determining $\text{sub}_{\text{sig}}(D)$ through the ordered set of indexes $S = (s_1, s_2, \ldots, s_n)$ where $s_1 < \ldots < s_n$, (2) for all pairs of entries $i$ and $j$ in $\text{sig}(Q)$ and $\text{sub}_{\text{sig}}(D)$, $i, j = 1, 2, \ldots, |Q| - 1$ and $i + j \leq |Q|$, whenever $\text{post}(q_{i+j}) > \text{post}(q_i)$ it is also true that $\text{post}(d_{s_{i+j}}) > \text{post}(d_{s_i})$ and whenever $\text{post}(q_{i+j}) < \text{post}(q_i)$ it is also true that $\text{post}(d_{s_{i+j}}) < \text{post}(d_{s_i})$.

**Proof.** Because the index $i$ increases according to the preorder sequence, node $i + j$ must be either the descendent or the following node of $i$. If $\text{post}(q_{i+j}) < \text{post}(q_i)$, the node $i + j$ in the query is a descendent of the node $i$, thus also $\text{post}(d_{s_{i+j}}) < \text{post}(d_{s_i})$ is required. By analogy, if $\text{post}(q_{i+j}) > \text{post}(q_i)$, the node $i + j$ in the query is a following node of $i$, thus also $\text{post}(d_{s_{i+j}}) > \text{post}(d_{s_i})$ must hold. 

Observe that Lemma 1 defines a weak inclusion of the query tree in the data tree, in the sense that the parent-child relationships of the query are
implicitly reflected in the data tree as only the ancestor-descendant. However, due to the properties of preorder and postorder ranks, such constraints can easily be strengthened, if required.

For example, consider the data tree \( D \) in Fig. 1 and the query tree \( Q \) in Fig. 3. Such query qualifies in \( D \), because \( \text{sig}(Q) = \langle h, 3; o, 1; p, 2 \rangle \) determines \( \text{sub}_\text{sig}(T) = \langle h, 8; o, 6; p, 7 \rangle \) through the ordered set \( S = (8, 9, 10) \), because (1) \( q_1 = d_8 \), \( q_2 = d_9 \), and \( q_3 = d_{10} \), (2) the postorder of node \( h \) is higher than the postorder of nodes \( o \) and \( p \), and the postorder of node \( o \) is smaller than the postorder of node \( p \) (both in \( \text{sig}(Q) \) and \( \text{sub}_\text{sig}(T) \)). If we change in our query tree \( Q \) the label \( h \) for \( f \), we get \( \text{sig}(Q) = \langle f, 3; o, 1; p, 2 \rangle \). Such a modified query tree is also included in \( D \), because Lemma 1 does not insist on the strict parent-child relationships, and implicitly considers all such relationships as ancestor-descendant. However, the query tree with the root \( g \), resulting in \( \text{sig}(Q) = \langle g, 3; o, 1; p, 2 \rangle \), does not qualify, even though it is also sequence-included (on the level of names) as the sub-signature \( \text{sub}_\text{sig}(D) = \langle g, 4; o, 6; p, 7 \rangle \mid S = (6, 9, 10) \). The reason is that the query requires the postorder to go down from \( g \) to \( o \) (from 3 to 1), while in the sub-signature it actually goes up (from 4 to 6). That means that \( o \) is not a descendant node of \( g \), as required by the query, which can be verified in Fig. 1.

Multiple nodes with common names may result in multiple tree inclusions. As demonstrated in [ZADR03], the tree signatures can easily deal with such situations just by simply distinguishing between node names and their unique occurrences.

2.2.2 Path inclusion evaluation

The path inclusion evaluation is a special case of the ordered tree inclusion evaluation as all the relationships between the nodes in any path \( P \) are of parent-child (ancestor-descendant) type. Following the numbering scheme of a path \( P \) signature \( \text{sig}(P) = \langle t_1, \text{post}(t_1); \ldots; t_n, \text{post}(t_n) \rangle \), it means that the post-order values of subsequent entries \( i \) and \( j \) \((i, j = 1, 2, \ldots n - 1 \) and
$i + j \leq n$) satisfy the inequality $\text{post}(q_i) < \text{post}(q_{i+j})$. The lemma below easily follows from the above observation and from the fact that inequalities are transitive.

**Lemma 2** A path $P$ is included in the data tree $D$ if the following two conditions are satisfied: (1) on the level of node names, $\text{sub}	ext{-}\text{sig}_P(Q)$ is sequence-included in $\text{sig}(D)$ determining $\text{sub}	ext{-}\text{sig}_S(D)$ through the ordered set of indexes $S = (s_1, \ldots, s_n)$ where $s_1 < \ldots < s_n$, (2) for each $i \in [1, |P| - 1]$: $\text{post}(d_{s_i}) < \text{post}(d_{s_{i+1}})$.

### 2.2.3 Unordered tree inclusion evaluation

Let $Q$ and $D$ be ordered labelled trees. An unordered tree inclusion of $Q$ in $D$ is identified by a total mapping from nodes in $Q$ to some nodes in $D$, such that only the ancestor-descendent structural relationships between nodes in $Q$ are satisfied by the corresponding nodes in $D$. The unordered tree inclusion evaluation essentially searches for a node mapping keeping the ancestor-descendent relationships of the query nodes in the target data nodes. Using the concept of signature, the query tree $Q$ is included in the data tree $D$ in an unordered fashion if at least one qualifying index set exists.

**Lemma 3** The query tree $Q$ is included in the data tree $D$ in an unordered fashion if the following two conditions are satisfied: (1) on the level of node names, an ordered set of indexes $S = (s_1, s_2, \ldots, s_n)$ exists, $1 \leq s_i \leq m$ for $i = 1, \ldots, n$, such that $d_{s_i} = q_i$, for $i = 1, \ldots, n$, (2) for all pairs of entries $i$ and $j$, $i, j = 1, 2, \ldots, |Q| - 1$ and $i + j \leq |Q|$, if $\text{post}(q_{i+j}) < \text{post}(q_i)$ then $\text{post}(d_{s_{i+j}}) < \text{post}(d_{s_i}) \land s_{i+j} > s_i$.

Notice that the index set $S$ is ordered but, unlike the ordered inclusion of Lemma 1, indexes are not necessarily in an increasing order. In other words, an unordered tree inclusion does not necessarily imply the node-name inclusion of the query signature in the data signature. Should the signature $\text{sig}(Q)$ of the query not be included on the level of node names in the signature $\text{sig}(D)$ of the data, $S$ would not determine the qualifying sub-signature $\text{sub}	ext{-}\text{sig}_S(D)$. Anyway, as shown in [ZMM04], any $S$ satisfying the properties specified in Lemma 3 can always undergo a sorting process in order to determine the corresponding sub-signature of $\text{sig}(D)$ qualifying the unordered tree inclusion of $Q$ in $D$. 

---

9
3 A formal account of twig pattern matching

Given a twig pattern \( Q \) and a data tree \( D \), represented by the signatures

\[
sig(Q) = \langle q_1, \text{post}(q_1); q_2, \text{post}(q_2); \ldots; q_n, \text{post}(q_n) \rangle
\]

and

\[
sig(D) = \langle d_1, \text{post}(d_1); d_2, \text{post}(d_2); \ldots; d_m, \text{post}(d_m) \rangle
\]

we denote with \( \text{ans}_Q(D) \) the set of answers to the ordered inclusion of \( Q \) in \( D \) and with \( \text{Uans}_Q(D) \) the set of answers to the unordered inclusion of \( Q \) in \( D \). For the sake of brevity we will use the notation \( (U)\text{ans}_Q(D) \) to designate situations which apply to both the cases. Obviously, if \( Q \) is a path \( P \) then \( \text{ans}_P(D) = \text{Uans}_P(D) \). In previous section, we have shown the properties an index set must satisfy so that it is an answer to the inclusion of \( Q \) in \( D \). In all the three cases, the matching on the level of node name is required. Let \( \Sigma_i \) designate the set (domain) of all positions \( j \) in the data signature where the query node name \( q_i \) occurs (i.e. \( d_j = q_i \)). The set of answers \( (U)\text{ans}_Q(D) \) is a subset of the Cartesian product of the domains \( \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n \) determined by Lemma 1 or Lemma 3, respectively. Obviously, if one of the \( \Sigma_i \) sets is empty, \( Q \) is not included in \( D \), because the Cartesian product is empty.

A naïve strategy to compute the desired Cartesian product subset is to first compute the sets \( \Sigma_i \), for \( i \in [1, n] \) and then discard from the Cartesian product \( \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n \) tuples, the corresponding sub-signatures of which do not satisfy the properties required by specific pattern matching constraints. The intrinsic limitation of this approach is twofold: it can produce very large intermediate results and, even more important, such evaluation procedure does not exploit the sequential nature of tree signatures.

In this paper we present algorithms that perform the twig pattern matching by sequentially scanning the data signature. Our algorithms exploit properties of the preorder/postorder numbering scheme adopted in the construction of tree signatures.

At each step \( j \) of the sequential scan of \( D \), for \( j = 1, \ldots, m \), \( \Sigma_i^j \) denotes the set of all positions in the range from 1 up to \( j \) where \( q_i \) occurs and \( (U)\text{ans}^j_Q(D) \) denotes the set of answers to the (un)ordered inclusion of the twig pattern \( Q \) in \( D \) computable from the domains \( \Sigma_1^j, \Sigma_2^j, \ldots, \Sigma_n^j \). Notice that the sets \( \text{ans}^j_Q(D) \) and \( \text{Uans}^j_Q(D) \) are subsets of the Cartesian product \( \Sigma_1^j \times \Sigma_2^j \times \ldots \times \Sigma_n^j \) and that \( \text{ans}^j_Q(D) \subseteq \text{Uans}^j_Q(D) \) as any answer to the ordered inclusion of \( Q \) in \( D \) is an answer to the unordered
\[\text{sig}(D) < d_1, \text{post}(d_1); \ldots; d_k, \text{post}(d_k); \ldots; d_m, \text{post}(d_m) >\]

\[\text{sig}(Q) < q_1, \text{post}(q_1); \ldots; q_h, \text{post}(q_h); \ldots; d_k, \text{post}(d_n) >\]

\[\Delta \Sigma^k_1 \ldots \Delta \Sigma^k_h \ldots \Delta \Sigma^k_n\]

Figure 4: Snapshot in the sequential scanning

inclusion of \(Q\) in \(D\). Obviously, an inclusion relationship between the answer sets holds, 
\((U)\text{ans}_Q^1(D) \subseteq (U)\text{ans}_Q^2(D) \subseteq \ldots \subseteq (U)\text{ans}_Q^m(D)\), and the last answer set is the set of answers to the (un)ordered inclusion of \(Q\) in \(D\), 
\((U)\text{ans}_Q^n(D) = (U)\text{ans}_Q(D)\). Thus, the set of answers can be incrementally constructed by sequentially scanning the data signature. In principle, at each step new answers can be added to the answer set of the earlier one. Assuming \(\text{ans}_Q^0(D) = \emptyset\), we denote with \(\Delta(U)\text{ans}_Q^j(D) = (U)\text{ans}_Q^j(D) \setminus (U)\text{ans}_Q^{j-1}(D)\) the set of answers, which can be computed (decided) at step \(j\) and which have not been computed at an earlier step. Notice that the set of answers to the inclusion of \(Q\) in \(D\) can also be written as 
\((U)\text{ans}_Q(D) = \bigcup_{j=1}^m \Delta(U)\text{ans}_Q^j(D)\). Also notice that \(\Delta\text{ans}_Q(D) \subseteq \Sigma_1^j \times \Sigma_2^j \times \ldots \times \Sigma_n^j\). On the other hand, not all indexes (representing data nodes of \(D\)) in \(\Sigma_i^j\) are necessary for the generation of \(\Delta(U)\text{ans}_Q^j(D)\). Thus we can denote with \(\Delta\Sigma_i^j\) “reduced” versions of the original domains \(\Sigma_i^j\) which are needed to decide \(\Delta(U)\text{ans}_Q^j(D) \subseteq \Delta\Sigma_1^j \times \Delta\Sigma_2^j \times \ldots \times \Delta\Sigma_n^j\).

For illustration, consider in Figure 3 a snapshot of such sequential scanning process occurring when \(j = k\) and \(d_k\) matches \(q_h\) and thus \(j\) should be added to \(\Sigma_h^k\). In the following, our main aim is to determine the conditions under which a data node is no longer necessary for the generation of the delta answer sets \(\Delta(U)\text{ans}_Q^j(D)\), for each \(j \geq k\), and thus it is not necessary in any \(\Delta\Sigma_i^j\). Moreover, we will characterize the set of answers which can be generated at each step.

3.1 Conditions on pre-orders

Let us first consider pre-order codes and the ordered case. Recall that a sequential scan of a signature means that the data nodes are visited according to their increasing pre-order codes. Moreover, any set of indexes (pre-order values) \(S = (s_1, s_2, \ldots, s_n)\) qualifying the ordered inclusion is also ranked according to pre-order, \(1 \leq s_1 < \ldots < s_n \leq m\), i.e. a total order is required.
A direct consequence of this property is given by the following Lemma (Condition PRO1 - PR stands for PRe-order, O stands for Ordered). It states that a data node matching the last query node \( q_n \) then it will never belong to the solutions that can be computed in the following steps.

**Lemma 4 (Condition PRO1)** If \( h = n \) then \( k \notin S \) for each \( S \in \Delta\text{ans}_Q^j(D) \) for each \( j \in [k+1,m] \). Thus \( k \) does not belong to \( \Delta\Sigma_h^j \).

From the previous Lemma, it follows that \( \Delta\Sigma_h^j \) is always empty but when \( d_k = q_n \) and, in this case, \( \Delta\Sigma_h^k \) only contains \( k \).

Besides the previous Lemma, the following Lemmas states the conditions under which \( d_k \) is no longer necessary for the generation of the delta answers. The first one states that if at the \( k \)-th step a domain \( \Delta\Sigma_h^k \) preceding \( \Delta\Sigma_h^k \) is empty then \( k \) will never belong to the solutions that can be computed in the \( k \)-th step and in the following ones.

**Lemma 5 (Condition PRO2 applied to \( k \))** If \( \Delta\Sigma_h^k = \emptyset, i \in [1,h-1] \), then for each \( S \in \Delta\text{ans}_Q^j(D) \), for each \( j \in [k,m] \): \( k \notin S \). Thus \( k \) does not belong to \( \Delta\Sigma_h^j \).

**Proof.** Let us suppose that \( j \in [k,m] \) exists such that \( S = (s_1,\ldots,s_n) \in \Delta\text{ans}_Q^j(D) \) and \( s_h = k \). Notice that it should be \( s_i < k \) but \( \Delta\Sigma_i^k = \emptyset \) and thus no index \( s_i \) exists such that \( d_{s_i} = q_i \) and \( s_i < k \). □

The second Lemma extends the condition of the previous one to the subsequent steps. Its proof is similar to the previous one.

**Lemma 6 (Condition PRO2 applied to \( k' < k \))** If \( k' \in \Delta\Sigma_h^{k'-1} \) and \( \Delta\Sigma_i^{k'} \cap \Delta\Sigma_i^k = \emptyset, i \in [1,h'-1] \), then for each \( S \in \Delta\text{ans}_Q^j(D) \), for each \( j \in [k,m] \): \( k' \notin S \). Thus \( k' \) does not belong to \( \Delta\Sigma_h^j \).

The following Theorem show that the three previous conditions together constitute the sufficient conditions such that a data node, due to its pre-order value, is no longer necessary.

**Theorem 1 (Completeness)** For the ordered case, beyond the conditions expressed in Lemmas 4, 5, and 6, there is no other condition ensuring that at each step \( k \), any data node due to its pre-order value does not belong to the solutions which will be generated in the following steps.

**Proof.** The proof is ab absurdo. In particular we will show that the following four facts together constitute an absurd:
1. Let \( d_{k'} \) be a data node with \( k' \leq k \) and \( d_{k'} = q_{k'} \), then \( k' \) does not belong to \( \Delta \text{ans}^j_Q(D) \), \( j \in [k, m] \), because for each \( (s_1, \ldots, s_{h'-1}) \in \Delta \Sigma_1^j \times \ldots \times \Delta \Sigma_{h'-1}^j \), either \( i \) exists such that \( s_i > k' \) or given that \((s_1, \ldots, s_{h'-1})\) belongs to an answer \( S \) and \( k \) belongs to it too, then \( S \) can not belong to \( \Delta \text{ans}^j_Q \).

2. for each \( i \in [1, h'-1] \), at step \( k' \), \( \Delta \Sigma_i^{k'} \neq \emptyset \) (condition of Lemma 5)

3. for each \( i \in [1, h'-1] \), at step \( k \), \( \Delta \Sigma_i^{k'} \cap \Delta \Sigma_i^k \neq \emptyset \) (condition of Lemma 6)

4. \( h' \neq n \) (condition of Lemma 4)

Indeed, wherever the facts 2, 3, and 4 are true then \((s_1, \ldots, s_{h'-1}) \in \Delta \Sigma_1^k \times \ldots \times \Delta \Sigma_{h'-1}^k \) exists, such that for each \( i \), \( s_i < k' \). As, for each \( i < h' \), \( \Delta \Sigma_i^{k'} \neq \emptyset \) and \( \Delta \Sigma_i^{k'} \cap \Delta \Sigma_i^k \neq \emptyset \) then \( s_i \in \Delta \Sigma_i^k \) exists such that \( s_i < k' \). Moreover \( h \neq n \) and thus for any solution \( S = (s_1, \ldots, s_h, s_{h+1}, \ldots, s_n) \) such that \( s_{h'} = k' \) it must be \( s_{h+1} > k', \ldots, s_n > k' \) that is \( S \not\in \text{ans}^j_Q(D) \). Therefore, without any knowledge about the data nodes following \( k' \) in the sequential scanning, \( S \) could belong to \( \Delta \text{ans}^j_Q \). □

As to the unordered case, notice that the pre-order values of any qualifying set of indexes are not required to be completely ordered as it is for the ordered evaluation. For this reason, the Lemmas above are no longer sound. However, the unordered evaluation requires a partial order among the pre-order values of a qualifying set of indexes \((s_1, \ldots, s_n)\). In particular, whenever \( \text{post}(q_{i+j}) < \text{post}(q_i) \) it is required that \( \text{post}(d_{s_{i+j}}) < \text{post}(d_{s_i}) \) and that \( s_{i+j} > s_i \). Thus Lemmas 5, and 6 rewritten in the following way are still sound.

**Lemma 7 (Condition PRU applied to \( k \))** If \( \Delta \Sigma_i^k = \emptyset \), \( i \in [1, h-1] \) and \( \text{post}(q_i) > \text{post}(q_{i+j}) \) then for each \( S \in \Delta \text{ans}^j_Q(D) \), for each \( j \in [k, m] \): \( k \notin S \). Thus \( k \) does not belong to \( \Delta \Sigma_i^k \).

**Lemma 8 (Condition PRU applied to \( k' < k \))** If \( k' \in \Delta \Sigma_i^{k'-1}, \Delta \Sigma_i^{k'} \cap \Delta \Sigma_i^k = \emptyset \), and \( \text{post}(q_i) > \text{post}(q'_{i+j}), i \in [1, h'-1] \), then for each \( S \in \Delta \text{ans}^j_Q(D) \), for each \( j \in [k, m] \): \( k' \notin S \). Thus \( k' \) does not belong to \( \Delta \Sigma_i^{k'} \).

On the other hand, there is no counterpart for Lemma 4. Indeed, as no total order among the pre-order values is required, the “position” of the query node matching the data node does not influence the use of such data.
node in the solutions which will be generated in the following steps. More precisely, in the ordered case a concept of “last” query node from a pre-order point of view exists and thus whenever $d_k = q_n$ any solution $(s_1, \ldots, s_{n-1}, k)$ involving $d_k$ is generated at the $k$-th steps as the pre-order values $s_1, \ldots, s_{n-1}$ of all the other data nodes are required to be smaller than $k$ (and thus accessed in the steps preceding the $k$-th). Instead, in the unordered case, the data node $d_k$ accessed at the $k$-th step can always be useful for the generation of the answers in the subsequent steps unless no data node in the delta sets exists for which it is required that its pre-order value is smaller than $k$. This last aspect is considered in the two Lemmas above. For this reason the following Theorem is sound and the proof is similar to that of Theorem 1.

**Theorem 2 (Completeness)** For the unordered case, beyond the conditions expressed in Lemmas 7 and 8, there is no other condition ensuring that at each step $k$, any data node due to its pre-order value does not belong to the solutions which will be generated in the following steps.

In this way, we have shown the sufficient and necessary pre-order conditions for the exclusion of a data node in the generation of the delta solutions of the ordered and unordered inclusion of a query tree in a data tree.

### 3.2 Conditions on post-orders

As far as post-order requirements are involved, here the distinction is between the path matching and the (un)ordered twig matching. Indeed, path matching requires a total order among the post-order values of the data nodes belonging to a match whereas in the twig matching only a partial order is sufficient. Thus, we first introduce some general rules which will then be used to study each of the matching approaches.

First of all, the following Lemma easily follows from the property on post-order values a solution for twig pattern matching must satisfy. It allows us to prevent the introduction of the current data node $d_k$ in the pertinence domain $\Delta \Sigma_h^j$ for the construction of the solutions in the steps from $j = k$ to $j = m$.

**Lemma 9 (Condition POT1)** If $i \in [1, h - 1]$ exists such that for each $s_i \in \Delta \Sigma_{k-1}^i$, $\text{post}(d_k) < \text{post}(d_i)$ is required but $\text{post}(d_k) > \text{post}(d_i)$ or $\text{post}(d_k) > \text{post}(d_i)$ is required but $\text{post}(d_k) < \text{post}(d_i)$ then $k \notin S$, for each $S \in \Delta(U)\text{ans}_Q^j(D)$ for each $j \in [k, m]$. Thus $k$ can be deleted from $\Delta \Sigma_h^j$. 

14
Different is the case of the deletion of a data node preceding the $k$-th in the sequential scanning and thus already belonging to a delta domain. Notice that, as in the preorder case, at the $k$-th step of the sequential scanning, a node belonging to a delta domain will no longer be necessary for the generation of the delta answer sets due to its postorder value if one of the required post-order relationships w.r.t. the other nodes will always be violate by the data nodes following the $k$-th in the sequential scanning. It means that either $d_j$ with $j < k$ has already been used in the generation of the previous delta answer set or it will never be used. In the patter matching definition, two kinds of relationships are taken into account between the post order values of two data nodes $d_i$ and $d_j$: either it is required that $\text{post}(d_i) < \text{post}(d_j)$ or $\text{post}(d_i) > \text{post}(d_j)$. Given the relationship between the post order value of the $k$-th node, $\text{post}(d_k)$, and that of a preceding node $\text{post}(d_j)$ ($j < k$), we want to predict what kind of inequality relationship will hold between the post-order value of $d_j$ and those of the nodes following $d_k$ in the sequential scanning. Only if we are able to do it, we can state that $d_j$ is no longer necessary due to its post-order value. At a first glance, by considering the properties of the preorder and postorder ranks given in Fig. 2, it seems that the post-order relationships between $\text{post}(d_j)$ and $\text{post}(d_k)$ and $\text{post}(d_j)$ and $\text{post}(d_{j'})$ with $j' > k$ are completely independent. It is true when $\text{post}(d_j) > \text{post}(d_k)$ and it can be straightforwardly noticed also in the postorder relationships depicted in Fig. 1: node $b$ precedes node $c$ which in turn precedes nodes $g$ and $f$ in the sequential scanning and, as to postorder values, $\text{post}(b) > \text{post}(c)$ but $\text{post}(b) < \text{post}(f)$ and $\text{post}(b) > \text{post}(g)$. Different is the case of the other postorder relationship $\text{post}(d_j) < \text{post}(d_{k'})$ which is taken into account in the following Lemma.

**Lemma 10** Let $j < k$ and $\text{post}(d_j) < \text{post}(d_k)$. It follows that $\text{post}(d_j) < \text{post}(d_{j'})$, for each $j' \in [k, m]$.

**Proof.** Obviously, due to the premise, $\text{post}(d_j) < \text{post}(d_{j'})$ when $j' = k$. As to $j' \in [k+1, m]$, let us consider the two possible alternatives: either $\text{post}(d_k) < \text{post}(d_{j'})$ or $\text{post}(d_k) > \text{post}(d_{j'})$. In the former case, it easily follows that $\text{post}(d_j) < \text{post}(d_{j'})$. The latter case means that $d_k$ is an ancestor of $d_{j'}$ since $k < j'$. Moreover $d_j$ is a preceding of $d_k$ since $\text{post}(d_j) < \text{post}(d_k)$ and $j < k$. It follows that $d_j$ is a preceding of $d_{j'}$ and thus $\text{post}(d_j) < \text{post}(d_{j'})$. □

Thus, both in the cases of path and (un)ordered twig matching, a data node $d_i$ can be deleted from the pertinence delta domain if and only if it is required that its postorder value is greater than that of another data
node but that condition shall never be verified from a particular step of the scanning process. We deeply analyze such situations in the following.

Let us first consider the case of path matching. Lemma 10 allows the introduction of the following Lemma showing a sufficient condition such that a data node, due to its postorder value, is no longer necessary to generate the answers to the path inclusion evaluation.

**Lemma 11 (Condition POP)** Let \( Q \) be a path \( P \), \( s_i \in \Delta \Sigma^k_i \) where \( i \in [1, n] \) and \( \text{post}(d_{s_i}) < \text{post}(d_k) \). It follows that \( s_i \not\in S \), for each \( S \in \Delta \text{ans}^j_{P}(D) \) for each \( j \in [k, m] \). Thus \( s_i \) can be deleted from \( \Delta \Sigma^j_i \).

**Proof.** Let us suppose that \( j \in [k, m] \) exists such that \( S = (s_1, \ldots, s_n) \in \Delta \text{ans}^j_{Q}(D), s_i \in S \). Notice that \( (s_1, \ldots, s_n) \in \Delta \text{ans}^j_{Q}(D) \) iff \( j \) exist such that \( s_{j'} > k \) for each \( j' \geq j \). Let us consider \( s_j \). Being \( s_i < s_j \) then \( S \) is a solution iff \( \text{post}(d_{s_i}) > \text{post}(d_{s_j}) \). But it is impossible since \( \text{post}(d_{s_i}) < \text{post}(d_k) \) and, from Lemma 10 follows that \( \text{post}(d_{s_i}) < \text{post}(d_{s_j}) \). □

Given the situation depicted in Lemma 9, the previous Lemma produces the same effect on the current data node \( d_k \) as Lemma 9, i.e. its deletion from \( \Delta \Sigma^j_i \). On the other hand, Lemma 11 also acts on the nodes preceding \( d_k \) in the sequential scanning. For this reason, Lemma 11 can be used in place of Lemma 9. The proof is given in the following proposition. Notice that, being the query \( Q \) a path \( P \), we only consider the case \( \text{post}(q_i) < \text{post}(q_h) \).

**Lemma 12** If for each \( i \in [1, h - 1] \), for each \( s_i \in \Delta \Sigma^{k-1}_i \), \( \text{post}(d_{s_i}) < \text{post}(d_k) \) then, due to Lemmas 5 and 11, \( k \not\in S \), for each \( S \in \Delta \text{ans}^j_{P}(D) \) for each \( j \in [k, m] \). Thus \( k \) can be deleted from \( \Delta \Sigma^j_h \).

**Proof.** Notice that due to Lemma 11 each data node in \( \Delta \Sigma^{k-1}_i \), for \( i \in [1, h - 1] \), can be deleted from its domain. In this way, each \( \Delta \Sigma^k_h \) is empty and thus due to Lemma 5, \( k \) can be deleted from \( \Delta \Sigma^j_h \). □

It should be emphasized that Condition POP is a necessary and sufficient condition, i.e. it states the only possible condition such that a data node due to its post-order value can be deleted. The proof is Theorem 4 where we show that in the computation of the delta answers there is no condition on post-order values to be checked.

The problem of the generic twig inclusion evaluation is more involved than the path matching problem. In these cases, both the two kind of postorder relationship are in principle allowed between any two data nodes in a set qualifying the pattern matching and only a partial order is required. The following Lemma is the counterpart of Lemma 11 as it shows the postorder conditions under which the nodes preceding \( d_k \) in the sequential scanning...
Lemma 13 (Condition POT2) Let $s_i \in \Delta \Sigma_i^k$ and $\text{post}(d_{s_i}) < \text{post}(d_k)$. If $i \in [1,n]$ exists such that $\text{post}(q_i) < \text{post}(q_i)$ and $\Delta \Sigma_i^k = \emptyset$ or for each $s \in \Delta \Sigma_i^k$ such that $s > s_i$, $\text{post}(d_s) > \text{post}(d_{s_i})$, then $s_i \notin S$, for each $S \in \Delta(U)\text{ans}_\mathcal{Q}(D)$ for each $j \in [k,m]$. \\
Proof. Let us suppose that $j \in [k,m]$ exists such that $S = (s_1, \ldots, s_n) \in \Delta(U)\text{ans}_\mathcal{Q}(D)$ and $s_i \in S$, and that $i \in [1,n]$ exists such that $\text{post}(q_i) < \text{post}(q_i)$. Notice that, being $\text{post}(q_i) < \text{post}(q_i)$, the node matching with $q_i$, i.e. that with preorder value equal to $s_i$, both in the case of ordered and unordered matching must satisfy the following property: $\text{post}(d_{s_i}) > \text{post}(d_{s_i})$. On the other hand, wherever $\Delta \Sigma_i^k = \emptyset$ or for each $s \in \Delta \Sigma_i^k$, $\text{post}(d_s) > \text{post}(d_{s_i})$, the condition $\text{post}(d_{s_i}) > \text{post}(d_{s_i})$ can be satisfied if $s_i > k$. But being $\text{post}(d_{s_i}) < \text{post}(d_k)$, due to Lemma 10 $\text{post}(d_{s_i}) < \text{post}(d_s)$ for each $s > k$ and thus also for $s = s_i$. Thus the above condition can not be satisfied. Finally, whenever Condition POT2 involves the root domain, i.e. $\Delta \Sigma_1^k$, no check on the other delta domains is required and all the nodes $s_i \in \Delta \Sigma_1^k$ such that $\text{post}(d_{s_i}) < \text{post}(d_k)$ can be deleted from $\Delta \Sigma_1^k$.

Lemma 14 (Condition POT3) Let $s_i \in \Delta \Sigma_1^k$ and $\text{post}(d_{s_i}) < \text{post}(d_k)$ then $s_i \notin S$, for each $S \in \Delta(U)\text{ans}_\mathcal{Q}(D)$ for each $j \in [k,m]$. \\
Proof. First notice that for each $q_i$ for $i \in [2,n]$, $\text{post}(q_i) > \text{post}(q_i)$ since $q_i$ is the root of the twig pattern, thus for each $j \in [k,m]$ for each solution $S = (s_1, s_2, \ldots, s_n) \in \Delta(U)\text{ans}_\mathcal{Q}(D)$ involving $s_1$ it should be $\text{post}(d_{s_1}) > \text{post}(d_{s_i})$. On the other hand being $\text{post}(d_{s_i}) < \text{post}(d_k)$, from Lemma 10 it follows that $\text{post}(d_{s_i}) < \text{post}(d_s)$ for each $s > k$ and $S \in \Delta(U)\text{ans}_\mathcal{Q}(D)$ iff at least one $s_i$ is greater than $k$.

Theorem 3 (Completeness) For the twig case, beyond the conditions expressed in Lemmas 13 and 14, there is no other condition ensuring that at each step $k$, any data node due to its post-order value does not belong to the solutions which will be generated in the following steps.

Proof. As we have shown, for a data node $s_i \in \Delta \Sigma_1^k$ whenever $\text{post}(d_{s_i}) > \text{post}(d_k)$ there is no way to predict the relationship between the post-order of the data node $s_i$ and that of the nodes after $k$. On the other hand, whenever $\text{post}(d_{s_i}) < \text{post}(d_k)$ we will show that the following two facts together constitute an absurd:
1. $s_i$ does not belong to $\Delta ans_j^k(D)$, $j \in [k, m]$, because for each $(s_{i+1}, \ldots, s_n) \in \Delta \Sigma_{i+1}^j \times \ldots \times \Delta \Sigma_n^j$, $i'$ exists such that it is required that $post(d_{s_{i'}}) < post(d_{s_i})$ but $post(d_{s_{i'}}) > post(d_{s_i})$.

2. $i \neq 1$ and for each $\bar{i} \in [1, n]$ such that $post(q_{\bar{i}}) < post(q_i)$ and $\Delta \Sigma^k_{\bar{i}} \neq \emptyset$ and exists $s \in \Delta \Sigma^k_{\bar{i}}$ such that $s > k$ and $post(d_s) > post(d_{s_i})$ (condition of Lemma 13).

Indeed, wherever the fact above is true, $(s_{i+1}, \ldots, s_n) \in \Delta \Sigma_{i+1}^j \times \ldots \times \Delta \Sigma_n^j$ exists such that for each $i' > i$ such that $post(q_{i'}) < post(q_i)$, $s_{i'} > s_i$ and $post(d_{s_{i'}}) < post(d_{s_i})$. Indeed, the data nodes of such a partial solution can be the ones specified in the fact above. □

3.3 On the computation of new answers

In this subsection, we want to detect at which step of the sequential scanning new matches can be decided. For this problem, we must again consider two cases: the ordered and the unordered. For the ordered case, we exploit the total order among the pre-order values of the data nodes in a match. A direct consequence of this property is given by the following fact, stating that the set of matches which can be computed at step $k$ is empty unless $l_k(D)$ matches with the “last” query node $l_n(Q)$.

**Lemma 15** If $h \neq n$ then $\Delta ans_n^k(Q) = \emptyset$.

For the unordered case, where only a partial order is required, new matches can only be decided when the data node $k$ matches with a query node which is a leaf.

**Lemma 16** If $i > h$ exists such that $post(q_h) > post(q_i)$ then $\Delta ans_n^k(Q) = \emptyset$.

**Proof.** Let us suppose that $S = (s_1, \ldots, s_h, \ldots, s_n)$ exists such that $s_h = k$. Moreover by hypothesis $i > h$ exists such that $post(q_h) > post(q_i)$. Thus $S$ is a solution if $s_i > k$ that is $s_i$ must be accessed after $k$ in the sequential scanning. For this reason $S$ can not belong to $\Delta ans_n^k(Q)$.

3.4 Characterization of the delta answers

In this subsection, we characterize the delta answers generated at each sequential step, respecting our three kinds of pattern matching strategies.

The following Theorem represents a considerable result for the path matching. It shows how the set of delta answers can be computed at each
step of the sequential scanning. It also shows that Lemma 11 together with Theorem 1 delete all the data nodes which are no longer necessary.

**Theorem 4** If Lemmas 4, 5, 6, and 11 have been applied at each step of the sequential scanning then the set of answers $\Delta \text{ans}^k_P(D)$ which can be generated at step $k$ for the path $P$ is such subset of the cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ defined as $\{(s_1, \ldots, s_n) | s_i \in \Delta \Sigma^k_{i+1} \text{ for each } i \in [1, n-1]\}$

**Proof.** The set of answers $\Delta \text{ans}^k_P(D)$ is a subset of the cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ as, by applying Lemmas 4, 5, 6, and 11, we never delete useful data nodes.

Given that premise, we have to show that if $(s_1, \ldots, s_n) \in \Delta \text{ans}^k_P(D)$ then $s_i \in \Delta \Sigma^k_{i+1}$ for each $i \in [1, n-1]$ and vice versa. If $(s_1, \ldots, s_n) \in \Delta \text{ans}^k_P(D)$ then $s_1 < \ldots < s_n$ and thus $s_i$ must be processed before $s_{i+1}$ in the sequential scanning that is $s_i \in \Delta \Sigma^k_{i+1}$. The other way around, we have to show that if $s_i \in \Delta \Sigma^k_{i+1}$ for each $i \in [1, n-1]$ then $(s_1, \ldots, s_n) \in \Delta \text{ans}^k_P(D)$. We first show that $(s_1, \ldots, s_n)$ is a solution. Notice that $(s_1, \ldots, s_n)$ can be a solution as $s_1 < s_2 < \ldots < s_n$ because $s_i \in \Delta \Sigma^k_{i+1}$ and, for each $i \in [1, n-1]$, $\text{post}(d_{s_i}) > \text{post}(d_{s_{i+1}})$ since, due to Lemma 11, we have that $\text{post}(d_{s_i}) > \text{post}(d_{s_j})$ for each $j$ such that $s_i < j \leq k$ being $s_i \in \Delta \Sigma^k_i$. Finally, we show that $(s_1, \ldots, s_n) \notin \Delta \text{ans}^{k-1}_P(D)$. It is true as $s_n \in \Delta \Sigma^k_n$ if, due to Lemma 4, $k = s_n$ and thus, it straightforwardly follow that $s_n \in \Delta \Sigma^k_n$ which is one of the domains of $\Delta \text{ans}^{k-1}_P(D)$.

For the ordered case, post-order values must be checked. On the other hand, the completeness of the conditions shown in the previous section ensures that we cover all possible node deletions from the pre- and post-order points of view. In particular, in this case, due to Lemma 5, the “last” delta domain $\Delta \Sigma^k_n$ is always empty unless the current data node $k$ matches with the $n$-th query node and, in this case, $\Delta \Sigma^k_n$ only contains $k$. Thus, whenever $\Delta \text{ans}^k_Q(D)$ is not empty, it only contains new matches.

**Theorem 5** If Lemmas 4, 5, 6, 9, 13, and 14 have been applied at each step of the sequential scanning then the set of answers $\Delta \text{ans}^k_Q(D)$ which can be generated at step $k$ for the twig $Q$ is such subset of the Cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ where for each $(s_1, \ldots, s_n)$: (1) $s_i \in \Delta \Sigma^k_{i+1}$ for each $i < n$ (2) condition on the post-order values expressed in Lemma 1 are satisfied.

**Proof.** The set of index tuples specified in the theorem is a subset of the cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ as, by applying Lemmas 4, 5, 6, 9, 13, and 14, we never delete useful data nodes. Moreover any index tuple $(s_1, \ldots, s_n)$ which satisfies conditions 1 and 2 is a solution because, as in the
path case, condition 1 implies that $s_1 < s_2 < \ldots < s_n$ whereas condition 2 explicitly requires that the relationships between post-orders are satisfied. Finally the fact that $(s_1, \ldots, s_n) \not\in \text{ans}^{k-1}_Q(D)$ is the same as in the path case.

In the unordered case, instead, as the delta domains of the query leaves are not always empty, we must avoid producing redundant results.

**Theorem 6** If Lemmas 7, 8, 9, 13, and 14 have been applied at each step of the sequential scanning, then the set of answers $\Delta \text{ans}^k_Q(D)$, which can be generated for the twig $Q$ at step $k$ whenever $h$ is a leaf, is such subset of the Cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ so that for each $(s_1, \ldots, s_h, \ldots, s_n)$: (1) $s_h = k$, (2) for all pairs of entries $i$ and $j$, $i, j = 1, 2, \ldots n - 1$ and $i + j \leq n$, if $\text{post}_{i+j}(Q) < \text{post}_i(Q)$ then $\text{post}_{s_{i+j}}(D) < \text{post}_j(D)$ and $s_i \in \Delta \Sigma^k_{s_{i+j}}$.

**Proof.** The set of index tuples specified in the theorem is a subset of the cartesian product $\Delta \Sigma^k_1 \times \ldots \times \Delta \Sigma^k_n$ as, by applying Lemmas 7, 8, 9, 13, and 14, we never delete useful data nodes. Moreover any index tuple $(s_1, \ldots, s_n)$ which satisfies conditions 1 and 2 is a solution and the proof is similar to the above ones.

4 Pattern matching algorithms

In the previous section we have introduced a theoretical framework consisting of a set of pre/post-order conditions for a node to be deleted. The next challenge is to conceive sequential pattern matching algorithms that exploit the theoretical framework to manage the domains efficiently. The set of conditions is complete thus ensuring that the domains are maintained as compact as possible from a numbering scheme point of view. At the same time, to efficiently put the theoretical framework in practice, it also means to find implementation solutions consuming little time. It should be noticed that a smart management of the domains in the sequential scanning does not prevent the adoption of other improvements like filters or the use of indexes.

In this section we show how the conditions presented so far can be used in pattern matching algorithms to manage the domains associated with the query nodes. In particular, we propose three classes of algorithms which perform twig pattern matching on tree signatures by sequentially scanning a limited portion of the data signature. Such algorithms associate one domain to each query node and rely on two principles: generate the qualifying index set as soon as possible and delete from the domains those data nodes which
are no longer needed for the generation of the subsequent answers. With reference to the previous section, during the scanning process the algorithms generate the “delta” answer sets $\Delta(U)\text{ans}_Q^k$, that is the set of answers which can be computed at step $k$.

4.1 Notation and common basis

Let $\text{sig}(D) = \langle d_1, \text{post}(d_1); d_2, \text{post}(d_2); \ldots; d_m, \text{post}(d_m) \rangle$ denote the signature of the data tree and $\text{sig}(Q) = \langle q_1, \text{post}(q_1); q_2, \text{post}(q_2); \ldots; q_n, \text{post}(q_n) \rangle$ denote the signature of a query twig pattern. To distinguish a path from a general tree, we use the capital letter $P$ in place of $Q$. For each query node $q_i$, we assume that the $\text{post}(q_i)$ operator accesses its post-order value and $D_i$ represents its domain. Together with the domain, the maximum and the minimum postorder values of the data nodes stored in $D_i$, accessible by means of the $\text{minPost}$ and $\text{maxPost}$ operators, respectively, are associated to each query node. $\text{first}(q_i)$ is the preorder value of the first occurrence of the name of $q_i$ in $D$, i.e. the minimum preorder, and $\text{last}(q_i)$ is the last one, i.e. the maximum preorder value of node with name $q_i$. Recall that a preorder of a node is also the node’s position (index) in the signature. Notice that both these values can be computed while constructing the data tree signature.

Nodes are scanned in sorted order of their pre-order values, in a range returned by the $\text{getRange()}$ function. In the future, this function will exploit advanced filtering techniques in order to retrieve the smallest possible range with respect to the query. For now, thanks to Condition PRO2, the sequential scanning can start from $\text{first}(q_1)$. In the ordered case, it can end at $\text{last}(q_n)$ due to Lemma 15 whereas, in the unordered case, Lemma 16 suggests to set the end as the maximum value among $\text{last}(q_l)$ for each leaf $l$ in the query.
Insertions in the domains are always performed on the top by means of the `push()` operator. Thus the data nodes from the bottom up to the top of each domain are in pre-order increasing order. Moreover, each data node $d_k$ in the pertinence domain $D_h$ consists of a pair: $(\text{post}(d_k), \text{pointer to a node in } D_{\text{prev}(h)})$ where $\text{prev}(h)$ is $h-1$ in the ordered case whereas it is the parent of $h$ in $Q$ in the unordered case. When the data node $d_k$ is inserted into $D_h$, its pointer indicates the pair which is at the top of $D_{\text{prev}(h)}$. For illustration see Figure 5, where a node $d_k'$ preceding $d_k$ is represented with its pointers. In this way, the set of data nodes in $D_{\text{prev}(h')}$ from the bottom up to the data node pointed by $k'$ implements $\Delta \Sigma_{\text{prev}(h')}$ and the whole content of $D_{\text{prev}(h')}$ at step $k$ implements $\Delta \Sigma_{k'}^{\text{prev}(h')}$. By recursively following such links from $D_{\text{prev}(h')}$ to $D_1$, we can derive $\Delta \Sigma_{\text{prev}(\text{prev}(h'))}, \ldots, \Delta \Sigma_1^{k'}$. As a final note, we allow the access to a particular position (from the bottom to the top) in each domain by means of the dot notation. For instance $D_{3.2}$ means the second entry from the bottom of $D_3$.

### 4.2 Path Matching

<table>
<thead>
<tr>
<th>DATA STACK MANAGEMENT</th>
<th>SOLUTION CONSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty(stack)</code></td>
<td>Empties stack</td>
</tr>
<tr>
<td><code>isEmpty(stack)</code></td>
<td>Returns true if <code>stack</code> is empty, false otherwise</td>
</tr>
<tr>
<td><code>pointer(elem)</code></td>
<td>Returns the pointer of the given element</td>
</tr>
<tr>
<td><code>pointerToTop(stack)</code></td>
<td>Returns a pointer to the top of the given stack</td>
</tr>
<tr>
<td><code>pop(stack)</code></td>
<td>Pops (and returns) an element from <code>stack</code></td>
</tr>
<tr>
<td><code>push(stack, elem)</code></td>
<td>Pushes given <code>elem</code> in <code>stack</code></td>
</tr>
<tr>
<td><code>showSolutions(...)</code></td>
<td>Recursively builds the path solutions</td>
</tr>
</tbody>
</table>

Table 1: Path matching functions

In this subsection we propose the algorithm for the matching of a path pattern in a data tree. In this case, domains can be treated as stacks, that is deletions are implemented following a LIFO policy. Table 1 provides a summary of the path matching auxiliary functions. In particular, the top of the table presents all the functions which are needed in order to manage the stacks; in addition to the already introduced `push()` function, notice the complementary `pop()` one, which performs the deletions. Further, `isEmpty()` checks whether a given stack is empty, whereas `empty()` empties the stack. In the lower part of the table the functions needed for the solutions
**Input:** path $P$ having signature $\text{sig}(P)$; rew($Q$)  
**Output:** $\text{ans}_P(D)$

```
algorithm PathMatch(P)
(0) getRange(start, end);
(1) for each $d_k$ where $k$ ranges from start to end
(2) for each $h$ such that $q_h = d_k$ in descending order
(3) for each $D_i$ where $i$ ranges from 1 to $n$
(4) while(!isEmpty($D_i$) ∨ post(top($D_i$))<post($d_k$))
(5) pop($D_i$);
(6) if(isEmpty($D_i$))
(7) for each $D_{i'}$ where $i'$ ranges from $i+1$ to $n$
(8) empty($D_{i'}$);
(9) if(!isEmpty($D_{h-1}$))
(10) push($D_h$, (post($d_k$), pointerToTop($D_{h-1}$)));
(11) if($h = n$)
(12) showSolutions($h, 1$);
(13) pop($D_h$);
(14) for each $D_i$ where $i$ ranges from 1 to $n$
(15) if(isEmpty($D_i$) ∧ last($q_i$)<$k$)
(16) exit;

procedure showSolutions($h, p$)
(1) index[$h$] ← $p$;
(2) if($h=1$) output($D_1$.index[1], ...,$D_n$.index[n])
(3) else
(4) for $i = 1$ to pointer($D_h$.index[$h$])
(5) showSolutions($h-1, i$);
```

**Figure 6: The path matching algorithm**

construction are shown, in our case the only showSolutions(). We assume that the query pattern has unique node names. The algorithm for path matching, which is depicted in Fig. 6, can be easily extended to the case where multiples query nodes have the same names and we will show it in the following. Finally, for path matching we do not need the maximum and the minimum postorder values associated to each stack.

The key idea of the Algorithm is to scan the portion of the signature from $\text{start}$ to $\text{end}$, which in this case will be from $\text{first}(q_1)$ to $\text{last}(q_n)$.
For each data node, from Line 2 to Line 8, it deletes those data nodes in the stacks which are no longer useful to generate the delta answers. Then it adds the $k$-th data node in the proper stack (Line 10) and, if the data node matches with the leaf of the query path, it shows the answers which can be generated (Line 12). Notice that the $k$-th data node points to that data node which matches $q_{h-1}$ and which has the biggest preorder value smaller than $k$. Such pointers will be used in the `showSolutions()` function. At line 13, due to the PRO1 condition, the algorithm delete $k$ if it is the last node.

Instead of checking all nodes as specified in Condition POP, we stop looking at the nodes in $D_i$ whenever $\text{post(top}(D_i)) > \text{post}(d_k)$ (line 4). It fully implements Condition POP because, as we will prove in the following, if $\text{post(top}(D_i)) > \text{post}(d_k)$ then $\text{post}(s_i) > \text{post}(d_k)$ for each $s_i \in D_i$ and thus Condition POP can no longer be applied. Moreover, Condition PRO2 is implemented in Lines 6-9. Observe that, instead of checking the intersection between the state of the domains at different steps as required by Condition PRO2, we only check whether $\text{isEmpty}(D_i)$ is true at any step $j'$ with $k \leq j' \leq j$. Indeed, we will show that, in order to delete the nodes belonging to a domain $D_i$ at step $k$, it is first necessary to delete the nodes belonging to $D_i$ at a step preceding the $k$-th one.

Before demonstrating the correctness of the algorithm, we present three properties of the data nodes stored in each stack $D_i$.

**Lemma 17** If $\text{post(top}(D_i)) > \text{post}(d_k)$ then $\text{post}(d_{k'}) > \text{post}(d_k)$ for each $k' \in D_i$.

**Proof.** Let $k' \in D_i$ and let $\text{top}(D_i) = k''$ then $k' < k''$ as $k''$ is at the top of $D_i$ and the data signature is scanned in a sequential way. Then there are two alternatives: either $\text{post}(d_{k'}) > \text{post}(d_{k''})$ or $\text{post}(d_{k'}) < \text{post}(d_{k''})$. In the first case, it straightforward follows that $\text{post}(d_{k'}) > \text{post}(d_k)$ as due to the premise $\text{post}(d_{k''}) > \text{post}(d_k)$ whereas the second case is impossible as when $k''$ was added to $D_i$, the algorithm should have deleted $k'$ from $D_i$ (see Line 5). \hfill $\Box$

**Lemma 18** For each $i \in [1, n]$ $\Delta \Sigma^i_k \cap \Delta \Sigma^i_j = \emptyset$ iff the condition is\texttt{isEmpty}(D$_i$) is true at any step $j'$ with $k \leq j' \leq j$.

**Proof.** Let us suppose that $\Delta \Sigma^i_{j'} = \emptyset$ with $k \leq j' \leq j$, then being $k \leq j'$ and $j' \leq j$ it easily follows that $\Delta \Sigma^i_k \cap \Delta \Sigma^i_j = \emptyset$.

As far as the opposite direction is involved, we show that if for each step $j'$ with $k \leq j' \leq j$ we have that $\Delta \Sigma^i_{j'} \neq \emptyset$ then $\Delta \Sigma^i_k \cap \Delta \Sigma^i_j \neq \emptyset$. The proof
is by induction. When \( j = k \) and \( \Delta \Sigma_i^k \neq \emptyset \), then \( \Delta \Sigma_i^k \cap \Delta \Sigma_i^k = \Delta \Sigma_i^k \neq \emptyset \). Let us suppose that the statement is true for \( j = r \), we show it for \( j = r + 1 \). As if for each step \( j' \) with \( k \leq j' \leq r \) \( \Delta \Sigma_i^{j'} \neq \emptyset \) then \( \Delta \Sigma_i^k \cap \Delta \Sigma_i^k \neq \emptyset \), let us suppose that \( \Delta \Sigma_i^r \cap \Delta \Sigma_i^k \neq \emptyset \). We have to show that if for each step \( j' \) with \( k \leq j' \leq (r + 1) \) \( \Delta \Sigma_i^{j'} \neq \emptyset \) then \( \Delta \Sigma_i^{r+1} \cap \Delta \Sigma_i^k \neq \emptyset \). Notice that \( \Delta \Sigma_i^{r+1} \cap \Delta \Sigma_i^k = \emptyset \) iff at step \( r + 1 \) we delete the index set \((i_1, \ldots, i_n)\) from \( \Delta \Sigma_i^{r+1} \). But, as such domains are stacks and \( k \leq (r + 1) \) thus \((i_1, \ldots, i_n)\) are at the bottom of the stack and the deletion of \((i_1, \ldots, i_n)\) implies the deletion of all the data node in \( \Delta \Sigma_i^{r+1} \), but it is impossible because the \( \Delta \Sigma_i^{r+1} \neq \emptyset \). □

Lemma 19 At each step \( j \) and for each query index \( i \), the stack \( D_i \) is a subset of \( \Sigma_i^j \) containing only the data entries that cannot be deleted from \( \Sigma_i^j \), i.e. it has the same content as \( \Delta \Sigma_i^j \) when Lemmas 4, 5, 6, and 11 have been applied.

Proof. First, notice that \( D_i \subseteq \Sigma_i^j \) as for each data node \( d_j \), at Line 10 the algorithm adds \( d_j \) to the right stack. Moreover the algorithm deletes some indexes from \( D_i \) by means of the pop and empty operators. If \( k \) can not belong to \( \Delta \Sigma_i^j \) due to its preorder value, it can be due to one of three possible alternatives shown in Theorem 1. The algorithm detects all these conditions and delete \( k \) from \( D_i \). At Lines 6-8, assuming that \( D_i = \Delta \Sigma_i^j \), due to Lemma 6 it deletes all the nodes in \( D_{i+1} \cup \ldots \cup D_n \). Indeed, in Lemma 18 it has been shown that \( \Delta \Sigma_i^j \) must be empty in order that \( \Delta \Sigma_i^k \cap \Delta \Sigma_i^{j'} = \emptyset \), for \( j' \geq j \). Thus, at the \( j \)-th step we delete all the “unnecessary” data nodes. At Lines 9-10, it applies Lemma 5. In particular, as whenever a stack is empty we empty all the stacks at “its right”, then it is sufficient to check stack \( D_{n-1} \). Finally, at Line 13, it applies Lemma 4. Moreover, the algorithm deletes \( k \) from \( D_i \) due to its postorder value at Lines 4-5 where it applies Lemma 11. Notice that, due to Lemma 17, the algorithm correctly avoid to go on with \( D_i \) whenever \( \text{post(top}(D_i)) > \text{post}(d_j) \) as, in this case, no other data node can be deleted from that stack due to its postorder value. It follows that the algorithm never deletes from the stacks data nodes which could belong to a delta answer set \( \Delta \text{ans}_{P'}(D) \) for \( j' \in [j, m] \). Thus, \( k \in D_i \) at step \( j \) iff \( k \in \Delta \Sigma_i^j \). □

Starting from the data node in the leaf stack \( D_n \) (function call at Line 12 of the main algorithm), function \texttt{showSolutions()} uses the pointer associated with each data node \( d_k \) to “jump” from one stack \( D_i \) to the previous one \( D_{i-1} \) (up to \( D_1 \)) and recursively combines \( d_k \) with each node from the bottom of \( D_{i-1} \) up to the node pointed by \( d_k \). The correctness of the algorithm
follows from the properties shown so far.

**Theorem 7** For each data node \( d_j \), \( S = (s_1, \ldots, s_n) \) \( \in \Delta\text{ans}^j_p(D) \) iff the algorithm, by calling the function `showSolutions()`, generates the solution \( S \).

**Proof.** Due to Theorem 4, \( S = (s_1, \ldots, s_n) \) belongs to \( \Delta\text{ans}^j_p(D) \) iff for each \( i \in [1,n] \), \( s_i \in \Delta \Sigma_{i+1}^{s_i} \) and \( s_i \in \Delta \Sigma_i^k \). Lemma 19 states that \( s_i \in \Delta \Sigma_i^k \) iff \( s_i \in D_i \) at step \( k \). Moreover, the “chain” of pointers followed by the function `showSolutions()` allows us to state that such a function only generates those solutions \( S = (s_1, \ldots, s_n) \) such that \( s_i \in \Delta \Sigma_{i+1}^{s_i} \). Indeed, whenever the algorithm adds a new data node to any stack \( D_h \), it sets the pointer to the top of the “preceding” stack \( D_{h-1} \) (Line 10). Thus, as the algorithm sequentially scans the data signature, for each data node \( s_{i+1} \) in \( D_{i+1} \), the nodes from the bottom of \( D_i \) up to the node pointed by \( s_{i+1} \) are all those nodes matching \( q_i \) and whose preorder value is less than \( s_{i+1} \), i.e. all those in \( \Sigma_{i+1}^{s_i} \).

Finally, let us consider the correctness of the scanned range, which is between the first occurrence of \( q_1 \) and the last occurrence of \( q_n \) in \( \text{sig}(D) \). As \( D_1 \) is empty before \( \text{first}(q_1) \) has been accessed, we can avoid to access all the data nodes before \( \text{first}(q_1) \) which should be inserted in \( D_2, \ldots, D_n \) but which will never been used due to Lemma 5. On the other hand, from Lemma 4, it follows that \( \Delta\text{ans}^k_q(D) = \emptyset \) for \( k \in [\text{last}(q_n) + 1, m] \). Therefore \( \text{ans}_Q(D) = \bigcup_{k=\text{last}(q_n)}^{\text{last}(q_n)} \Delta\text{ans}^k_q \). Moreover, the algorithm exits whenever any stack \( D_i \) is empty and no data node matching with \( q_i \) will be accessed, i.e. \( \text{last}(q_i) < k \) (Lines 14-16). It means that \( \Delta \Sigma_i^{k'} = \emptyset \) for each \( k' \in [k,m] \) and thus that \( \Delta\text{ans}^{k'}_q(D) = \emptyset \).

### 4.3 Ordered Twig Pattern Matching

Differently from the path case, the domains of the twig matching algorithms cannot be stacks because they are not ordered on post-order values, thus deletions can be applied at any position of the domains. Therefore, in this case we will implement the query domains as lists. Table 2 shows a summary of the functions employed in the ordered twig matching algorithms; note that we omit the functions which we already discussed in the path case (functions such as `empty()` or `push()`, which are the same but applied to lists instead of stacks), `deleteLast()` is the equivalent of the `pop()` used for stacks. Besides the functions managing lists, the rest of the table shows the functions which interact with the main algorithm in order to produce the matching results: The boolean functions `isCleanable()` and `isNeededOrd()`, checking
### DATA LIST MANAGEMENT

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decreasePointer(elem)</code></td>
<td>Decreases by 1 the pointer in given <code>elem</code></td>
</tr>
<tr>
<td><code>delete(list, elem)</code></td>
<td>Deletes given <code>elem</code> from <code>list</code></td>
</tr>
<tr>
<td><code>deleteLast(list)</code></td>
<td>Deletes last <code>elem</code> from <code>list</code></td>
</tr>
<tr>
<td><code>index(list, elem)</code></td>
<td>Returns the position of <code>elem</code> in <code>list</code></td>
</tr>
<tr>
<td><code>noEmptyLists()</code></td>
<td>Returns false if there is at least an empty list</td>
</tr>
<tr>
<td><code>pointerToLast(list)</code></td>
<td>Returns a pointer to the top of the given <code>list</code></td>
</tr>
</tbody>
</table>

### MATCHING

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isCleanable(pre, elem)</code></td>
<td>Returns true if current <code>elem</code> can be deleted from the <code>pre</code>-th list, false otherwise</td>
</tr>
<tr>
<td><code>isNeededOrd(pre, elem)</code></td>
<td>Checks if insertion of given <code>elem</code> is needed in the <code>pre</code>-th list (ordered version)</td>
</tr>
<tr>
<td><code>updateRightLists(pre, pos)</code></td>
<td>Updates the pointers in the <code>(pre + 1)</code>-th list, possibly propagating the update, following the deletion of the <code>pos</code>-th element from the <code>pre</code>-th list</td>
</tr>
</tbody>
</table>

### TWIG QUERY NAVIGATION

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>descendants(pre)</code></td>
<td>Return the descendants of the given twig node</td>
</tr>
</tbody>
</table>

### SOLUTION CONSTRUCTION

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findSolsOrd(...)</code></td>
<td>Recursively builds the ordered twig solutions</td>
</tr>
<tr>
<td><code>checkPostDir(node, precNode)</code></td>
<td>Returns true if given nodes have the correct post-order direction w.r.t. the query</td>
</tr>
<tr>
<td><code>twigBlock(pre)</code></td>
<td>Returns the block information for the <code>pre</code>-th query node (1 for block opening, -n for n blocks closing, 0 otherwise)</td>
</tr>
<tr>
<td><code>updateStackMax(stack, post)</code></td>
<td>Updates top element of <code>stack</code> if less than <code>post</code></td>
</tr>
</tbody>
</table>

Table 2: Ordered twig matching functions

whether a node can be deleted and whether a node insertion can be avoided, respectively, and `updateRightLists()`, updating domains after a deletion. Each of these matching functions will be shown in detail, together with the main algorithm. Further, as we will show, twig query navigation functions and more advanced solution construction functions are now needed, in this case `descendants()` and the other ones shown in the lower part of the table and that will be discussed later while explaining the solution construction algorithm. Finally, in the twig matching algorithms we also need the maximum and the minimum postorder values associated to each domain, and thus we will also exploit the `minPost` and `maxPost` operators discussed in Section 4.1.

The ordered twig matching algorithm is shown in Figure 7. The scanned
Input: query $Q$ having signature $sig(Q)$; $rew(Q)$
Output: $ans_Q(D)$

algorithm OrderedTwigMatch($Q$)
(0) getRange($start$, $end$);
(1) for each $d_k$ where $k$ ranges from $start$ to $end$
(2) for each $h$ such that $q_h = d_k$ in descending order
(3) for each $D_i$ where $i$ ranges from 1 to $n$
(4) for each $d_i$ in $D_i$ in ascending order
(5) if($post(d_i) < post(d_k) \land isCleanable(i,d_i)$)
(6) $pos \leftarrow$ index($D_i,d_i$);
(7) delete($D_i,d_i$);
(8) if($i \neq n$)
(9) updateRightLists($i,pos$);
(10) if($\neg isEmpty(D_{h-1}) \land isNeededOrd(h,d_k)$)
(11) push($D_h,(post(d_k),pointerToLast(D_{h-1}))$);
(12) if($h = n$)
(13) findSolsOrd($h,1$);
(14) deleteLast($D_h$);
(15) for each $D_i$ where $i$ ranges from 1 to $n$
(16) if($isEmpty(D_i) \land last(q_i)<k$)
(17) exit;

Figure 7: The ordered twig matching algorithm

range is the same as the one we previously discussed for paths. Further, as for the path algorithm, also in this case we implement the required conditions in the most effective order. First, we try to delete nodes by means of the post-order conditions, in particular POT2 and POT3, (Lines 3-9) and, if a deletion is performed, we update the pointers in the right lists (Line 9). Then, we work on the current node (Lines 10-14), checking if an insertion is needed (condition PRO2 and POT1, Line 10) and verifying if new solutions can be generated (Lines 12-13). As in path matching, condition PRO1 is used to delete a node from the last stack (Line 14) and the algorithm exits whenever any stack $D_i$ is empty and no data node matching with $q_i$ will be accessed (Lines 15-17).

Let us first analyze the deletion part of the algorithm (Lines 3-9). Here, the boolean function $isCleanable()$ (see Figure 8 for the complete code)
function isCleanable(i, d_i)
(1) if(i = 1)
(2) return true;
(3) for each i in descendants(i)
(4) if(isEmpty(D_i) ∨ minPost(D_i) > post(d_i))
(5) return true;
(6) return false;

function isNeededOrd(i, d_i)
(1) if(i = 1)
(2) return true;
(3) if(isEmpty(D_parent(i)) ∨ maxPost(D_parent(i)) < post(d_i))
(4) return false;
(5) if(isEmpty(D_i−1) ∨ minPost(D_i−1) > post(d_i))
(6) return false;
(7) return true;

procedure updateRightLists(i, pos)
(1) for each d_i+1 in D_i+1 in descending order
(2) if(pointer(d_i+1)=1)
(3) for each d_i+1 from d_i+1 in descending order
(4) rPos ← index(D_i+1, d_i+1);
(5) delete(D_i+1, d_i+1);
(6) if(i + 1 ≠ n)
(7) updateRightLists(i + 1, rPos);
(8) return;
(9) else if(pointer(d_i+1)<pos)
(10) return;
(11) else
(12) decreasePointer(d_i+1);

Figure 8: The ordered twig matching auxiliary functions

checks whether d_i can be deleted. In particular, if i is the index of the twig root, it simply returns true (Condition POT3), otherwise it checks the conditions expressed in Condition POT2. In this case, links connecting each domain D_i with the domains of the descendants i in of the i-th
twig node and the minPost operator are exploited to speed up the pro-
cess. In particular, instead of checking the post-order value of each data
node in the domains, we check if \( \text{minPost}(D_i) > \text{post}(d_i) \) for each domain
\( D_i \) (Line 4 of the isCleanable() code). Whenever a node \( d_i \) is deleted,
the updateRightLists() function, shown in detail in Figure 8, updates
the pointer of all the nodes pointing to \( d_i \) in order to make it point to the
node below \( d_i \) (Line 12 of the function). Such an update is performed in
a descending order and stops when a node pointing to a node below \( d_i \) is
accessed (Line 10).

As to the current node insertion (Lines 10-11 of the main algorithm),
condition PRO2 is exploited, making it sufficient to only check \( D_{h-1} \). This is
because, whenever Condition PRO2 is applied, if a domain \( D_h \) is empty then
all the domains “following” \( D_h \) are emptied. Such emptying are performed
in the updateRightLists() code from Lines 3 to 8, and are the application
of PRO2 to a node inserted at step \( k' \) (\( d_{i+1} \) in the algorithm) preceding the
current \( k \)-th data node and thus already belonging to a domain \( D_{h'} \), where
\( h' > i \). In this case, \( d_{i+1} \) is deleted when, due to the deletions applied in the
main algorithm, its pointer becomes dangling. We recall that, for instance,
\( \Delta \Sigma_{k'-1}^{h'-1} \) is implemented by that portion of \( D_{h'-1} \) between the bottom and the
data node pointed by \( d_{i+1} \) and \( \Delta \Sigma_{k'-1}^{h'-1} \) is the current state of \( D_{h'-1} \). Thus,
intuitively, if the pointer of \( d_{i+1} \) is dangling it means that \( \Delta \Sigma_{k'-1}^{h'-1} \cap \Delta \Sigma_{k'-1}^{h'-1} =
\emptyset \) as required by Conditions PRO2. The same can be recursively applied to
the other domains \( \Delta \Sigma_{k-2}^{h'-1}, \ldots, \Delta \Sigma_{1}^{h'-1} \).

Before inserting a new node (Lines 10-11 of the main algorithm), we also
call the boolean function isNeededOrd(), which checks the condition shown
in Condition POT1 by using the minPost(\( D \)) and maxPost(\( D \)) values for
each domain \( D \) instead of comparing the current data node with each data
node in \( D \). Notice that, in order to speed up the process, we only perform
the check in the parent domain (Line 3 of isNeededOrd()) and in the first
left sibling (Line 5), due to the transitivity of the relationships. Finally, by
analogy to the path matching algorithm, we check if new solutions can be
generated (Lemma 15) and, in this case, call (Line 13 of the main algorithm)
the recursive function findSolsOrd() implementing Theorem 5.

From the above analysis, it easily follows the Lemma below.

**Lemma 20** At each step \( j \) and for each query index \( i \), the list \( D_i \) is a subset
of \( \Sigma_i^j \) containing only the data entries that cannot be deleted from \( \Sigma_i^j \), i.e.
it has the same content as \( \Delta \Sigma_i^j \) when Lemmas 4, 5, 6, 9, 13, and 14 have
been applied.

The ordered twig matching solution construction, shown in detail in
procedure findSolsOrd(h,p)
(1) index[h] ← p;
(2) if(h = 1) output(D_1.index[1],...,D_n.index[n])
(3) else
(4) if(twigBlock(h)<0)
(5) for i = 0 to twigBlock(h)
(6) push(postStack,post(D_h.index[h]));
(7) else if (twigBlock(h)=0)
(8) updateStackMax(postStack,post(D_h.index[h]));
(9) if(twigBlock(h−1)>0)
(10) curPost ← pop(postStack);
(11) for i = 1 to pointer(D_h.index[h])
(12) okToContinue ← false;
(13) if(checkPostDir(D_h.index[h],D_h−1.i))
(14) if(twigBlock(h−1)<=0)
(15) okToContinue ← true;
(16) else
(17) if(post(D_h−1.i)>curPost)
(18) okToContinue ← true;
(19) updateStackMax(postStack,post(D_h−1.i));
(20) if(okToContinue)
(21) findSolsOrd(h−1,i);

Figure 9: The ordered twig matching solution construction

Figure 9, is inspired by the path one, i.e. it is a function which recursively builds each solution one step at a time, starting from the last domain and outputting the current solution when reaching the first domain (Line 2). However, the function is more complex since the solutions have to be checked while being built; in particular, in the ordered twig matching we would have to do all the post-order checks defined in Lemma 1. Instead of performing all these checks, and in order to maintain the step-by-step backward construction behavior, at each step the algorithm verifies, by means of the checkPostDir() function, if the post order of the current node and the one in the preceding domain have the correct post-order direction (increasing / decreasing) w.r.t. the corresponding twig query nodes (Line 13). Moreover, the algorithm checks that the pre-orders of all the children of a given node are actually smaller than the parent one: This is done by using a
stack structure named \texttt{postStack}, in which the post orders of the children nodes are kept, and a function named \texttt{twigBlock()}, which helps in identifying the structure of the query, i.e. its “blocks” of children. In particular, for a given node, the \texttt{twigBlock()} function returns an integer: 1 for block opening (i.e. the given node is the father of other nodes), -n for n blocks closing (i.e. the given node is a leaf and is the last children for n parents), 0 otherwise (no blocks opening or closing). If a block is closing (Line 4), and this will be the first case since we are constructing the solutions from the last domain, the current post order is saved in the stack (Line 6); such post order is then updated in case of other siblings (Lines 7-8) in order to keep the maximum one of the current block. Then, in case of a parent node (block opening, Line 9) such value is retrieved and, if the post order direction check succeeds (Line 13), Line 17 checks if such value, representing the maximum post order of the children, is less than the post of the current node (Line 17). Finally, if all the check succeed, the algorithm continues by recursively calling itself (Line 21).

\textbf{Theorem 8} For each data node $d_j$, $S = (s_1, \ldots, s_n) \in \Delta\text{ans}^j_Q(D)$ iff the algorithm, by calling the function \texttt{findSolsOrd()}, generates the solution $S$.

\section*{4.4 Unordered Twig Pattern Matching}

In this section we will show the complete unordered twig matching algorithm, commenting on the parts which differ from the ordered one discussed in previous section. Table 2 shows a summary of the functions employed which have not already introduced for the ordered case. In particular, the upper part shows the new functions which interact with the main algorithm in order to produce the matching results: \texttt{isNeededUnord()}, checking whether a node insertion can be avoided, and \texttt{updateDescLists()}, updating domains after a deletion. Such functions are the unordered counterparts of the \texttt{isNeededOrd()} and \texttt{updateRightLists()} discussed in the ordered case. Further, additional twig query navigation functions are needed, which are quite self-explanatory, and, since the solution construction is different from the ordered case, new functions are also needed in this respect (\texttt{findSolsUnord()}, \texttt{preVisit()} and \texttt{extendSols()}). These functions will be discussed later while explaining the solution construction algorithm.

The unordered twig matching algorithm is shown in Figure 10. As in the other two algorithms we analysed, we first try to delete nodes by means of the post-order conditions, in particular POT2 and POT3, (Lines 3-9) and, if a deletion is performed, we update the pointers in the subsequent lists, in
MATCHING

isNeededUnord\(\text{pre, elem}\) Checks if insertion of given elem is needed
in the pre-th list (unordered version)

updateDescLists\(\text{pre, pos}\) Updates the pointers in the descendants of the
pre-th list, possibly propagating the update,
following the deletion of the pos-th element from
the pre-th list

TWIG QUERY NAVIGATION

firstChild\(\text{pre}\) Returns the pre-order of the first child of the
given twig node, -1 if the node is a leaf

firstLeaf() Returns the pre-order of the first leaf in the twig

isLeaf\(\text{pre}\) Returns true if given twig node is a leaf, false
otherwise

parent\(\text{pre}\) Returns the pre-order of the parent of the given
twig node

siblings\(\text{pre}\) Returns the pre-orders of the siblings of the
given twig node

SOLUTION CONSTRUCTION

findSolsUnord(...) Recursively builds the unordered twig solutions

preVisit(...) Used by findSolsUnord to navigate the domains

extendSols(...) Used by findSolsUnord to build the solutions

Table 3: Unordered twig matching functions

this case in the descendant ones (Line 9). Then, we work on the current node
(Lines 10-13), checking if an insertion is needed (condition PRU and POT1,
Line 10) and verifying if new solutions can be generated (Lines 12-13). This
time, condition PRO1 is not available and, thus, we do not delete a node
from the last stack after solution construction as in the other algorithms.
Finally, the algorithm exits whenever any stack \(D_i\) is empty and no data
node matching with \(q_i\) will be accessed (Lines 14-16).

The boolean function isCleanable() is the same of the ordered case and
will not be further discussed. In this case, whenever a node \(d_i\) is deleted,
the updateDescLists() function, shown in detail in Figure 11, updates the
pointer of all the nodes pointing to \(d_i\) in order to make it point to the node
below \(d_i\) (Line 16 of the function). It basically works in the same manner
as updateRightLists() for the ordered case but, instead of updating the
pointers on the following domain, at each call it updates the pointers in all
the domains children of the given one, possibly propagating the update to
the descendants.

As to the current node insertion (Lines 10-11 of the main algorithm),
**Input:** query $Q$ having signature $\text{sig}(Q)$; $\text{rew}(Q)$

**Output:** $\text{ans}_Q(D)$

algorithm UnorderedTwigMatch($Q$)
(0) getRange($\text{start}$, $\text{end}$);
(1) for each $d_k$ where $k$ ranges from $\text{start}$ to $\text{end}$
   (2) for each $h$ such that $q_h = d_k$ in descending order
      (3) for each $D_i$ where $i$ ranges from 1 to $n$
         (4) for each $d_i$ in $D_i$ in ascending order
            (5) if($\text{post}(d_i) < \text{post}(d_k) \land \text{isCleanable}(i,d_i)$)
                pos ← index($D_i$, $d_i$);
            (6) delete($D_i$, $d_i$);
            (7) if($\neg\text{isLeaf}(i)$)
                updateDescLists($i$, pos);
            (8) if($\neg\text{isEmpty}(\text{D}_{\text{parent}}(h)) \land \text{isNeededUnord}(h,d_k)$)
                push($D_h$, $\text{post}(d_k)$,$\text{pointerToLast}(\text{D}_{\text{parent}}(h)))$);
            (9) if($\text{isLeaf}(h) \land \text{noEmptyLists()}$
                findSolsUnord(firstLeaf(), -1, $h$, indexesList);
         (10) if($\neg\text{isEmpty}(\text{D}_{\text{parent}}(h)) \land \text{isNeededUnord}(h,d_k)$)
            (11) push($D_h$, $\text{post}(d_k)$,$\text{pointerToLast}(\text{D}_{\text{parent}}(h)))$);
            (12) if($\text{isLeaf}(h) \land \text{noEmptyLists()}$
                findSolsUnord(firstLeaf(), -1, $h$, indexesList);
      (13) for each $D_i$ where $i$ ranges from 1 to $n$
         (14) if($\text{isEmpty}(D_i) \land \text{last}(q_i) < k$
             exit;

Figure 10: The unordered twig matching algorithm

all the considerations done for the ordered case are still true, but in this case condition PRU is exploited instead of PRO2, making it sufficient to only check $D_{\text{parent}}(h)$. Then, before inserting a new node (Lines 10-11 of the main algorithm), we call the boolean function $\text{isNeededUnord()}$, which, like $\text{isNeededOrd()}$, checks the condition shown in Condition POT1. In this case the only relation required and, thus, the only one that has to be checked, is the parent child one (Line 3 of $\text{isNeededUnOrd()}$). Finally, as in the other algorithms, we check if new solutions can be generated (following Lemma 16) and, in this case, call (Line 13 of the main algorithm) the recursive function $\text{findSolsUnord()}$ implementing Theorem 6.

**Lemma 21** At each step $j$ and for each query index $i$, the list $D_i$ is a subset of $\Sigma^j_i$ containing only the data entries that cannot be deleted from $\Sigma^j_i$, i.e. it has the same content as $\Delta\Sigma^j_i$ when Lemmas 7, 8, 9, 13, and 14 have been applied.
function isNeededUnord(i, d_i)
(1) if(i = 1)
(2) return true;
(3) if(isEmpty(D_{parent(i)}) \lor maxPost(D_{parent(i)} < post(d_i))
(4) return false;
(5) return true;

procedure updateDescLists(i, pos)
(1) i ← firstChild(i);
(2) if(i \neq -1)
(3) children ← i \cup siblings(i);
(4) for each h in children
(5) for each d_h in D_h in descending order
(6) if(pointer(d_h) = 1)
(7) for each \bar{d}_h from d_h in descending order
(8) dPos ← index(D_h, \bar{d}_h);
(9) delete(D_h, \bar{d}_h);
(10) if(!isLeaf(h))
(11) updateDescLists(h, dPos);
(12) break;
(13) else if(pointer(d_h) < pos)
(14) break;
(15) else
(16) decreasePointer(d_h);

Figure 11: The unordered twig matching auxiliary functions

The unordered twig matching solution construction, and all its required functions, are shown in detail in Figure 12 and 13. In this case the step-by-step backward construction behavior of the other cases would not be the best choice, since the pointers are now from children to parents and not from right to left domains. In this case the solution construction starts from the first leaf (see the initial call at Line 13 of the main algorithm), then navigates all the query nodes one by one, gradually checking and producing all the answers by extending them with the nodes contained in the associated domains with the extendSols() function. The solutions are kept in indexesList, which contains, for each of them, an index array pointing to
procedure findSolsUnord(h, prec, lastLeaf, indexesList)
(1) if(isLeaf(h))
(2) extendSols(h, prec, 0, lastLeaf, indexesList)
(3) if(h>1)
(4) findSolsUnord(parent(h), h, lastLeaf, indexesList);
(5) else
(6) extendSols(h, prec, 1, lastLeaf, indexesList);
(7) for each s in siblings(prec)
(8) preVisit(s, lastLeaf, indexesList);
(9) if(h>1)
(10) findSolsUnord(parent(h), h, lastLeaf, indexesList);

procedure preVisit(h, lastLeaf, indexesList)
(1) extendSols(h, parent(h), -1, lastLeaf, indexesList);
(2) h ← firstChild(h);
(3) if(h ≠ -1)
(4) preVisit(h, lastLeaf, indexesList);
(5) for each s in siblings(h)
(6) preVisit(s, i, cont+1);

Figure 12: The unordered twig matching solution construction (part 1)

the domain nodes, as in the path and ordered matching case.

Let us examine the way in which the query domains are navigated in order to build the solutions: Starting from the first leaf, findSolsUnord() goes up the query twig by recursively calling itself up to the query root node (Lines 4, 10 of its code), thus covering the left most path. For each of the navigated nodes having right sibilings, before navigating up it calls on each of the siblings the preVisit() function (Line 8), which recursively explores in pre-visit, from parent to child, the subtrees of the given nodes (Lines 4,6 of its code). In this way, all the query nodes (domains) are covered and we move from one domain to the other in the most suitable way w.r.t. the pointers in the domain nodes and the solution construction. Indeed, first we go up from a leaf to its parent, thus exploiting the available node pointers going in the same direction; in this way we extend the current solution only with the pointed node, which is a sort of upper bound, and the ones underlying it (the same as in path and ordered construction) (Lines
procedure extendSols(h, prec, dir, lastLeaf, indexesList)
(1) indexesListOrig ← indexesList;
(2) for each index in |indexesListOrig|
(3) if(dir=0)
(4) for i = 1 to |D_h|
(5) if (lastLeaf=h) i ← |D_h|;
(6) index[h] ← i;
(7) put index in indexesList;
(8) else
(9) d_prec ← D_prec.index[prec];
(10) if(dir>0)
(11) for each i = 1 to pointer(d_prec)
(12) if(checkPostDir(D_h.i, d_prec))
(13) extend indexes in indexesList;
(14) else
(15) i ← |D_h|
(16) while(pointer(D_h.i) >= index[prec])
(17) if(checkPostDir(D_h.i, d_prec))
(18) extend indexes in indexesList;
(19) if (lastLeaf=h) break;
(20) i ← i-1;

Figure 13: The unordered twig matching solution construction (part 2)

10-13 of extendSols()). Then, when we have reached a parent node, we
go downward from it to its other children, doing the opposite: Starting
from the last node in the child domain to be included in the solution, we
extend the solutions with all the nodes which point to the parent node, or
to any node above it (Lines 15-20 of extendSols()). In other words, in the
“downward” solution extension the parent node acts as a lower bound. The
dir parameter actually codes the direction in which to perform the extension:
1 if going up, -1 if going down. 0 is used to insert the first domain nodes in
the solutions (no actual extension, Line 2 of findSolsUnord(), Lines 3-7 of
extendSols()). During the extensions, all the post-order checks defined in
Lemma 3 (Lines 12, 17) are performed. Finally, the parameter lastLeaf is
the pre-order of the query node which started solution construction: This is
used at Line 19 of extendSols() in order to limit the extension relative to
this domain only to the last inserted node. This is necessary since, as we said
before, the node starting solution construction is not deleted immediately after it, as in the other matching algorithms, and therefore, without this check, duplicate solutions would be generated.

**Theorem 9** For each data node \( d_j \), \( S = (s_1, \ldots, s_n) \in \Delta \text{Uns}^i_Q(D) \) iff the algorithm, by calling the function \( \text{findSolsUnord}() \), generates the solution \( S \).

5 Experimental evaluation

In this section we present the results of the experimental evaluation of the matching algorithms we described in the previous section. In particular, we show the performance of each of the algorithms and we evaluate the benefits offered by each of the conditions discussed above, both in terms of the reduced size of the domains and in the amount of saved time w.r.t. their standard execution.

5.1 Experimental setting

![Table 4: The XML data collections used for experimental evaluation](image)

In our tests, we used both real and synthetic data sets. In this paper, we present the results we obtained on one real and two synthetic collections (see Table 4). In order to show the performance of the matching algorithms on real-world “data-centric” XML scenarios, we chose the complete DBLP Computer Science Bibliography archive. The file consists of over 3.8 Millions of elements. It is a very “flat” (3 levels) and wide (very high root fan out) data tree, as in typical “data-centric” XML documents. As in most real data sets, the distribution of the node labels is non equiprobable. In fact, the whole set presents repetitions of typical patterns (for instance, “article-author”). Since typical real data sets are very flat, this would not allow us to test some of the most complex conditions, such as POT2. For this reason, we also generated two synthetic data sets, Gen1 and Gen2, as random trees.
using the following parameters: depth (5 and 8, respectively), fan-out (3) and root fan-out (50000 and 30, respectively). Both synthetic collections differ from the DBLP set in their labels distribution as they are uniformly distributed. However, while Gen1 proposes a similarly wide and slightly deeper tree, Gen2 is very deep and has smaller root fan-out, thus simulating more "text centric" trees. Note that the size of collections (in particular the root fan-out) is not very important. Our aim is mainly to analyze the behavior of the algorithms and the trends of the sizes of domains. This is typically clear after a significant portion of the data is scanned.

Figure 14: The query paths twigs used in the tests

Figure 14 shows our testing queries. The upper row shows the queries on the DBLP collection (denoted with Dn), while the lower one defines queries for the synthetically generated collections (denoted with Gn). In both cases, we provide a path and two twigs. For DBLP, the query depth is limited to 2, therefore we tried to differentiate the queries by means of an increasing fan-out. The labels are specifically chosen amongst the less selective, in order to test our algorithms in the most demanding settings. As to Gen1 and Gen2, we created queries G2 and G3 deeper than the DBLP ones. They are specifically conceived to test all the conditions which would not be activated in the shallow DBLP setting.

The algorithms are implemented in Java JDK 1.4.2; the experiments are executed on a Pentium 4 2.5Ghz Windows XP Professional workstation,
equipped with 512MB RAM and a RAID0 cluster of 2 80GB EIDE disks with NT file system (NTFS).

5.2 General performance evaluation

<table>
<thead>
<tr>
<th>Query</th>
<th>Time (msec)</th>
<th># sols</th>
<th>sols / constr</th>
<th>MDS</th>
<th>Insertions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#</td>
</tr>
<tr>
<td>P-D1</td>
<td>1109</td>
<td>260540</td>
<td>1</td>
<td>1</td>
<td>390008</td>
</tr>
<tr>
<td>O-D2</td>
<td>2890</td>
<td>553997</td>
<td>2.32</td>
<td>1.44</td>
<td>1038632</td>
</tr>
<tr>
<td>O-D3</td>
<td>5984</td>
<td>98934</td>
<td>2.58</td>
<td>1.17</td>
<td>1843762</td>
</tr>
<tr>
<td>U-D2</td>
<td>3750</td>
<td>559209</td>
<td>2.3</td>
<td>1.43</td>
<td>1041697</td>
</tr>
<tr>
<td>U-D3</td>
<td>7266</td>
<td>149700</td>
<td>2.56</td>
<td>1.17</td>
<td>2245014</td>
</tr>
<tr>
<td>P-G1</td>
<td>941</td>
<td>13209</td>
<td>1.09</td>
<td>1.05</td>
<td>390036</td>
</tr>
<tr>
<td>O-G2</td>
<td>1392</td>
<td>848</td>
<td>1.59</td>
<td>1.32</td>
<td>392751</td>
</tr>
<tr>
<td>O-G3</td>
<td>2394</td>
<td>94</td>
<td>2.35</td>
<td>1.59</td>
<td>393119</td>
</tr>
<tr>
<td>U-G2</td>
<td>2238</td>
<td>1412</td>
<td>1.53</td>
<td>1.47</td>
<td>487100</td>
</tr>
<tr>
<td>U-G3</td>
<td>2854</td>
<td>920</td>
<td>2.59</td>
<td>1.75</td>
<td>587464</td>
</tr>
<tr>
<td>P-G1</td>
<td>90</td>
<td>1320</td>
<td>1.4</td>
<td>1.2</td>
<td>8425</td>
</tr>
<tr>
<td>O-G2</td>
<td>113</td>
<td>884</td>
<td>7.82</td>
<td>2.93</td>
<td>8511</td>
</tr>
<tr>
<td>O-G3</td>
<td>190</td>
<td>3173</td>
<td>6.47</td>
<td>5.99</td>
<td>8709</td>
</tr>
<tr>
<td>U-G2</td>
<td>180</td>
<td>2788</td>
<td>10.64</td>
<td>5.4</td>
<td>11061</td>
</tr>
<tr>
<td>U-G3</td>
<td>961</td>
<td>26854</td>
<td>132.9</td>
<td>7.92</td>
<td>14039</td>
</tr>
</tbody>
</table>

Table 5: Pattern matching results for the different queries and collections

Table 5 summarizes the results obtained by applying our pattern matching algorithms to solve the proposed queries, for each of the three collections. Queries are denoted with a prefix signifying the applied matching algorithm (P- for path matching, O- for ordered and U- for unordered twig matching). For each query setting, we present the fundamental details of the algorithms execution, i.e. the total execution time (in milliseconds), the total number of solutions retrieved, the mean number of solutions constructed each time a solution construction is started, the mean domain size (denoted with MDS), the total number of node insertions, and the percentage of avoided insertions with respect to their total number. Observe that in all the cases the time is in the order of a few seconds (7 seconds at most for query U-D3), even though each of the settings presents non-trivial query execution challenges: a very wide data tree for both DBLP and Gen1, a considerable repetitiveness
for DBLP labels and patterns (notice the very high number of solutions, over half a million for queries D2) and a very deep and involved tree for Gen2 (notice the high number of solutions for each solution construction, nearly one or even two orders of magnitude larger than for the other two collections). Also observe the large number of node insertions, and especially the high percentage of avoided insertions, which is very significant for all collections (e.g. nearly a million avoided insertions for DBLP O-D3 query, while in Gen1 and Gen2 twigs the number of non-inserted nodes is much higher than the ones inserted). Finally, the MDS parameter is particularly significant for all the queries. It represents the mean size of the domains measured each time a solution construction is called for the whole size of the collection. Its low values in each of the settings (less than 1.8 nodes for DBLP and Gen1, reaching 7.92 for the most complex query in the deep Gen2) testify the good efficacy of our reduction conditions. In particular, since the mean domains size is low, this means that the number of deletions is very near to the number of insertions. Keeping the MDS low is essential for efficiency reasons, since the time spent in constructing the solutions is roughly proportional to the Cartesian product of the domains size, but, in many cases, it is also essential for the good outcome of the matching, since an overflow of the domains would mean a total failure.

5.3 Evaluating the impact of each condition

<table>
<thead>
<tr>
<th>Case</th>
<th>Post-order</th>
<th>Pre-order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POP POT1 POT2 POT3</td>
<td>PRO1 PRO2(PRU)</td>
</tr>
<tr>
<td>P-A</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>P-B / T-E</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>P-C / T-F</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>T-A</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>T-B</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>T-C</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>T-D</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Summary of the discussed cases (disabled conditions are marked with a ‘x’)

Now that the effectiveness of the algorithms and of the conditions is clear, we still need information about the benefits offered by enabling each of them. In the following, we deeply analyze the MDS, i.e. the domains behavior and execution time, measuring how much each of the conditions
applied in the algorithms influence their trend. In order to simplify the analysis we discuss the path and twig matching separately, identifying the most significant cases for each of them. We will present all the specific results and graphs only for some of the most complex and interesting cases, shortly discussing the others in words. Table 6 presents a summary of the cases we will discuss, together with the associated disabled conditions. Let us start by considering the simpler path matching scenario, where we distinguish between cases P-A, P-B and P-C. In the following, we refer to the case where all the conditions are turned on as the “standard” (std) case. First is the case P-A where we turn off the post-order condition and which is expected to significantly degrade the matching performance since such a condition is clearly the key to the high number of node deletions. In fact, case P-A produces an uncontrolled growth in the domains’ size, preventing the conclusion of the matching for all the query settings, both for domain size overflow and for the consequently exploded execution time. Then are the cases involving deactivation of pre-order conditions (P-B, P-C), which again influence deletions but in a lesser manner. Case P-B means we do not empty the domains on the “right” when a given domain becomes empty (thus there will be “dangling” pointers), while in P-C the nodes in the last domain are no longer deleted after solution construction. Case P-B produces larger but still controlled domain sizes (20%-30% higher MDS than the “standard” cases), while the execution time is nearly unchanged (only 2% less). This is expected, since, at least for short paths, the time required to apply the conditions compensates the shorter solution generation time. Notice that the modified path algorithm for Case P-B is equivalent to the one proposed in [BKS02]. Finally, with case P-C we obtained results which were almost identical to the standard case. The deletion of the nodes in the last stacks, which would be immediately provided by condition PRO1, is equally carried out by the other conditions (i.e. POP, PRO2) just a few steps later, on mean. This results in nearly identical execution time and mean domain size (nearly 6% larger).

As to twig matching, the number of available conditions requires a deeper analysis. As for paths, we will first inspect the pre-order conditions (i.e. POT), which are the main source of avoided insertions and deletions. As shown in previous section, the following are key functions which activate POT conditions: `isCleanable()` for deletions (exploiting POT2 and POT3) and `isNeeded()` (which will be `isNeededOrd()` for the ordered and `isNeededUnord()` for the unordered case) for insertions (exploiting POT1). We started by analyzing the “percentages of success” of such functions for each call in each of the queries – Table 7 provides such statistics. For
### Table 7: Behavior of the `isCleanable()` and `isNeeded()` functions

<table>
<thead>
<tr>
<th>Query</th>
<th><code>isCleanable()</code></th>
<th><code>isNeeded()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Calls</td>
<td>% true (POT3)</td>
</tr>
<tr>
<td><strong>DBLP collection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O-D2</td>
<td>1389881</td>
<td>17.36%</td>
</tr>
<tr>
<td>O-D3</td>
<td>7477979</td>
<td>3.23%</td>
</tr>
<tr>
<td>U-D2</td>
<td>1395093</td>
<td>17.29%</td>
</tr>
<tr>
<td>U-D3</td>
<td>7866217</td>
<td>3.07%</td>
</tr>
<tr>
<td><strong>Gen1 collection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O-G2</td>
<td>417154</td>
<td>68.48%</td>
</tr>
<tr>
<td>O-G3</td>
<td>456767</td>
<td>62.54%</td>
</tr>
<tr>
<td>U-G2</td>
<td>542592</td>
<td>52.65%</td>
</tr>
<tr>
<td>U-G3</td>
<td>762319</td>
<td>37.47%</td>
</tr>
<tr>
<td><strong>Gen2 collection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O-G2</td>
<td>27645</td>
<td>17.1%</td>
</tr>
<tr>
<td>O-G3</td>
<td>57733</td>
<td>8.17%</td>
</tr>
<tr>
<td>U-G2</td>
<td>55344</td>
<td>8.54%</td>
</tr>
<tr>
<td>U-G3</td>
<td>109385</td>
<td>4.3%</td>
</tr>
</tbody>
</table>

The `isCleanable()` “success” means allowing a node deletion (returning true, both for POT3 or POT2), while for `isNeeded()` means preventing an useless insertion (returning false for POT1). Notice that POT1 can be satisfied by examining nodes in the parent domain (denoted in table with POT1p) or, only for unordered, in the siblings ones (POT1s in table). The percentage of success for both functions is considerable in all cases. In DBLP, the percentage of deletion success is lower than in the other collections. This is due both to the more repetitive and simple structure of its data tree and to the inapplicability of condition POT2 (DBLP queries have only two levels). Such condition proves instead quite useful in the other collections, particularly for unordered matching, where its application comes often near the one of POT3. As to POT1, the main contribution is given in situation POT1p, while also POT1s can give a good contribution in the ordered matching. About the two functions we are discussing, we also performed some CPU utilization tests and found out that their contribution is generally significant also from an execution time point of view, since their percentage of CPU utilization is typically less than 4% of the total CPU times.

To quantify the specific effects of the conditions on the domain size and time, we distinguish cases T-A to T-D (see Table 6). The first three
cases will clearly produce less node deletions, while case T-D will allow useless insertions. Case T-A is conceptually equivalent to the case P-A, i.e. deletions are almost totally prevented. Like in P-A, as expected, we found out that the domain sizes grow uncontrollably, preventing the termination of the matching in acceptable time. As an example, Graph 15-a shows a plot comparing, after each data node, the mean stack size of case T-A to the standard one for O-G3 query, collection Gen1. As to cases T-B and T-D, the algorithms generally produced domain sizes which were larger than the standard case (see Graph 15-b). Even if the difference in size may not seem particularly significant, we have to consider that the time spent in constructing the solutions is roughly proportional to the cartesian product of the size of each domains, thus the differences in execution time may become more evident. For instance, if the domains are on mean one and a half larger,
as for query UG-3 (collection Gen2, case T-D), each solution construction run becomes nearly 20 times longer. While for the most simple queries we found out that execution time is still not much affected, since the time required to check the conditions compensates the shorter solution generation time, for the most complex settings the difference in execution time can be remarkable. As an example, Figure 16 shows, for the most complex queries and for collection Gen2, the comparison between the standard case time and the one of the T-D case. Notice that, in order to verify execution time savings in more complex situations, we also employed new queries specifically for these tests, e.g. a modified version of query U-G2, named U-G2b, where second level nodes have two children instead of one. As seen in the graph, the difference in execution time can reach a proportion of 1:5 (query U-G2b). The results obtained for case T-C are very different between the ordered and unordered settings. Disabling condition POT3 produces almost no variations in the ordered matching (condition POT2 produces the same deletions at the cost of a little more time spent in checking the hypotheses), while it proves essential for unordered matching (time and domain size grow uncontrolably). Note that if we disabled PRO1 together with POT3, case T-C would degenerate for the ordered setting too, since the last domain would not be empty and POT2 could no longer be always applied.

Finally, as for the paths, we can also briefly analyze the cases involving the activation of the pre-order conditions (PRO, PRU), denoted as T-E and T-F in Table 6. Simulating case T-E (which is conceptually similar to the P-B one for paths) means to disable the deletions produced by the pointers update, i.e. there will be dangling pointers. Differently from P-B, such case produces uncontrolled growth in domains size and in time, proving that such conditions are essential for more complex queries. As to case T-F, this is equivalent to case P-C and the results obtained for ordered matching confirm the one discussed for such case.

6 Conclusions

In this paper, we dealt with the three problems of pattern matching (path, ordered and unordered twig matching) by exploiting the tree signature approach. In particular, we studied the properties and conditions allowing us to maintain the storage structures used in the algorithms as compact as possible. This is essential for efficiency reason but also for avoiding overflows which obviously compromise the good outcome of the process. In particular, as to our knowledge, this is the first attempt of defining a set of reduction
conditions, based on the pre/post-order numbering scheme, which is complete and which is applicable to the three kinds of tree pattern matching. We showed that such a theoretical framework can be applied for building pattern matching algorithms whose efficiency is proved by extensive experimental evaluation.

References


