XFIS: an XML filtering system based on string representation and matching

Panagiotis Antonellis* and Christos Makris

Computer Engineering and Informatics Department
Patras University
Rio 26500, Greece
E-mail: adonel@ceid.upatras.gr
E-mail: makri@ceid.upatras.gr
*Corresponding author

Abstract: Information-filtering systems constitute a critical component of modern information-seeking applications. As the number of users grows and the amount of information available becomes even bigger, it is imperative to employ scalable and efficient representation and filtering techniques. Typically, the use of eXtensible Markup Language (XML) representation entails profile representation with the use of the XPath query language and the employment of efficient heuristic techniques for constraining the complexity of the filtering mechanism. In this paper, we propose an efficient technique for matching user profiles that is based on the use of holistic twig-matching algorithms and is more effective, in terms of time and space complexities, in comparison with previous techniques. The proposed algorithm is able to handle order matching of user profiles, while its main positive aspect is the envisaging of a representation based on Prüfer sequences that permits the effective investigation of node relationships. Experimental results showed that the proposed algorithm outperforms the previous algorithms in XML filtering both in space and time aspects.

Keywords: eXtensible Markup Language; XML; filtering; string representation.

Reference to this paper should be made as follows: Antonellis, P. and Makris, C. (2008) 'XFIS: an XML filtering system based on string representation and matching', *Int. J. Web Engineering and Technology*, Vol. 4, No. 1, pp.70–94.

Biographical notes: Panagiotis Antonellis graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in July 2005. He is now an MSc student in the Department of Computer Engineering and Informatics. His research interests include databases, information retrieval and software quality.

Christos Makris graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in December 1993. He received his PhD degree from the Department of Computer Engineering and Informatics in 1997. At present he works as an Assistant Professor in the Department of Computer Engineering and Informatics, University of Patras. His research interests include data structures, computational geometry, databases and information retrieval. He has published over 50 papers in scientific journals and refereed conferences.

1 Introduction

Information-filtering systems (Aguilera *et al*., 1999; Carzanica *et al*., 2001; Tian *et al*., 2004) are systems that provide two main services: document selection (*i.e.*, determining which documents match which users) and document delivery (*i.e.*, routing matching documents from data sources to users). In order to implement these services efficiently, information-filtering systems rely upon representations of user profiles, which are generated either explicitly by asking the users to state their interests, or implicitly by mechanisms that track the user behaviour and use it as a guide to construct the user profile. Initial attempts to construct such profiles typically used 'bag of words' representations and keyword similarity techniques (closely related to the well-known vector space model representation in the Information Retrieval area) to represent user profiles and match them against new data items. These techniques, however, often suffer from limited ability to express user interests, being unable to fully capture the semantics of the user behaviour and user interests. As an attempt to face this lack of expressiveness, a number of systems have appeared lately that use XML representations for both documents and user profiling and that employ various filtering techniques for matching the XML representations of user documents with the provided profiles (Altinel and Franklin, 2000; Chan *et al*., 2002; Diao *et al*., 2003; Kwon *et al*., 2005; Tian *et al*., 2004).

eXtensible Markup Language (XML) is becoming the standard for information exchange, especially on the internet. XML is a textual representation of data that is designed for the description of the content rather than the presentation of data. The language permits the description of new structures, the nesting of structures in arbitrary depth and the optional description of its grammar. The basic lexical notion used in XML is the element, which is the piece of text used bounded by matching tags, where within an element we may have text, other elements or a mixture of the two. XML allows us to associate attributes with elements, where the term 'attribute' denotes a (name, value) pair. The structure of an XML document is best modelled as a labelled tree: elements and attributes are mapped to nodes in the tree and direct nesting relationships are mapped to edges in the tree.

The basic mechanism used to describe user profiles in XML format is through the XPath query language (Berglund *et al*., 2002). XPath is a query language for addressing parts of an XML document, while also providing basic facilities for the manipulation of strings, numbers and booleans. XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes; and XPath defines a way to compute a string-value for each type of node. Other known XML query languages used in various applications are XQuery (Chamberlin, 2002) and XQL.[1]

The process of XML filtering is related to, but different from, the more traditional XML data-retrieval problem, where, given a stored collection of XML data objects and a query, the system needs to identify those data instances which satisfy the given query. XML query-processing approaches concentrate on finding effective mechanisms for indexing XML data objects to efficiently retrieve or check structural relationships. In contrast, in XML filtering, instead of the data (which is transitionary), the collection of filter patterns (user profiles) needs to be indexed. When an XML document arrives, the system filters it through the stored profiles to identify with which of them the document fits. After the filtering process is finished, the document can be sent to the corresponding

users with matching profiles. User profiles are expressed as XML twig patterns and are stored in XPath format. In such a system it is vital to filter the XML document towards all user profiles in one pass to save time and avoid complexity. Such a system usually contains some other subsystems, such as two XPath parsers (one for twigs and one for documents) and an Index (for user profiles). However, the main purpose of an XML filtering system is to find all the user profiles that have a match with a specific XML document.

1.1 Existing approaches and challenges

The existing XML filtering systems can be categorised as follows:

1.1.1 Automata-based systems

The prominent examples of automata-based systems are XFilter (Altinel and Franklin, 2000), Yfilter (Diao *et al*., 2003) and Distributed XML Stream Filtering (Uchiyama *et al*., 2005). Systems in this category incorporate Finite State Automata (FSA) to quickly match the document with the user profiles. In these systems, each data node causes a state transition in the underlying FSA representation of the filters. In XFilter, user profiles are represented as queries using the XPath language and the filtering engine employs a sophisticated index structure and a modified Finite State Machine (FSM) approach to quickly locate and examine relevant profiles. XFilter employs a separate FSM per path query and a novel indexing mechanism to allow all of the FSMs to be executed simultaneously during the processing of a document. A major drawback of XFilter is its lack of twig-pattern support, as it handles only linear path expressions. Building on the insights of the XFilter work, a new method was described in Diao *et al*. (2003) termed Yfilter, which combined all of the path queries into a single Nondeterministic Finite Automaton (NFA) and exploited commonality among queries by merging common prefixes of the query paths, such that they were processed once at the most. The resulting shared processing provided tremendous improvements to the performance of structure matching but complicated the handling of value-based predicates. Unlike XFilter, YFilter handles twig patterns by decomposing them into individual linear paths and then performing postprocessing over linear path matches. The novelty of the Distributed XML Filtering System is the distribution of the filtering and transferring load among many Filtering Servers (FSs). Each FS uses a DFA mechanism to filter the incoming XML documents; and the total transferring load, for publishing the filtered XML data to the corresponding subscribers, is shared among the FSs. To further improve its scalability, the system utilises a technique to forecast the transfer load of each FS, based on the user profiles features.

1.1.2 Sequence-based systems

Systems in this category represent both the user profiles and the XML documents as string sequences and then perform subsequence matching between the document's sequence and profile sequences. FiST (Kwon *et al*., 2005) employs a novel holistic matching approach that, instead of breaking the twig pattern into separate root-to-leaf paths, transforms (through the use of the Prüfer (1918) sequence) the matching problem into a subsequence matching problem. In order to provide more efficient filtering, user

profile sequences are indexed using hash structures. Moreover, FiST is able to handle efficient, ordered twig-pattern matching. XTrie (Chan *et al*., 2002) encodes each node with its root path and constructs a sequence for every user profile. Those sequences are indexed in a trie-like structure and the index is then used to filter incoming XML documents against the stored user profiles.

1.1.3 Stack-based systems

The representative system of this category is AFilter (Canadan *et al*., 2006). AFilter utilises a stack structure while filtering the XML document against user profiles. Its novel filtering mechanism leverages both prefix and suffix commonalities across filter statements, avoids unnecessarily eager result/state enumerations (such as NFA enumerations of active states) and decouples the memory management task from result enumeration to ensure correct results even when the memory is tight.

1.1.4 Push-down approaches

XPush (Gupta and Suciu, 2003) translates the collection of filter statements into a single deterministic push-down automaton. The XPush machine uses a SAX parser that simulates a bottom-up computation and hence does not require the main memory representation of the document. The construction of the XPush machine is done in a lazy manner and though there is a high cost associated with computing a state for the first time, the cost is recovered later when the state is to be reused. The lazy XPush machine has a number of advantages in dealing with the inconsistencies and regularities in the Document Type Definitions (DTD) and also in avoiding the construction of states that do not occur in the given data set. XSQ (Peng and Chawathe, 2005) utilises a hierarchical arrangement of push-down transducers augmented with buffers. A notable feature of XSQ is that it buffers data only for as long as it must be buffered by any streaming XPath query engine.

Table 1 summarises the main characteristics of existing XML filtering schemes.

Table 1 Existing XML filtering systems

XML filtering system	*Filtering mechanism*	*Twig support*	*Additional characteristics*
XFilter	FSM	No	–
YFilter	NFA/DFA	Yes	Detection of common prefixes
FiST	Subsequence matching	Yes	High scalability, ordered matching
AFilter	Stack	No	Exploitation of prefix and suffix commonalities, lazy techniques
XTrie	Subsequence matching	Yes	Substring indexing, substring sharing, ordered matching
XPush	Push-down automaton	Yes	High scalability, lazy techniques
XSQ	Push-down transducers	Yes	Streaming processing
Distributed XML Stream Filtering	NFA/DFA	Yes	High scalability, distributed filtering and transfer-load balancing, lazy techniques

1.2 Motivation and paper's contribution

Filtering systems supporting twig-pattern user profiles (*e.g.*, YFilter, FiST) always require an additional postprocessing step to finalise the right results. YFilter and XTrie handle twig-pattern user profiles by decomposing them into individual linear paths and then performing postprocessing over linear path matches. FiST, XPush and Distributed XML Stream Filtering, while utilising the holistic process of twig patterns, require a final step in order to identify and discard false matches derived from false branch node matches.

Our motivation is to design a filtering system that utilises a holistic matching approach in such a way that no extra branching node verification phase is needed. In order to provide more efficient filtering, the system should also be able to filter the incoming XML documents online. With these in mind, we introduce XFIS (Section 2), which encompasses the following advantages:

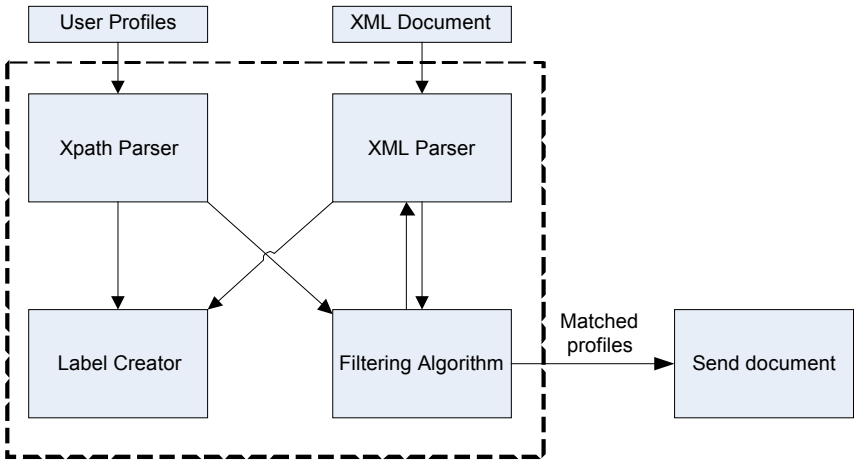
- *Holistic matching of twig-pattern user profiles.* The proposed system utilises a novel string encoding of nodes and edges to construct a string sequence for every user profile and XML document, transforming the matching problem into a subsequence matching problem.
- *Elimination of extra postprocessing step.* XFIS, based on the properties of the adopted string encoding, utilises an extra conditional check while performing subsequence matching in order to smartly discard false branching node matching. This eliminates the need for an extra branching node verification phase, which adds to the complexity of the previous algorithms.
- *Online filtering of XML documents.* Providing support for online filtering, our system minimises its memory footprint and increases its throughput, as the filtering of an XML document can start at the time the document arrives in our system.
- *Ordered twig matching.* XFIS provides ordered twig matching for applications that require the nodes in a twig pattern to follow document order in XML.

The rest of the paper is structured as follows: Section 2 introduces the Tree Structure Sequences and presents the main components of the XFIS system; Section 3 discusses the experimental results; Section 4 discusses the practical-application aspect of the XFIS system; Section 5 presents our conclusions; and, finally, the Appendix presents in detail the pseudo codes of our XFIS implementation.

2 Tree-Structure Sequence and XFIS system architecture

The architecture of XFIS is depicted in Figure 1. The system consists of four basic subsystems: the *XPath parser*, the *XML parser*, the *Label Creator* and the *Filtering Algorithm*. Before continuing with each subsystem separately, we need to introduce the notation of *Tree-Structure Sequence (TSS)*, a novel string representation for XML trees, utilised by XFIS to encode both user profiles and XML trees.

Figure 1 XFIS architecture



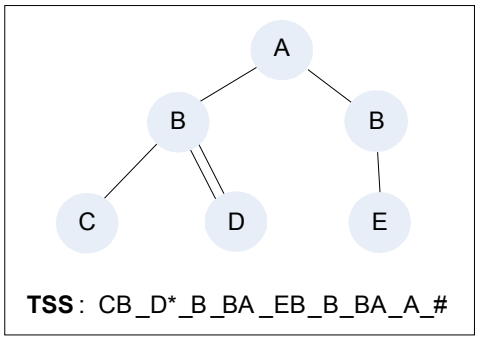
2.1 Tree-structure sequence

In order to be able to eliminate the final *branch node refinement* phase as well as to encode both parent-child and ancestor-descendant relationships of a user profile, we would like an appropriate string representation that could be applied to both twigs and XML documents. Based on postorder traversal and the Prüfer sequences, we introduce a new string representation of XML documents that can be applied in any tree-structured document, the TSS. Given a tree-structured document T, the TSS of T is a string representation of T's nodes and edges. Each edge is represented by a pair of char labels, defining the edge's attached nodes. Each internal node is represented in TSS(T) by its char label. The construction of the TSS begins from the leftmost leaf and follows a postorder traversal of the tree. Internal nodes appear in TSS only after all their outgoing edges have been represented. The limitation introduced is that all the nodes must have one-char labels, hence a preprocessing step is needed to assign one-char labels to each distinct tag.

The algorithm to construct the TSS of a given tree T_n works as follows: Let S denote the TSS of the tree. Initially, S is an empty string. Because gaps are important in TSS, from now on, we denote them by the underline character: '_'. Begin from the leftmost leaf of the tree, following postorder traversal, and let t_1 and t_2 denote the label of this node and its parent respectively. If the current node is a leaf, then append $t_1t_2\_$ to S and continue with the next node in the postorder traversal; if the current node is an internal node, then append $t_1\_t_1t_2\_$ to S; if the current node is the root node, then append $t_1\_\#$ to S, where t_1 is the root node label. Continue with this process until the root node is reached. In the case of twig patterns, where ancestor-descendant relationships exist, the algorithm works in the same way; but when a node is reached that is a descendant of its 'parent' node (*e.g.*, it has a ancestor-descendant relationship with its parent), instead of appending t_1t_2 to S, we append t_1*. This technique is very important during the process of twig matching using subsequence string matching; its importance will be explained later in this paper.

Figure 2 depicts an example of a twig's TSS constructed in the way mentioned above. Let us denote with *S* the TSS of the XML tree in Figure 2. The first leftmost leaf of the tree is Node C and its parent is Node B. Thus, we append CB_ to *S*. Following the postorder traversal we reach Node D, which is a descendant of Node B. Thus, we append D*_ to *S*. The next node in the postorder traversal is Node B, which is an internal node. As a result, we append B_BA_ to *S*. By this time, *S* equals CB_D*_B_BA_. Continuing analogously, the final TSS of the XML tree is CB_D*_B_BA_EB_B_BA_A_#.

Figure 2 TSS example



Because of the way the TSS is constructed, it is easy to compute an approximate bound of its size. Let us consider a tree T with *n* nodes and *m* edges. The size required to represent the edges of T is 2*m*, because each edge of T is represented by a pair of char labels in TSS(T). Additionally, each internal node is represented by a single char label, so the required space is at most *n* (depending on the number of leaf nodes). Until now, the required size is upper bounded by *O(m+n)*. Because the edges' and nodes' representations are separated by gaps in TSS, an additional cost of *O(m+n)* must be added in the upper bound. The resulting upper bound of TSS's size is *O(m+n)*. It is obvious that in the case of large documents, the TSS's size becomes big enough. On the other hand, in the case of twig patterns, usually with a small number of nodes and edges, the size of the corresponding TSS remains small. In order to avoid storing the whole TSS of the XML document, XFIS progressively constructs the document's TSS while parsing the XML document and simultaneously compares it with the user profiles' TSS. So only a small part (usually 3–4 chars) of the document's TSS is stored each time.

Because of the method of its computation, the TSS has the following important property:

Property 1 *Given two trees P and T, if P is a subtree of T, then TSS(P) is a substring of TSS(T).*

The proof of this property is as follows: Let us denote $P(V_P, E_P)$ and $T(V_T, E_T)$ as the two given trees. Based on the assumption that *P* is a subtree of *T*, then $V_P \subseteq V_T$ and $E_P \subseteq E_T$. Additionally, the nodes in V_P during a postorder traversal in *P* appear in the same order as they appear in a postorder traversal in *T*. This leads to the result that TSS(P) is a substring of TSS(T).

2.2 Label Creator

Because of the one-char label limitation introduced by TSS, it is vital for our system to assign distinct char labels to every distinct tag. This task is accomplished by the Label Creator subsystem, which constructs and keeps track of a one-to-one correspondence between distinct tags and char labels. Every distinct tag is assigned a distinct char label, derived from an internal char label source. The Label Creator interacts both with XPath parser and XML parser in order to compute user profile and document TSSs.

2.3 XPath parser

The XPath parser processes user profiles expressed in XPath, and computes and stores the TSS of each profile. As mentioned before, user profiles are expressed using XPath notations and can be easily represented as XML trees. The XPath parser processes XPath expressions and accordingly constructs a memory tree representation for every user profile. For every such tree, it computes the respective TSS using the aforementioned methodology. The computed user profile TSSs are permanently stored by the system in text files.

2.4 XML parser

Similarly to the XPath parser, the XML parser is assigned the task of parsing incoming XML documents and constructing their corresponding TSSs. Simultaneously, every new element appended to the document's TSS is sent to the filtering algorithm in order to be checked online against user profiles, thus providing online filtering of the XML document. The XML parser subsystem utilises a SAX parser, its handlers and a stack structure, named *tagStack*, in order to increasingly compute and construct a document's TSS. The *startTag* handler is invoked whenever the start of an element's tag is reached, while the *endTag* handler is invoked whenever the end of an element's tag is reached.

Figure 3 presents the code for the *startTag* and *endTag* SAX parser handlers. When the *startTag* handler is invoked with a tag name, the system first locates the tag's char label using the Label Creator and then pushes this label into *tagStack.* When the *endTag* handler is invoked with a tag name, the system first finds the corresponding char label using Label Creator.

If the tag is not a leaf node (*e.g.*, it has children), the char label of the node is added into the document's TSS by calling the function *addCurrentNode*(), and then the top char label from the *tagStack* (which is the current node's label) is popped. If the *tagStack* is not empty (*e.g.*, the root node has not been reached), the system finds the char label of the current node's parent (which is the top element of *tagStack*) and appends the pair (*label, plabel*) representing the corresponding edge into the document's TSS by calling the function *addEdgeNode*(). On the other hand, if *tagStack* is empty (*e.g.*, the root node has been reached), the system appends to the document's TSS the root node characteristic sequence by calling the *addRootNode*() function.

Figure 4 depicts an example of the above method. For demonstration purposes, we assume that every tag consists of a single char, thus eliminating the need for the Label Creator. Figure 4(a) summarises the system's state after the *endTag* handler is invoked with the tag C. At this time, the *tagStack* contains A, B and C, while the document's TSS is still empty. Node C is a leaf, so when the *endTag* is invoked, the top element (C) of

tagStack is popped and the system appends CB_ to the document's TSS. Figure 4(b) summarises the system's state after the *endTag* handler is invoked with the tag D. At this time, the *tagStack* contains A, B, F and D, while the document's TSS is CB_. Node D is a leaf, thus the top element (D) of *tagStack* is popped and DF_ is appended to the document's TSS. Finally, Figure 4(c) depicts the case when the *endTag* handler is invoked with the tag F. Before the handler is invoked, the *tagStack* contains A, B and F. Node F is an internal node, so when the handler is invoked, the top element (F) of *tagStack* is popped and F_FB_ is appended to the document's TSS.

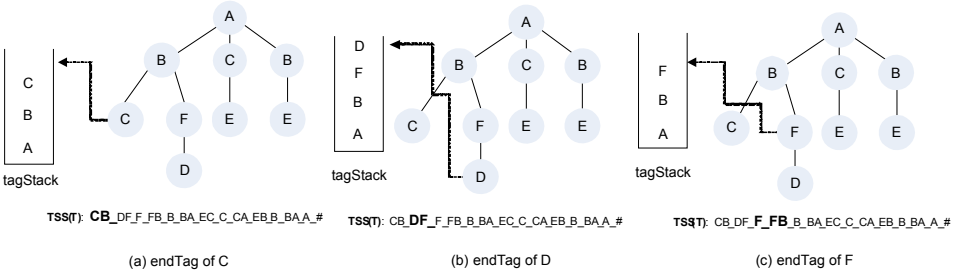
Figure 3 Pseudo code for SAX parser handlers

```
stack tagStack;
handler startTag (tag){
        label = assign_label(tag);
        tagStack.push(label);
}

handler endTag (tag){
        label = assign_label(tag);
        if (not leafnode)
                addCurrentNode(label);
        tagStack.pop();
        if (not tagStack.isEmpty() )
        {
                plabel = tagStack.top();
                addEdgeNode(label, plabel);
        }
        else
                addRootNode();

}
```

Figure 4 Using tagStack and SAX parser's handlers to create a document's TSS



(a) endTag of C (b) endTag of D (c) endTag of F

It is important to mention that the SAX parsers interact online with the filtering algorithm. Whenever a new element is appended to the document's TSS, it is immediately sent to the filtering algorithm to check it against user profiles' TSSs. We describe the filtering algorithm in the next section.

2.5 Filtering algorithm

While the XML parser computes the TSS of an incoming XML document, the filtering algorithm is used to identify those twigs that have a match in the document. The filtering is done by identifying those twigs that have their TSS as a subsequence of the document's TSS, according to some branch-based criteria to be described later. The key issue in the filtering algorithm is to find all the matching twigs in one pass of the document's TSS.

A naive approach to this problem is to compare each character of the document's TSS with the current character of all twigs' TSSs. This approach, although easy to implement, has the disadvantage of checking many unnecessary twig TSSs. We instead chose to use a more complicated though more efficient approach, which is to index the twig TSSs in order to limit the number of comparisons in every step of the filtering algorithm. The indexing method we used is a dynamic hash table called *twigPositions.* This hash table keeps track of the current char label of every twig's TSS, and uses the XML char labels as keys. For each label, the value stored in the hash table is a list of all twig IDs whose TSS's current char is the key label. Hence in every step, the filtering algorithm uses the document's TSS current character as a key in the hash table in order to locate those twigs that their corresponding TSS's current character is the same as the used key. In order to keep track of the twig TSSs' current positions, a global table, named *twigPointers*, is used with a size equal to the number of stored twigs. Each table position corresponds to a twig and stores the current integer position of the twig's TSS, starting from 0. Moreover, because of the TSS's particular structure, the subsequence matching is not performed by using the characters one by one, but by collecting them in groups. For example, consider a part of a twig's TSS: …*GB_ FB_B_BA*… Each of the pairs of char labels *GB*, *FB* and *BA* should be matched as a pair and not as individual nodes. For this purpose, in every step of the matching process, both the current and the next char labels of the twig and document TSSs are taken into consideration. Let us denote by *TC* and *TN* the twig TSS's current and next char label respectively, and by *DC* and *DN* the document TSS's current and next char label respectively.

A matching occurs in any one of the following three cases:

Case 1 (DN = '_' and TN='_') and (DC=TC)

Case 2 (DN != '_' and TN != '_') and (DC=TC) and (DN =TN)

Case 3 (DN != '_' and TN != '_') and (DC=TC) and (TN='*').

We distinguish between Cases 1 and 2, because each of these cases results in a different update of the twig and document TSSs' current position.

In Case 1, both the twig and document TSSs' current positions must be incremented by 2 in order to point to the next pair of char labels.

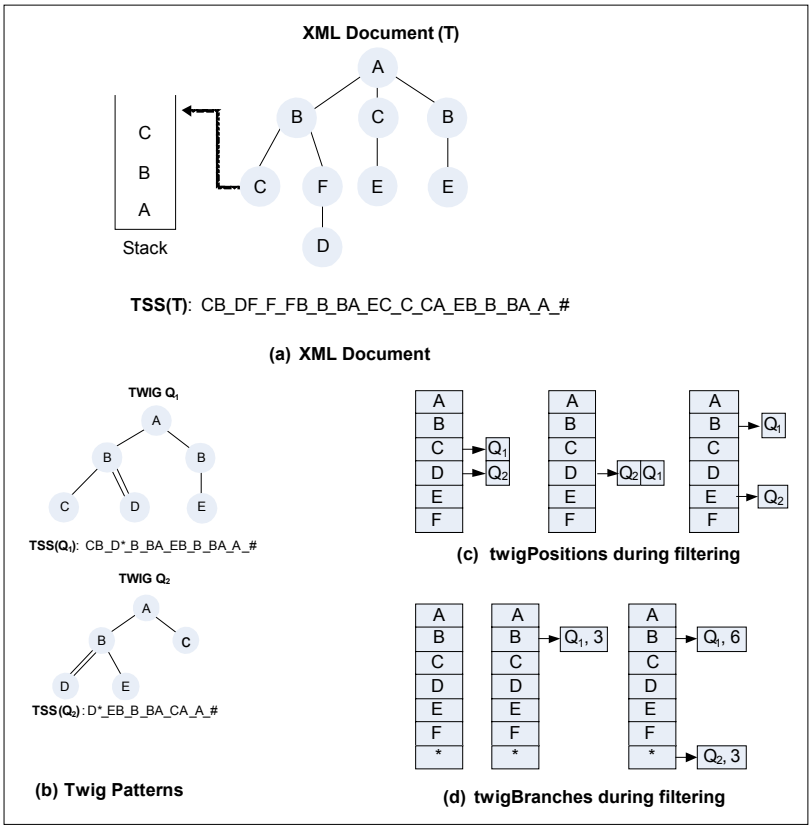
In Case 2, the current positions of the twig and the document TSSs must be incremented by 3, because after the current matched pair of char labels, there is a space character by default.

In Case 3 the matching refers to an ancestor-descendant relationship in a twig's TSS. The character '*' in the twig's TSS can be matched with any character in the document's TSS. In this case, the current positions of twig and document TSSs are incremented by 3.

Because, in every step, the current and the next char label of the document TSS are compared with more than one twig TSS, the current position of the document TSS is incremented at the end of each step, when all corresponding twig TSSs have been compared.

It is important to realise that in every step, the increment in the document TSS position depends only on *DN* (if the character is a white space, then the position is incremented by 2, otherwise by 3) because of the particular structure of the TSS. This means that, in every step, even if no twig TSS is matched, the document TSS's position is incremented as mentioned above. The filtering process is finished when the whole document's TSS has been parsed. At this point, the algorithm checks the positions of all twig TSSs and reports those twigs whose TSSs have been passed until their final char label.

Figure 5 Filtering algorithm example



The algorithm mentioned above may report some *false matches* because no branch node checking has been done and hence we must add some additional checks in order to avoid false matches of branching nodes.

It is important to realise that the second char label of a pair (*e.g.*, the char *B* in the pair *EB*) denotes the father of the first char label. This father node is called a *branch node*. Hence, when a node has two or more children in the document or the twig, the

corresponding TSS has two or more sequential pairs of char labels followed by a single char (the char label of the father node), with every pair having the same second char label (the father node's char label). Those pairs of char labels and the following single char label need to be matched as a unit in the matching algorithm, either all of them or none of them. The described algorithm matches a pair of char labels each time, ignoring the following pairs of char labels. This may lead to matching some first edges of a branch node (represented by the first pairs of char labels considering this node) but not all of them. In this case, the algorithm should identify the situation and backtrack the pointers of the document's and twig's TSS to point again at the first pair of char labels considering this branch node.

2.6 The branching verification method

The technique we use to avoid false branch node matchings is to keep track of the branch node to be verified for every twig at every step of the algorithm. The structure we use is a hash table named *twigBranches.* This hash table, which is being updated in every step of the filtering process, uses char labels as keys, and for each key it returns a list with the twigs that need to match a branch node with the specific tag at the current step of the process. For each twig to be matched, extra information of the offset in the twig TSS is kept, so that in case the branch node does not match, we can adjust the twig TSS's position (decrement it by the current offset) accordingly.

To demonstrate the branch node verification, let us consider the twig TSS part TQ:…*EB_CB_B_BA_*… and two parts of the document TSS: D1:…*EB_DB_B_BA*… and D2:…*EB_CB_B_BA*…. When the algorithm matches the pair *EB* of TQ with the corresponding pair of D1, it inserts into *twigBranches* a record for twig Q with branch label B and offset 3. This means that Q must match a branch node B, and if not, the position of Q's TSS must be decremented by 3. Next the algorithm tries to match pair *CB* (of TQ) with the pair *DB* (of D1) and fails. The next char label of D1's TSS is char label B. The algorithm searches *twigBranches* for twigs to match branch node B and finds twig Q. However, Q's TSS position does not point to a char label B and so the match fails. The algorithm decrements Q's TSS position by 3 to point to the pair *EB*.

The method we use is to maintain the global hash table *twigBranches*, which keeps track of the branch nodes to be matched for every twig. In addition to this, the structure kept in this hash table consists not only of the twig ID but additionally of an integer offset identifying the offset that the twig TSS's position must decrement, if the twig does not match the corresponding branch node. In this way, when the algorithm identifies that a twig failed to match a branch node, it immediately decrements the corresponding TSS's position in order to point to the previous right position.

Because of the TSS's structure, a twig enters in a *branch-node matching area* when it matches a pair of char labels. The second char label of the pair is the candidate branch node that the twig has to match. While the twig continues to match sequential pairs of labels (representing outgoing edges of the branch node to be matched), the current twig's offset is incremented analogously (increases by 3 at a match).

A special case occurs when the second label of the pair is '*'. This represents an ancestor-descendant edge and special handling is needed. In this case the label of the branch node to be verified is yet unknown and for this reason, *twigBranches* has the special key '*'. The entries under this key represent twigs that wait to match a yet unknown branch node. Those entries are *temporary*, which means that in some next step

those entries are moved under a new key. There are two conditions for a twig to be removed from the key *: Firstly, if it matches a pair of char labels with the second label different from *. In this case the offset field is updated accordingly and the twig's entry in *twigBranches* is moved under the right key, which is the second char label of the matched pair. Secondly, if the twig matches a single node, the corresponding entry of the twig in *twigBranches* is totally removed because the twig has left the *branch-node matching area* and has matched the branch node.

In order for a twig to leave the *branch-node matching area*, there are two cases:

Case 1 The document's TSS reaches a single character label (which is the label of the branch node to be matched) while the twig's TSS points to a pair of character labels. This situation means that the corresponding document node does not include the edge represented by the twig's TSS, and so there is no matching. The twig TSS's position must be decremented by the corresponding offset. Additionally, the twig's branch node information is deleted because the twig has left the *branch-node matching area*. It will enter it again, if in some next step it matches a pair of character labels.

Case 2 Both the document's TSS and the twig's TSS point to a single character label (representing the branch node to be matched). In this situation we have a match, and as a result the twig TSS's position is incremented by 2 to point to the next character. Additionally, the branch node information for this twig is deleted because the twig has left the *branch-node matching area*.

By incorporating this method into our filtering algorithm, we avoid the extra cost of finding false matches and then applying a refinement phase to identify those false matches. This is a major difference with previous proposed solutions, and results in a significant reduction in time complexity of our filtering method.

The details along with the pseudo codes of filtering algorithm implementation are presented in the Appendix.

In order to better illustrate the above described filtering algorithm, let us consider the following example:

Example 1 *Consider the XML Document T and twigs Q_1 and Q_2 shown in Figure 5. At first the twigs are stored in the system and their TSSs are computed. When the XML document arrives, the SAX parser starts and its handlers are invoked as needed. tagStack contents are shown in Figure 5(a). At this time the twigPositions contains two entries, twig Q_1 under key C and twig Q_2 under key D as shown in Figure 5(c), while twigBranches is empty as shown in Figure 5(d). Node C is a leaf node and so the top element of tagStack (label C) is popped. After this, the function addEdgeNode() is called with arguments C, B (the top element of tagStack). The twig list in the twigPointers for key C is twig Q_1. The current pair of labels of the document's and Q_1's TSS is CB and they match. This match results in incrementing by 3 the positions of both T's TSS and Q_1's TSS. Additionally, the twigPositions hash table is updated and Q_1 is moved under key D (which is the char label in the Q_1 TSS's current position). Next, endTag(D) is invoked and the handler calls addEdgeNode(D, F). The twig list in twigPointers for key D is Q_2 and Q_1. At first, Q_2 is checked.*

The Q_2 TSS's current position points to the pair D while the document TSS's current position points to pair DF. The final action of the function is to update the twigBranches because Q_1 has matched a pair of char labels and has entered a branch-node matching area. Twig Q_1 is inserted in twigBranches under key B (which is the second label of the matched pair of labels) with offset 3. The TSSs match (ancestor-descendant match) and the position of Q_2's TSS is incremented by 3. Additionally, twigPointers is updated and Q_2 is moved under key E.*

*Finally, because the match concerns a pair of labels, Q_2 has entered a branch-node matching area and Q_2 is inserted in twigBranches under key * (a temporary entry) and offset 2. The next twig in the list is Q_1. The corresponding TSS's position points to the pair D* and because document TSS's current pair is DF, we have a match. The Q_1 TSS's position is incremented by 3 and twigPointers is updated by moving Q_1 under key B. Additionally, twigBranches is updated as follows: Because Q_1 is already in twigBranches under key B (which means that it waits to match a branch node with label B) and the current pair's second label is *(ancestor-descendant match), the only action needed is to increase the offset of Q_1's entry in twigBranches. The offset is incremented by 3 and equals 6.*

2.7 Time complexity

Let us assume that the XML document consists of n nodes and m edges and the number of stored user profiles is k. As shown before, the document's TSS has a length of at most $O(m+n)$. XFIS requires only one pass of the document's TSS in order to filter it against the stored user profiles. In every step of the filtering process, the current element of the document's TSS is checked only against respective user profiles whose current TSS element equals the document's current element. In the worst case, each element of the document's TSS is checked against $O(k)$ elements (one for every user profile). Additionally, in each step, extra checks for identifying false branch node matches are performed, which in the worst case costs $O(k)$. Summing up, the total time complexity of XFIS, in a worst-case scenario, is $O(k(m+n))$. As can be easily seen, the total runtime depends on the size of the incoming XML document and the number of stored user profiles, but is independent of the size of stored user profiles. However, it should be mentioned that in the average case, each element of the document's TSS is checked only against a small portion of user profiles, which radically reduces the total runtime of XFIS.

3 Experiments

In our experiments, we compared XFIS with the FiST algorithm (Kwon *et al*., 2005), which is the state-of-the-art algorithm for filtering XML documents against twig-pattern user profiles. We chose FiST because it supports twig-pattern user profiles, unlike other systems (*e.g.*, AFilter, XFilter), which support only linear path expressions. XFIS was implemented in Java using the freeware Eclipse IDE and the Xerces XML parser (Apache).[2] In order to obtain comparable and reliable results, we also implemented a

FiST-like algorithm in Java using Eclipse. We ran all our experiments on a Mobile Pentium 2.0 GHz machine with 512 MB RAM running Windows XP SP2. XFIS code and FiST-like code were run using Eclipse 3.0.1 with Java Virtual Machine 1.4.2.

3.1 XML data sets and twig patterns

In our experiments we used data on Shakespeare's plays, provided in *The Plays of Shakespeare* in XML.[3] We used the DTD of Shakespeare's plays to generate 300 documents of different sizes and depth. The generated documents were categorised into three categories according to the total number of nodes in each document, without taking into consideration the document's depth. The corresponding categories were 0–2000, 2000–4000 and 4000–6000 nodes, indicated by '2000', '4000' and '6000'.

In order to generate a large number of different twig patterns, we used the XPath generator provided in the YFilter package. The number of branches of the generated twig patterns was 2, 4 and 6, resulting in a corresponding categorisation. Finally, the number of twig patterns stored as user profiles in both the filtering systems varied between 5000 and 20 000 in steps of 5000.

The experiments were performed for every combination of the document's number of nodes, number of branches per twig pattern and total number of twig patterns stored. Table 2 summarises the parameters of our experiments.

Table 2 Experimental parameters

Parameter	*Values*
Number of twig patterns	2000, 4000, 6000
Number of branches per twig pattern	2, 4, 6
Number of XML document's nodes	2000, 4000, 6000

3.2 Performance analysis

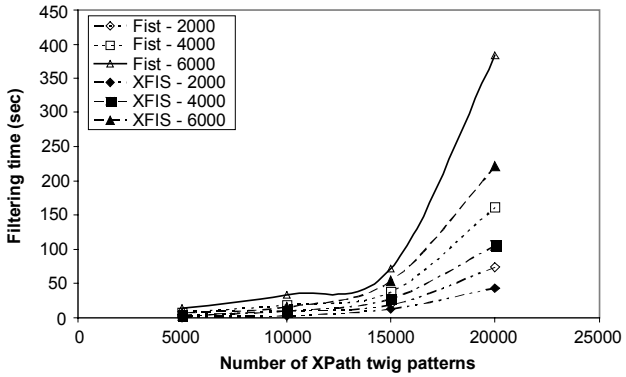
We measured the filtering time for a variety of document sizes and twig patterns. Additionally, we measured the total disk space required by each algorithm for storing a variety of twig patterns. As a general conclusion and concerning filtering time, both algorithms behave the same way in all cases, but XFIS is obviously faster (about a 25% factor). In both algorithms, the filtering time increases as the sizes of the XML document and the number of twig profiles increase, whereas it decreases when the number of average branches per twig increases. As far as disk space is concerned, XFIS was measured to require less disk space than FiST, independent of the number of branches per twig. The results show that XFIS outperforms FiST in all cases due to the elimination of the *branch node refinement* phase and the encoding of all the node relationships and document structures in TSS.

In the succeeding discussion we present in detail the above-mentioned experimental results.

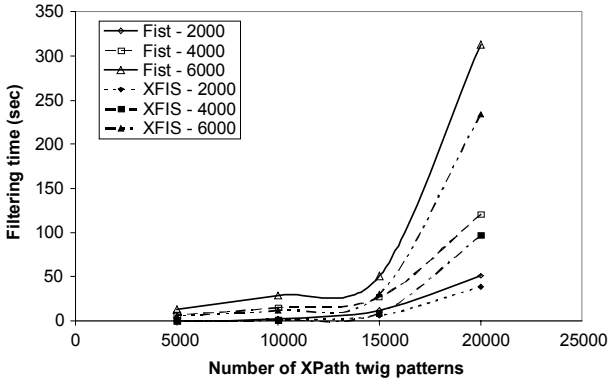
3.2.1 Varying number of twig patterns

We measured the filtering time required by XFIS and FiST for a varying number of twigs between 5000 and 20 000 in steps of 5000. The results are presented in Figure 6 for number of branches 2, 4 and 6 per twig, and number of XML document nodes 2000, 4000 and 6000.

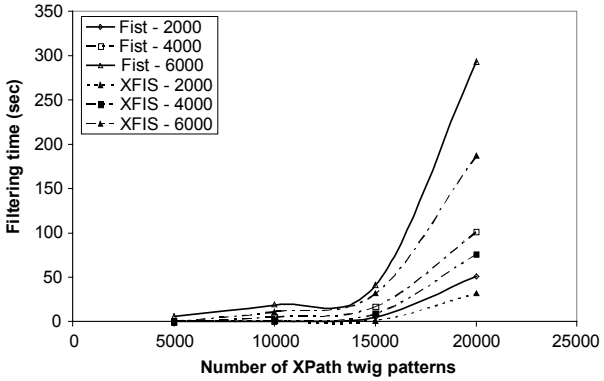
Figure 6 Varying number of twig patterns



(a) Number of branches – 2



(b) Number of branches – 4



(c) Number of branches – 6

In Figure 6(b) the number of branches was 4 and each line in the plot corresponds to a combination of an algorithm (XFIS or FiST) and the number of XML document's nodes (2000, 4000 or 6000). For example, the line labelled 'XFIS-4000' corresponds to the XFIS method and 4000 nodes in the XML document. In this figure we observe that the filtering time for both methods increases as the number of twig patterns stored

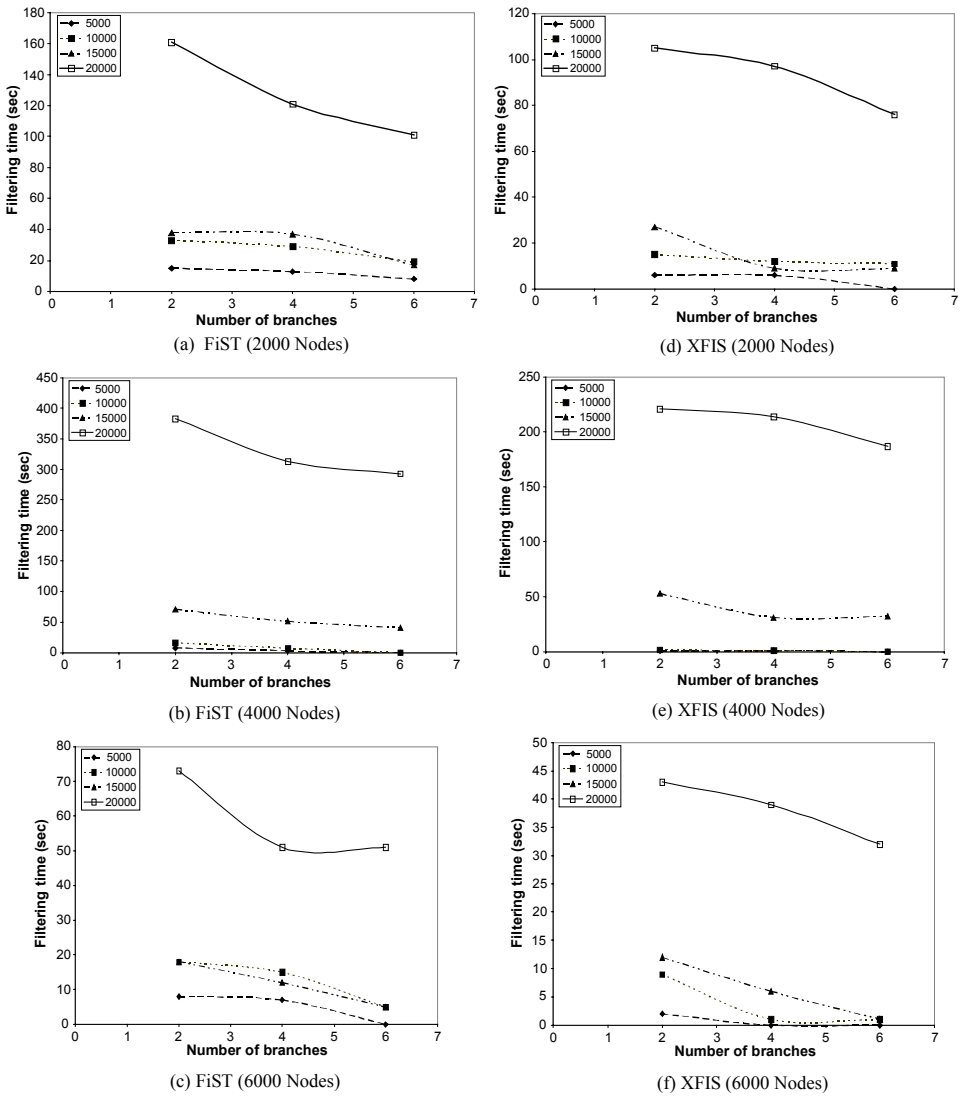
increases. Both methods behave in the same way, but XFIS is obviously faster in all cases. For data sets with 2000 and 4000 document nodes, XFIS is about 25% faster. In the case of 6000 document nodes, we observe that the filtering time increases suddenly for both methods. This is due to our system's limitations in memory and usage of disk swap space instead. However, even in this case XFIS performs better, 35% faster than FiST. Similar observations can been made in Figures 6(a) and 6(c).

The above results show that XFIS performs better than FiST as the total number of twig patterns increases.

3.2.2 Varying number of branches per twig pattern

We measured the filtering time required by XFIS and FiST for a varying number of branches per twig – 2, 4 and 6. The results are presented in Figure 7.

Figure 7 Varying number of branches



(a) FiST (2000 Nodes)

(d) XFIS (2000 Nodes)

(b) FiST (4000 Nodes)

(e) XFIS (4000 Nodes)

(c) FiST (6000 Nodes)

(f) XFIS (6000 Nodes)

In Figures 7(a) and 7(d) the number of branches is 4. The results in Figure 7(a) correspond to the FiST method and 2000 document nodes, while in Figure 7(c) they correspond to the XFIS method and 2000 document nodes. Each line in the plots corresponds to a number of stored twig patterns. For example, the line labelled '150 000' corresponds to 150 000 stored twig patterns. As can be observed, the total filtering time in both methods decreases as the number of branches increases. This may seem strange, but it is due to the fact that the number of matching twig patterns decreases as the number of branches increases, because of the twig's complexity. Both methods behave the same way, but XFIS performs better in all cases. For example, for 5000 and 10 000 twig patterns, XFIS is about 28% faster; while for 15 000 and 20 000 twig patterns it performs about 34% better.

Similar observations can be made in all figures through Figures 7(a) to 7(f) for different document sizes. These results demonstrate that XFIS performs better as the number of branches per twig pattern increases, due to the elimination of the *branch node refinement* phase used in FiST.

3.2.3 Varying number of XML document nodes

We measured the filtering time required by XFIS and FiST for a varying number of XML document nodes (2000, 4000 and 6000). The results are shown in Figure 8.

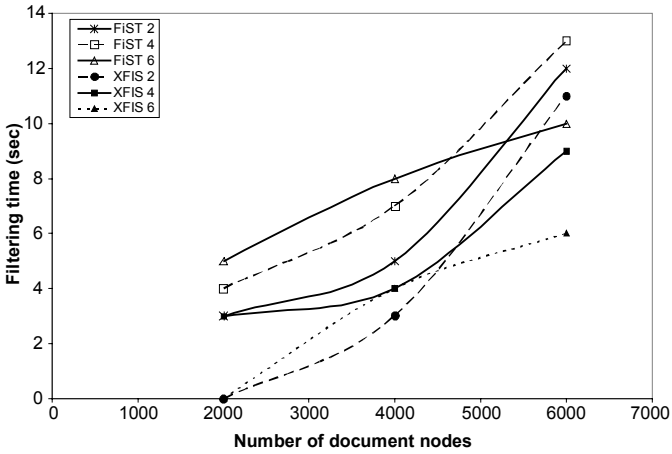
Each line in the plots corresponds to a combination of method and number of branches per twig. For example, the line labelled 'XFIS 6' corresponds to the XFIS method with six branches per twig. As can be seen, the filtering time in both methods increases as the total number of XML document nodes increases.

This observation is expected, as both TSS and Prüfer sequences of the XML document increase in length, requiring more time to be parsed. However, XFIS again performs better in all combinations of the total number of XML document nodes and the number of twig branches. For example, let us consider Figure 8(b), which presents the results for 10 000 twig patterns. As we can observe, for 2 and 4 branches per twig pattern, XFIS performs about 23% better than FiST. Additionally, this factor grows as the number of XML document nodes increases. In the case of 6 branches per twig, XFIS performs about 32% better than FiST, with this factor growing as the number of XML document nodes increases. The same trend can be observed in all three plots of Figure 8. These results show that, although they have the same behaviour, XFIS performs better than FiST as the number of XML document nodes increases.

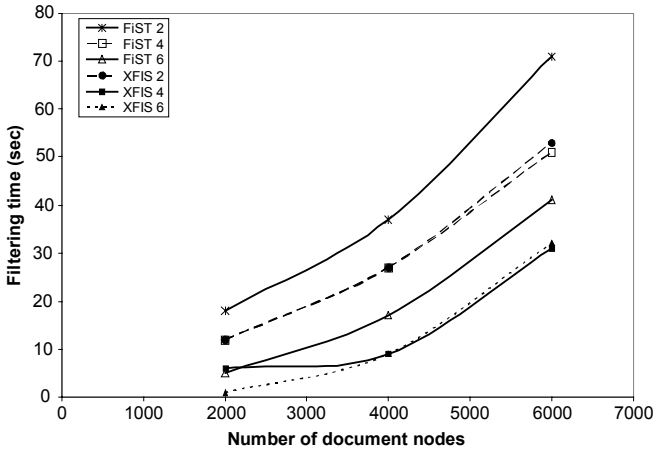
3.2.4 Required disk space

In this section we present the results of measuring the total disk space required by each method for storing all the information needed for the corresponding twig patterns. We ran our experiments for a varying number of twig patterns between 5000 and 20 000 in steps of 5000 and a varying number of branches per twig pattern. The results of our experiments are presented in Figure 9. Each plot in Figure 9 presents the results for different numbers of branches per twig pattern. Let us consider Figure 9(b). As can be seen, the required disk space of both XFIS and FiST increases as the total number of twig patterns increases, as expected.

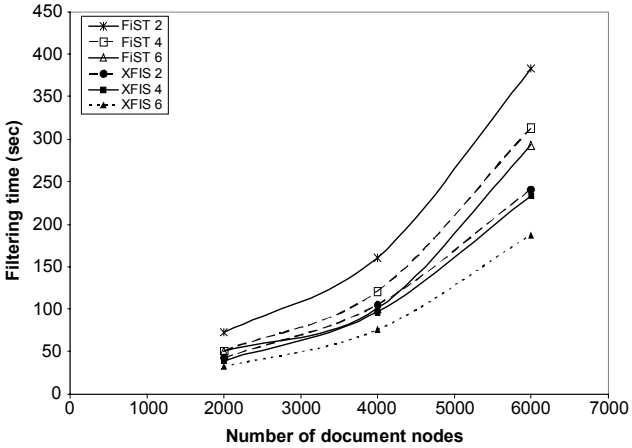
Figure 8 Varying size of XML document



(a) 5000 twig patterns

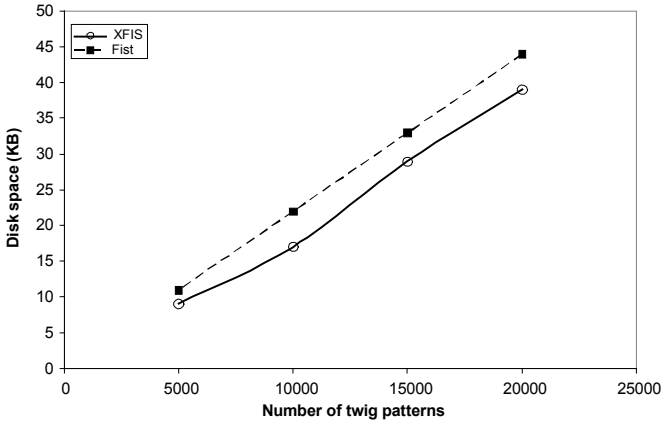


(b) 10 000 twig patterns

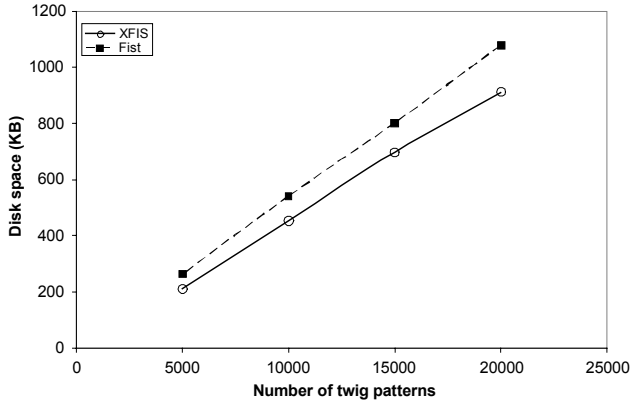


(c) 20 000 twig patterns

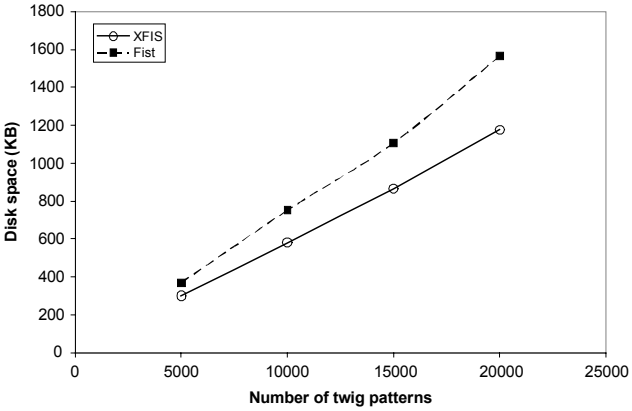
Figure 9 Required disk space for a varying number of twig patterns



(a) Number of branches – 2



(b) Number of branches – 4



(c) Number of branches – 6

This trend stands for both methods, XFIS and FiST, independent of the number of twig patterns. However, XFIS requires less disk space than FiST in all cases. Another observation is that the gap between FiST's space and XFIS's space increases as the number of branches per twig pattern increases.

In Figure 9(b), which corresponds to four branches per twig pattern, XFIS requires about 20% less space than FiST. In Figure 9(c), which corresponds to six branches per twig pattern, XFIS requires about 30% less space than FiST. This is due to the fact that FiST uses an additional structure for each twig pattern, called *ProfileSequence*, to store information about the relationships between a twig pattern's nodes. When a node is a branch node, extra information is stored in the *ProfileSequence* to indicate that this node is a branch node. As a result, the total required space by FiST grows faster when the number of branches per twig pattern increases.

4 Practical applications

With the wide penetration of the internet, XML has become a *de facto* standard of data representation and exchange. The number of applications using XML data representation is growing rapidly, thus the process of XML filtering is becoming an essential need of an increasing number of different application areas, such as publish/subscribe systems, peer-to-peer networks and web services.

Publish/subscribe systems grow rapidly, targeting many areas such as real estate sales, electronic personalised newspapers/advertisements and sensor-driven services. A publish/subscribe system is a middleware implementing the event-based communication paradigm: A publisher publishes event messages that announce the occurrence of events. Subscribers can subscribe to events that are of interest to them, called profiles. The system filters the incoming messages according to the profiles and forwards matched messages to their subscribers. Recently, XML-based messages or documents have been used to encode the event messages. Applications are e-businesses such as online catalogues or digital libraries. Such systems require online filtering of the incoming XML documents/messages against the user profiles. XFIS fits the above-mentioned needs of publish/subscribe systems, providing efficient, online and ordered filtering of incoming XML documents.

Peer-to-Peer (P2P) networks are typically used for connecting nodes via largely *ad hoc* connections. Such networks are useful for many purposes. Sharing content files containing audio, video, data or anything in digital format is very common. A pure P2P network does not have notion of clients or servers, but only equal *peer* nodes that simultaneously function as both 'clients' and 'servers' to the other nodes on the network. A peer can publish (advertise) its available services and resources as well as its interests in the P2P network. The type of peer advertisements vary between the different P2P networks, thus making them incompatible with each other. Recently, Sun Microsystems has developed JXTA,[4] an open-source programming and computing platform to ease the development of P2P networking. One of the most interesting characteristics of JXTA is that it utilises XML formatting for describing peer advertisements. XML is used to describe a peer's available services and resources, while XPath is used to describe a peer's interests. Thus, the process of filtering available resources against a peer's interests

is reduced to the XML filtering problem. In a P2P network the ability for online filtering of incoming documents is vital, thus every adopted filtering algorithm should be able to filter XML documents online. Additionally, due to the large number of peers, the filtering algorithm should scale well as the number of user profiles increases. XFIS fulfils both of the above-mentioned characteristics and can be efficiently used in such a P2P network.

5 Conclusion

In this paper, we have presented an innovative filtering system called XFIS. XFIS utilises a new string representation for tree structures, based on the Prüfer sequence, called Tree-Structure Sequence (TSS). TSS encodes all the nodes and structure of an XML document in a single string, without the need for additional structures to store information. XML twig patterns, representing user profiles, and XML documents are transformed into TSSs, and XFIS involves a progressive subsequence string matching to identify those twig patterns that match the XML document. XFIS is able to handle order matching of user profiles, a property that is needed in quite a few applications.

Our experimental results showed that XFIS outperforms the previous algorithms in XML filtering both in space and time aspects.

Acknowledgement

Panagiotis Antonellis's work was supported in part by a Bodossaki Foundation scholarship.

References

Aguilera, M.K., Strom, R.E., Stunnan, D.C., Astley, M. and Chandra, T.D. (1999) 'Matching events in a content-based subscription system', *PODC*, pp.53–61.

Altinel, M. and Franklin, M.I.J. (2000) 'Efficient filtering of XML documents for selective dissemination of information', *VLDB*, pp.53–64.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J. and Simon, J. (2002) 'XML path language (XPath) 2.0', W3C working draft 16, Technical Report WD-xpath20-20020816, World Wide Web Consortium, August.

Canadan, K., Hsiung, W., Chen, S., Tatemura, J. and Agrawal, D. (2006) 'AFilter: adaptable XML filtering with prefix-caching and suffix-clustering', *VLDB*, pp.579–590.

Carzanica, A., Rosenblum, D. and Wolf, A. (2001) 'Design and evaluation of a wide-area event notification service', *ACM Transactions on Computer Systems*, August, Vol. 19, No. 3, pp.332–383.

Chamberlin, D. (2002) 'XQuery: an XML query language', *IBM Systems Journal*, Vol. 41, No. 4, pp.597–615.

Chan, C.Y., Felber, P.L., Garofalakis, M.N. and Rastogi, R. (2002) 'Efficient filtering of XML documents with XPath expressions', *The VLDB Journal*, Vol. 11, pp.354–379.

Diao, Y., Altinel, M., Franklin, M.L.J., Zhang, H. and Fischer, P. (2003) 'Path sharing and predicate evaluation for high-performance XML filtering', *TODS*, Vol. 28, pp.467–516.

Gupta, A.K. and Suciu, D. (2003) 'Stream processing of XPath queries with predicates', *SIGMOD*, pp.419–430.

Kwon, J., Rao, P., Moon, B. and Lee, S. (2005) 'FiST: scalable XML document filtering by sequencing twig patterns', *VLDB*, pp.217–228.

Peng, F. and Chawathe, S. (2005) 'XSQ: a streaming XPath engine', *TODS*, Vol. 30, June, pp.577–623.

Prüfer, H. (1918) 'Neuer Beweis eines Satzes uber Permutationen', *Archiv für Mathematik und Physik 27*, pp.142–144.

Tian, F., Reinwald, B., Pirahesh, H., Mayr, T. and Myllymaki, J. (2004) 'Implementing a scalable XML publish/subscribe system using a relational database system', *SIGMOD*, pp.479–490.

Uchiyama, H., Onizuka, M. and Honishi, T. (2005) 'Distributed XML stream filtering with high scalability', *ICDE*, pp.968–977.

Notes

1 XQL, http://www.xml.com/pub/a/SeyboldReport/ipx981101.html.

2 Apache, 'Apache Xerces Java Parser', http://xml.apache.-org/xerces-j/.

3 The Plays of Shakespeare in XML, http://xml.coverpages.org/bosakShakespeare200.html.

4 JXTA, 'JXTA Project Home', http://www.jxta.org.

Appendix

In this appendix we present in detail the pseudo codes of the XFIS filtering algorithm implementation. The detailed pseudo codes can be seen in Figure 1.

The function *addEdgeNode()* tries to match a pair of char labels, representing an edge, with the twigs' TSSs. At first, the function locates those twigs whose position points to the first label of the pair. For each of those twigs, it checks whether the next label matches the second label of the pair. If so, the twig's pointer is incremented by 3 (to involve the space character), in order to point to the next character of the TSS, and the *twigPositions* is updated accordingly. In addition, because now the branch node is *plabel*, the function updates the branch information of this twig, by invoking the *updateBranchInfo()* function, which we will describe later.

The function *addCurrentNode()* tries to match a single char label, representing an internal node, with the twig's TSS. At first, the function finds those twigs whose position points to the specific label and for each of those twigs, it checks whether the next label matches the space character ('_'). If so, the twig's pointer is incremented by 2 (to involve the space character), in order to point at the next character of the TSS, and the *twigPositions* is updated accordingly by calling the *updateTwigPosition()* function. In addition, the function removes the branch information of this twig, as it has matched its current branch node, by calling the *removeBranchInfo*() function. After matching all corresponding twigs, the function accesses *twigBranches* to find all those twigs that still need to match a branch with the current char label. It is obvious that all the twigs that have matched this branch node in the previous step have been removed from the *twigBranches* by the function *removeBranchInfo*() that was called previously. This means that all of the twigs found in the *twigBranches* with the current label as key, failed to match the current branch node; and as a result their TSS's position must be decremented by the corresponding offset. This action is performed by invoking the function *correctBranchPos*().

Function *addRootNode()* is invoked by the SAX parser when the end of the root's node tag is reached. This means that the whole XML document has been parsed and it is time to report those twigs that have a match with the document. Matched twigs are those twigs that in the current step wait to match the character '#', which means that their TSS's position is at the end. The function accesses *twigPositions* to find the above-mentioned twigs and returns them as matched twigs. The filtering process is then finished.

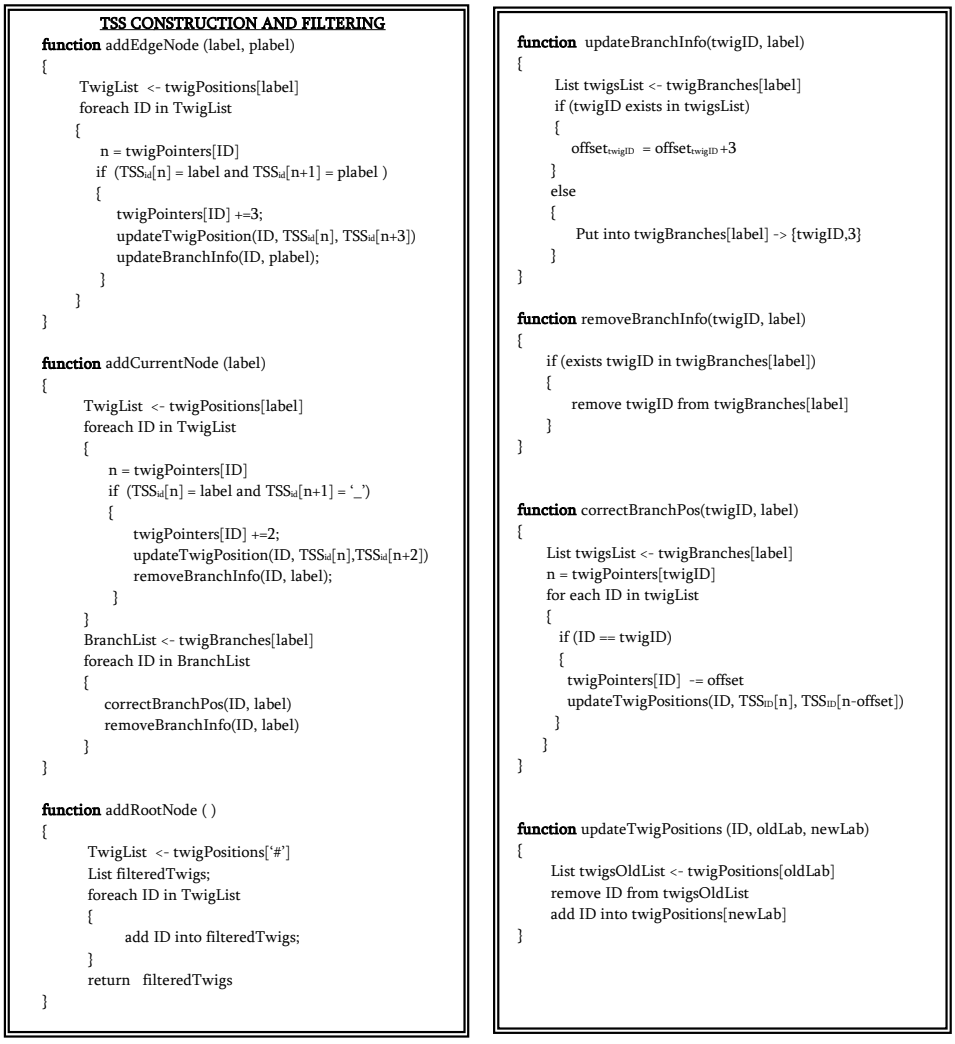
All the above-mentioned functions invoke the *updateTwigPositions()* function. This function is used to update the hash table *twigPositions* when a twig TSS's position is changed. It deletes the old entry for the current twig and adds a new entry with the current char label as key pointed from the twig's TSS. This function is invoked whenever a twig TSS's position changes to point to a new char label.

The function which updates the branch node information of the twigs is *updateBranchInfo*(). This function takes as arguments the twig ID and the char label of the branch node to be matched. It checks whether there is an entry in the *twigBranches* for the specific twig and branch node. If not, it inserts into *twigBranches* a corresponding entry with offset 3. If there is already an entry for this twig and branch node (*i.e.*, the twig has already entered the *branch-node matching area*), it increments the offset of this entry by 3.

The function which deletes the branch node information of a twig (when it leaves the *branch-node matching area)* is the *removeBranchInfo*(). This function simply finds and removes the entry in *twigBranches* for the corresponding twig and char label.

The final function for branching node verification is *correctBranchPos()*. This function is invoked for each of the twigs that failed to match a specific branch node. Its purpose is to correct the corresponding twigs' TSS positions by decrementing them by the corresponding offset. Its argument is the ID of the twig to be corrected and the char label of the corresponding branch node. The function accesses *twigBranches* using as key the branch node's label to find the entry involving the current twig. It reads the entry's current offset and decrements the twig TSS's position by this offset.

Figure 1 Pseudo codes of filtering algorithm implementation



```
TSS CONSTRUCTION AND FILTERING
function addEdgeNode (label, plabel)
{
    TwigList <- twigPositions[label]
    foreach ID in TwigList
    {
        n = twigPointers[ID]
        if (TSSid[n] = label and TSSid[n+1] = plabel )
        {
            twigPointers[ID] +=3;
            updateTwigPosition(ID, TSSid[n], TSSid[n+3])
            updateBranchInfo(ID, plabel);
        }
    }
}

function addCurrentNode (label)
{
    TwigList <- twigPositions[label]
    foreach ID in TwigList
    {
        n = twigPointers[ID]
        if (TSSid[n] = label and TSSid[n+1] = '_')
        {
            twigPointers[ID] +=2;
            updateTwigPosition(ID, TSSid[n],TSSid[n+2])
            removeBranchInfo(ID, label);
        }
    }
    BranchList <- twigBranches[label]
    foreach ID in BranchList
    {
        correctBranchPos(ID, label)
        removeBranchInfo(ID, label)
    }
}

function addRootNode ( )
{
    TwigList <- twigPositions['#']
    List filteredTwigs;
    foreach ID in TwigList
    {
        add ID into filteredTwigs;
    }
    return filteredTwigs
}

function updateBranchInfo(twigID, label)
{
    List twigsList <- twigBranches[label]
    if (twigID exists in twigsList)
    {
        offset_twigID = offset_twigID +3
    }
    else
    {
        Put into twigBranches[label] -> {twigID,3}
    }
}

function removeBranchInfo(twigID, label)
{
    if (exists twigID in twigBranches[label])
    {
        remove twigID from twigBranches[label]
    }
}

function correctBranchPos(twigID, label)
{
    List twigsList <- twigBranches[label]
    n = twigPointers[twigID]
    for each ID in twigList
    {
        if (ID == twigID)
        {
            twigPointers[ID] -= offset
            updateTwigPositions(ID, TSSID[n], TSSID[n-offset])
        }
    }
}

function updateTwigPositions (ID, oldLab, newLab)
{
    List twigsOldList <- twigPositions[oldLab]
    remove ID from twigsOldList
    add ID into twigPositions[newLab]
}
```