Approximate Structural Matching over Ordered XML Documents

Nitin Agarwal
Arizona State University
Nitin.Agarwal.2@asu.edu

Magdiel Galan Oliveras
Arizona State University
Magdiel.Galan@asu.edu

Yi Chen
Arizona State University
yi@asu.edu

Abstract

There is an increasing need for an XML query engine that not only searches for exact matches to a query but also returns "query-like" structures. We have designed and developed XFinder, an efficient top K tree pattern query evaluation system, which reduces the problem of approximate tree structural matching to a simpler problem of subsequence matching. However, since not all subsequences correspond to valid tree structures, it is expensive to enumerate common subsequences between XML data and query and then filter the invalid ones. XFinder addresses this challenge by detecting and pruning structurally irrelevant subsequence matches as early as possible. Experiments show the efficiency of XFinder on various data and query sets.

1 Introduction

Wide acceptance of XML as the standard data exchange format has led to a large amount of XML data that needs to be searched. XML documents are in general viewed as trees, as in Figure 1. The key component in XML query languages (e.g. XPath and XQuery) is tree pattern queries (twig queries).

For document-oriented XML data, such as Shakespeare's plays[1], legislative documents[2], and news in XML format[3], the order of sections, paragraphs, and sentences is important. For instance, we may express a query like "find all the acts before the act that is titled as ACT IV and has a speaker Philo" when searching Shakespeare's plays. This query can be represented as the query tree in Figure 1, if we set the tag A to "Play", B and D to "Act", G to "Title" with a value predicate, and F to "Speaker" with a value predicate.

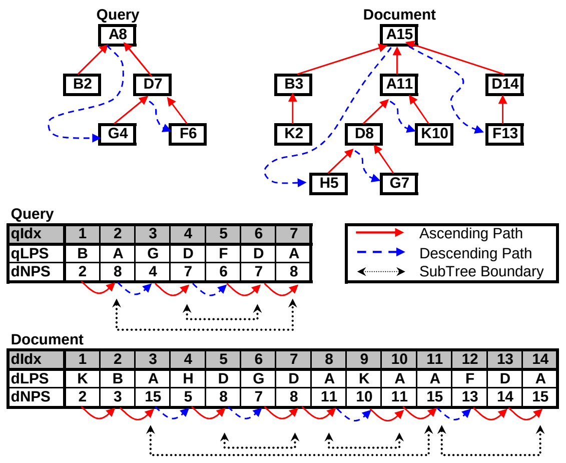
[1]http://www.ibiblio.org/xml/examples/shakespeare/
[2]http://xml.house.gov/
[3]http://www.xmlnews.net/

During a search, a user query may be over-specified. For example, no acts in Shakespeare's plays satisfy the above query. Returning an empty query answer can be frustrating to users. Furthermore, sometimes the user may only have a rough idea of what (s)he is looking for. It is desirable if the top K XML subtrees that *exactly* or *partially* match the query tree are returned, in the order of the degree of matching.

In this paper, we propose XFinder, a system that efficiently searches top K XML subtrees that exactly or partially match input ordered tree pattern query in ranked order. Rank of the match is gauged by the degree of tree structure match, as well as the degree of corresponding node tag and node value match. Existing approaches [9, 3, 6] can be plugged in XFinder to measure similarity of tags and values. In this paper, we focus discussion on approximate structural matching between ordered XML data and queries, the unique challenge in XML data processing.

Although we can compute approximate tree pattern matches directly [13], such an approach involves high complexity. Alternatively, we can relax the input query such that the data subtrees satisfying the relaxed queries are returned [16]. However, in general the size of relaxed queries can be exponential to the input query.

On the other hand, the problem of *exact* ordered tree pattern matching can be reduced to that of subsequence matching [12]. Using Prüfer method that constructs a one-to-one correspondence between trees and sequences, both XML data tree and query tree are transformed into sequences, and the document subsequences that are the same as the query sequence are computed and filtered as the result of evaluating a twig query. There are two advantages of such an approach. First, it allows holistic processing of a twig query to achieve efficiency. Indeed, the approach that decomposes a twig query to subqueries and processes each subquery individually can result in large intermediate results, and therefore can be expensive. Second, sequence matching is inherently simpler than tree pattern



Query

qIdx	1	2	3	4	5	6	7
qLPS	B	A	G	D	F	D	A
dNPS	2	8	4	7	6	7	8

Document

dIdx	1	2	3	4	5	6	7	8	9	10	11	12	13	14
dLPS	K	B	A	H	D	G	D	A	K	A	A	F	D	A
dNPS	2	3	15	5	8	7	8	11	10	11	15	13	14	15

Figure 1. Sample XML Document and Query with their Prüfer Sequences

matching, which enables various optimizations. The efficiency of a sequence-based tree pattern matching approach has been demonstrated in PRIX [12].

Intuitively, we can extend this technique to retrieve top K document subsequences that have largest length in common to query sequence, such that *partial matches* are returned at same time as exact matches.

However, such an intuitive approach can be inefficient due to a possibly large number of common subsequences that do not correspond to valid tree structures. Recall that after performing sequence matching, PRIX [12] requires a filtering step on each matched document subsequence to ensure that it forms a tree structure and this structure matches query tree. Therefore to retrieve top K valid subtree matches, the matching step needs to return $K'(K' \geq K)$ longest common subsequences, since some subsequences need to be filtered after matching step.

Example 1.1: In Figure 1, top 3 longest document subsequences that match query sequence are $B3$-$A15$-$G7$-$D8$-$F13$-$D14$-$A15$, $B3$-$A15$-$D8$-$F13$-$D14$-$A15$, and $B3$-$A15$-$F13$-$D14$-$A15$. However, the first two matched subsequences do not correspond to valid trees, and therefore are rejected in the filtering step. We only obtain the third subsequence as a valid match. In this example, to find the top $K = 1$ match we have to generate top $K' = 3$ common subsequences. ■

As we can see, the relationship between K and K' depends on the particular structure of document and query, therefore we are not able to set the value of K' a priori. We need to either set K' to be very large to guarantee that after filtering stage at least K valid sequences are returned; or we need to perform several document passes in matching stage if K' is relatively small and less than K common subsequences are valid. Neither approach is efficient.

To address afore-mentioned challenges, we developed XFinder and make the following technical contributions:

- We propose a novel approximate tree structural matching algorithm by reducing the problem to longest common Prüfer subsequence matching.
- By pruning matching subsequences that do not correspond to valid tree structures as early as possible, and discarding the common subsequences that are not top K matches immediately, we avoid to generate large intermediate query results and achieve efficiency.
- We introduced a ranking function such that similarity measurement between two trees can be obtained by similarity measurement between their Prüfer sequences.
- We developed XFinder system for approximate XML tree pattern query evaluation and demonstrated its effectiveness in experiments.

Next we introduce background knowledge about Prüfer sequence and its application in tree pattern matching in Section 2. Proposed algorithm is presented in Section 3. Section 4 demonstrates a thorough experimental evaluation of XFinder. After discussing related work in Section 5, Section 6 concludes the paper.

2 Background

We introduce PRIX method [12] which retrieves exact tree matches using Prüfer sequences [11].

Prüfer Sequences. Prüfer's method [11] establishes a one-to-one correspondence between trees and sequences. Given a tree T, we first add a dummy node as the child for every leaf in T forming tree T'_N of N nodes, as proposed in [12]. Then nodes in T'_N are numbered during post-order tree traversal. We construct T's Prüfer sequence by deleting nodes in T'_N in the increasing order of their post-order numbers as follows. We start with the deletion of the leaf node with the smallest number in T'_N and record the number of its parent: n_1. The resulting tree is denoted as T'_{N-1}. Then we delete the leaf node with the smallest number in T'_{N-1}, and record its parent's number n_2. We continue the process until only the root node remains. The number sequence obtained $(n_1, n_2, ...)$ is called the *Numbered Prüfer Sequence* (NPS). If each number in NPS is replaced by its corresponding XML label (element tag or attribute name), the new sequence obtained is called as *Labeled Prüfer Sequence* (LPS). We use *qNPS* and *qLPS* to denote the numbered and labeled Prüfer sequences for the XML query, respectively. Similarly, we define *dNPS* and *dLPS* for XML document.

Example 2.1: Consider the sample XML document and twig query in Figure 1, where each node is named as a concatenation of its node label and its post-order number (after adding dummy children to leaf nodes, which are not shown in the Figure). Using the method described above, LPS and NPS can be constructed as shown in Figure 1. *qIdx* and *dIdx*, represent the index of *qLPS/qNPS* and *dLPS/dNPS*, respectively. For instance, for $dIdx = 3$, $dNPS[3]$=15 and $dLPS[3]$=A.

We use dotted lines in the sequences to denote the corresponding subtrees, e.g., an arrow between query sequence at qIdx=4 and qIdx=6 denotes that the nodes between them correspond to a subtree. ■

Prüfer Sequence Based Tree Structural Matching. Tree pattern queries are evaluated as searching subsequences in dLPS that match qLPS [12]. LPS match ensures content matching between XML tree and query tree. Then filtering step needs to be performed to validate tree structure using their NPS. Three criteria are checked, connectedness, gap consistency and frequency consistency.

Connectedness ensures that nodes in a sequence form a tree. Let i be the index of last occurrence of a post-order number n in NPS. Then $NPS[i+1]$ should record post-order number of n's parent.

Example 2.2: In document sequence *dNPS* in Figure 1, the last occurrence of node $D8$ is at index 7, then the node at index 8: $A11$ is its parent. ■

Gap consistency ensures that node relationships with respect to trees are consistent between document subsequence and query sequence. Gap is defined as difference between two consecutive numbers of an NPS sequence, giving the information about tree structural relationship between corresponding nodes. A negative value indicates a *child-parent* relationship and a positive value indicates an *ancestor-descendant* or *parent-child* relationship. The query sequence Q is gap consistent with respect to a document matching subsequence D if (1) they are of same length; and for every pair of adjacent nodes in Q and the corresponding adjacent nodes in D, their gaps g_Q and g_D (2) have the same sign, and (3) if $|g_Q| > 0$ then $|g_Q| \leq |g_D|$, else $g_Q = g_D = 0$. Intuitively, condition (1) ensures that every node in the query has a match in the data. Condition (2) ensures that the parent-child, ancestor-descendant relationship between adjacent nodes in document sequence and query sequence are consistent. Condition (3) ensures that some document nodes may be skipped to match the whole query tree.

Example 2.3: In Figure 1, document subsequence $B3$-$A15$ with gap -12, is gap consistent with query subsequence $B2$-$A8$ with gap -6. ■

We define *path direction* based on the concept of gaps. Path direction is *ascending* if gap is negative, that is, the post-order number of next node is bigger than that of current node (such as $B3$-$A15$); and path direction is *descending*, otherwise (such as $A15$-$H5$). We call two paths are *direction matching* if they are both ascending, or both descending, indicating the corresponding tree structure match. The document and query tree in Figure 1 are annotated with arrows. Solid arrows indicate ascending paths, going from a node to its parent; dashed arrows indicate descending paths, going from a node to a child or descendant.

Frequency consistency ensures that number of children of a node in document subsequence matches that of query subsequence. Two sequences are frequency consistent if they are of same length, and nodes at same index in both sequences have the same number of occurrences and always occur at the same index positions. Number of occurrences of a node is determined by number of its children, and position of node occurrence depends on subtree size.

Example 2.4: Query subsequence $B2$-$A8$-$F6$-$D7$-$A8$ is frequency consistent with a document subsequence $B3$-$A15$-$F13$-$D14$-$A15$. $A8$ appears twice, at position 2 and 5 in the query subsequence; and $A15$ appears twice, at position 2 and 5 in the document subsequence. ■

3 Algorithm

We propose a novel algorithm to compute top K matches between an XML document and a twig query. As discussed in Section 1, it is not efficient to first compute longest common subsequences between the document and query sequences, and then prune invalid ones according to connectedness, gap consistency and frequency consistency, due to large intermediate results. To address this challenge, we explicitly embed gap consistency checking using path direction matching when we perform common subsequence search. Furthermore, our subsequence matching stage also implicitly ensures connectedness and frequency consistency. Therefore generated common subsequences are guaranteed to correspond to valid subtree matches without generating invalid common subsequences as intermediate results.

Algorithm 1
$getTopKCCS(k, qLPS, dLPS, qNPS, dNPS)$

```
1:  $dIdx \leftarrow 1; qIdx \leftarrow 1$
2:  $(dIdx, qIdx) \leftarrow pathInit(dIdx, qIdx)$
3:  for $i = dIdx + 1$ to $|dNPS|$ do
4:    $qLCCS[1] = qLPS[qIdx]; qNCCS[1] = qNPS[qIdx]$
5:    $dLCCS[1] = dLPS[dIdx]; dNCCS[1] = dNPS[dIdx]$
6:    for $j = qIdx + 1$ to $|qNPS|$ do
7:      if $inSubTree$ then $Seq = TEMP$
8:      else $Seq = CCS$
9:      end if
10:     if $qLPS[j] = dLPS[i]$ then $M$
11:     else $\overline{M}$
12:     end if
13:     if $qNCCS[|qNCCS|] > qNPS[j]$ then $q \downarrow$
14:     else $q \uparrow$
15:     end if
16:     if $dNCCS[|dNCCS|] > dNPS[i]$ then $d \downarrow$
17:     else $d \uparrow$
18:     end if
19:     case
20:       $M/q \uparrow /d \uparrow$: $i, j, Seq \leftarrow lccsAppendNode(i, j, Seq)$
21:       $M/q \uparrow /d \downarrow$: $i \leftarrow skipSubTree(i, dNPS)$
22:       $M/q \downarrow /d \uparrow$: $j \leftarrow skipSubTree(j, qNPS)$
23:       $M/q \downarrow /d \downarrow$: $i, j \leftarrow pathFinder(i, j, Desc)$
24:       $\overline{M}/q \uparrow /d \uparrow$: $i, j \leftarrow pathFinder(i, j, Asc)$
25:       $\overline{M}/q \uparrow /d \downarrow$: $i \leftarrow skipSubTree(i, dNPS)$
26:       $\overline{M}/q \downarrow /d \uparrow$: $j \leftarrow skipSubTree(j, qNPS)$
27:       $\overline{M}/q \downarrow /d \downarrow$: $i, j \leftarrow pathFinder(i, j, Desc)$
28:     end case
29:   end for
30:   $maximalCheck()$
31:   $scoreNRank(k, CCS)$
32:   $initializeVars(TEMP, CCS)$
33:   $(dIdx, qIdx) \leftarrow pathInit(dIdx + 1, qIdx + 1)$
34:   $i \leftarrow dIdx$
35: end for
```

3.1 Finding Top K Sequences

Our algorithm takes the labeled and numbered Prüfer sequences of query and document (qLPS, dLPS, qNPS, dNPS) as input, and outputs top K matches, as presented in Algorithm 1.

To start the search, we first invoke procedure *pathInit* (Alg 1: line 2) to find a "seed", which is an tree edge, or equivalently, a subsequence of length two in document and query sequences that match labels and have an ascending path direction. We only need to consider ascending paths since they are inherently connected, and every descending path has a corresponding ascending path to reconnect to a root of the subtree.

Procedure *pathInit* (refer to Algorithm 4) starts with finding an initial consecutive node pair in an ascending path ($dNPS[i+1] > dNPS[i]$) in the document sequence from a given starting index (`dIdx`). Then it searches the query sequence from a given starting index (`qIdx`) for an ascending pair ($qNPS[j+1] > qNPS[j]$) with matching labels ($dLPS[i] = qLPS[j]$ and $dLPS[i+1] = qLPS[j+1]$). If no matching query node pair is found for the initial document node pair, *pathInit* advances to the next document ascending node pair and resumes the query search from the initial query start point.

Example 3.1: In Figure 1, *pathInit* finds the first document ascending node pair: $K2$-$B3$. Since there is no matching pair in the query sequence, it advances document index and yields next ascending node pair: $B3$-$A15$. This time, there is a matching node pair $B2$-$A8$ in query sequence that matches label and ascending path direction. So $B3$-$A15$ and $B2$-$A8$ become the initial "seed" for document and query, respectively. ■

After the initial "seed" is found, the algorithm then proceeds to a pair of nested loops (Alg 1:lines 3 & 6). The outer loop builds the longest common-connected subsequence (CCS) from the seed match. The inner loop matches document and query sequences, and advances them accordingly. We traverse document sequence in outer loop and query sequence in inner loop so that in face of a mismatch only query sequence needs to be "rewound". During the traversal, we compare the labels of the query and document nodes, and compare their numbers with respect to the last node in their corresponding sequences for path direction. We differentiate 8 possible scenarios (Alg 1:lines 20-27) based on the following three conditions of the comparisons: (a) the labels for the next document and query nodes match (M) or not($\overline{M}$); (b) document path direction is ascending ($d \uparrow$) or descending ($d \downarrow$); (c) query path direction is ascending ($q \uparrow$) or descending ($q \downarrow$).

We then take one of three possible actions to handle the above 8 cases: *appending* the nodes to CCS, if labels match and both sequences ascend; *skipping* the subtree for the sequence with a descending path, if only one sequence descends; or *finding* path for reconnecting the root of respective subtrees, if both sequences descend. These actions are illustrated next.

Appending Node. We append the current node to CCS by invoking procedure *lccsAppendNode* (Algorithm 2). This action is taken when current query and document nodes have matching labels and both are on ascending paths ($M/q \uparrow /d \uparrow$ case), representing a matching connected subtree.

Example 3.2: In Figure 2, document's edge $F13$-$D14$ matches query's edge $F6$-$D7$, and also document's edge $D14$-$A15$ matches query's edge $D7$-$A8$. ■

Skipping the Subtree. We skip the subsequence that corresponds to a subtree by invoking procedure *skipSubTree* (Algorithm 3). This action is taken if the sequences do not match path direction, indicating a mismatch of their corresponding tree structure. The sequence (document or query) that follows a descending path, leads to a subtree rooted at the prior node. The sequence that follows an ascending path, leads to

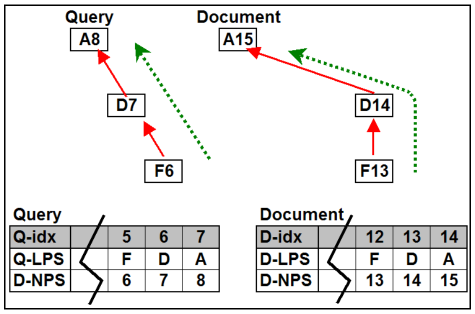


Figure 2. “*lccsAppendNode*” procedure

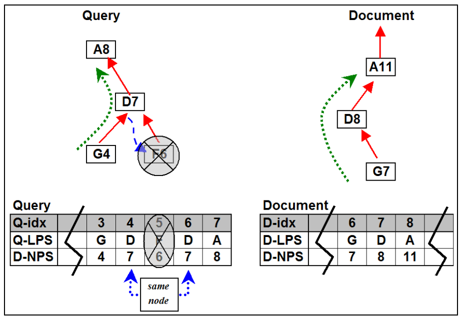


Figure 3. “*skipSubTree*” procedure

the parent node. In this case, the algorithm searches for a possible structure match after skipping the subtree of the descending sequence. To identify the subtree, we notice that for every descending path in a Prüfer sequence, there is an ascending path that reconnects to the root of the subtree, forming a closed loop. Therefore the algorithm records the NPS value of the current node in the descending sequence, and advance the sequence until this NPS value is re-encountered.

Example 3.3: In Figure 3, $G4$-$D7$ of query matches with $G7$-$D8$ of document. However, the next node after $G4$-$D7$ in the query sequence, node $F6$, leads to a descending path. While the next node after $G7$-$D8$ in the document sequence, node $A11$, leads to an ascending path. As such, we skip the query subtree rooted at node $D7$. The algorithm records the NPS value 7, and advances the query sequence till the node with the same NPS value is reached, i.e. node $qIdx[6]$. Although the subtree in Figure 3 has only one node, the same principle applies to a subtree of any size. ■

Finding a Reconnecting Path. We find a matching subsequence within document and query subtrees by invoking procedure *pathFinder* in Algorithm 5. This action is taken when the document and query sequences both follow a descending path direction, indicating that the next node in the sequence is the leftmost node in the subtree rooted at the current node. *pathFinder* searches for common subsequences in these subtrees and ensures that the found common subsequences correspond to a connected tree structure.

This search can be reduced to original problem of finding matching sequences in document and query trees, except that we limit the seed search and appending node to common connected sequence within the boundaries of a subtree and with the constraint that the last node in the common subsequence must contain the root nodes to ensure connectedness. Two stacks are used (`qStack` and `dStack`) to record the NPS values of query and document subtree root nodes (named as *pivot nodes*, `pNodes`), respectively. When the query and document sequences simultaneously descend into subtrees, the algorithm recursively treats the subtrees and pushes the root nodes onto stacks. The common sequence found in a subtree is recorded in temporary variable `TEMP` (Alg 5:lines 4-8). If the found common subsequence is connected to the corresponding `pNodes` at the top of the stacks, then it is appended to the current CCS. We clear off a pivot node that was added to the stacks after its subtree has been traversed.

Example 3.4: Figure 4 illustrates *pathFinder* in two cases. First case, shown in Figure 4(a), after $B3$-$A15$ and $B2$-$A8$ are found, their next corresponding nodes, $H5$ and $G4$ simultaneously follow descending paths into subtrees. *pathFinder* records NPS values of current nodes, 15 and 8, in respective stacks, and proceeds to search for a common subsequence within the subtrees that contain $H5$ and $G4$, rooted at $A15$ and $A8$. *pathInit* is invoked to find a seed in the subtrees, document's $G7$-$D8$ and query's $G4$-$D7$. Then the seed is extended by *lccsAppendNode* to $G7$-$D8$-$A11$ and $G4$-$D7$-$A8$. Looking at document and query stacks, we find that query subsequence connects to root of the subtree:$A8$; while the document subsequence does not connect to root of the subtree:$A15$. This indicates matching document sequence does not correspond to connected subtree; hence this match is discarded.

Then *pathFinder* identifies no valid matches in the current subtrees rooted at $A8$ and $A15$ and advances the document sequence to the next node beyond the subtree, $F13$, resets the query sequence index to the previous match node $G4$, and clears the stacks. This corresponds to the second case, shown in Figure 4(b). The current nodes $F13$ and $G4$ simultaneously follow descending paths into subtrees. *pathFinder* finds matching subsequences $F13$-$D14$-$A15$ and $F6$-$D7$-$A8$ in the subtrees rooted at $A15$ and $A8$, respectively. Since both subsequences simultaneously reach the roots of corresponding subtrees, they are appended to the current CCS, and we have $B3$-$A15$-$F13$-$D14$-$A15$ and $B2$-$A8$-$F6$-$D7$-$A8$ ■

Exit from inner loop (Alg 1:line 29) indicates a complete traversal of the query sequence. Now we check whether the newly found CCS has already been found using *maximalCheck*. Then *scoreNRank* procedure ranks this CCS and records it if it is among the top K results. All intermediate variables are cleared using *initializeVars*, document sequence is advanced, and the search for a new "seed" starts.

Algorithm 2 $lccsAppendNode(dIdx, qIdx, Seq)$

```
1: $docPopFlag \leftarrow false; qryPopFlag \leftarrow false$
2: if $dNPS[dIdx] = dStack.top()$ then $docPopFlag \leftarrow true$
3: end if
4: if $qNPS[qIdx] = qStack.top()$ then $qryPopFlag \leftarrow true$
5: end if
6: if $docPopFlag \wedge qryPopFlag$ then $dStack.pop(); qStack.pop()$
7: end if
8: if $docPopFlag = qryPopFlag$ then
9:   if $stacksempty$ then $InSubTree \leftarrow false$
10:  end if
11:  $dLSeq.append(dLPS[dIdx]); dNSeq.append(dNPS[dIdx])$
12:  $qLSeq.append(qLPS[qIdx]); qNSeq.append(qNPS[qIdx])$
13:  $dIdx \leftarrow dIdx + 1$
14: else
15:  $dIdx, qIdx \leftarrow pathFinder(dIdx, qIdx, \text{"Asc"})$
16: end if
17: return dIdx,qIdx,Seq
```

Algorithm 3 $skipSubTree(dqIndex, dqNPS)$

```
1: $skipVal \leftarrow dqNPS[dqIndex - 1]$
2: $dqIndex \leftarrow dqIndex + 1$
3: while $dqNPS[dqIndex] \leq skipVal \wedge dqIndex < dqNPS.size()$ do
4:   $dqIndex \leftarrow dqIndex + 1$
5: end while
6: return dqIndex
```

Algorithm 4 $pathInit(dIdx, qIdx)$

```
1: if $stack\ empty$ then $doc/qryStopIndex \leftarrow last\ doc/qIdx$
2: else $doc/qryStopIndex \leftarrow doc/qryTopOfStack$
3: end if
4: $i \leftarrow dIdx; j \leftarrow qIdx$
5: while $i < docStopIndex$ do
6:   if $document\ ascending\ wrt\ next\ node$ then
7:     while $j < qryStopIndex$ do
8:       if $query\ label = document\ label \wedge next\ query\ label = next\ document\ label \wedge query\ ascending\ wrt\ next\ node$ then return $i, j$
9:       end if
10:      $j \leftarrow j + 1$
11:    end while
12:  end if
13:  $i \leftarrow i + 1; j \leftarrow qIdx$
14: end while
15: return i,j
```

3.2 Ranking top K Sequences

A top K longest subsequence between query and document can be shorter than the original query sequence. We call each matching query subsequence as *modified query*, and record the number of deletions required to convert the query to a modified query as *edit cost.* We have shown that the matching document subsequence and modified query sequence are *connected*, *gap consistent* and *frequency consistent* in [1]. This

Algorithm 5 $pathFinder(dIdx, qIdx, Direction)$

```
1: $qryPivotNode \leftarrow qNTemp[|qNTemp|]$
2: $docPivotNode \leftarrow dNTemp[|dNTemp|]$
3: if $\underline{Direction =}$ "$Desc$" then
4:   if $\overline{InSubTree}$ then
5:     $qLTemp[1] = qLPS[qIdx]$
6:     $qNTemp[1] = qNPS[qIdx]$
7:     $dLTemp[1] = dLPS[dIdx]$
8:     $dNTemp[1] = dNPS[dIdx]$
9:     $qStack.push(qNCCS[|qNCCS|])$
10:    $dStack.push(dNCCS[|dNCCS|])$
11:  end if
12:  $InSubTree \leftarrow TRUE$
13:  if $qLTemp[|qLTemp|] < qryTopOfStack \wedge dLTemp[|dLTemp|] < docTopOfStack$ then
14:    $qStack.push(qLTemp[|qLtemp|])$
15:    $dStack.push(dLTemp[|dLTemp|])$
16:  end if
17:  $docReStart, qryReStart \leftarrow pathInit(dIdx, qIdx)$
18:  return $docReStart, qryReStart$
19: else
20:  if $InSubTree$ then
21:    $docReStart, qryReStart \leftarrow pathInit(docReStart + 1\ ,\ qryReStart + 1)$
22:    return $docReStart, qryReStart$
23:  else
24:    return $dIdx, |qNPS|$
25:  end if
26: end if
```

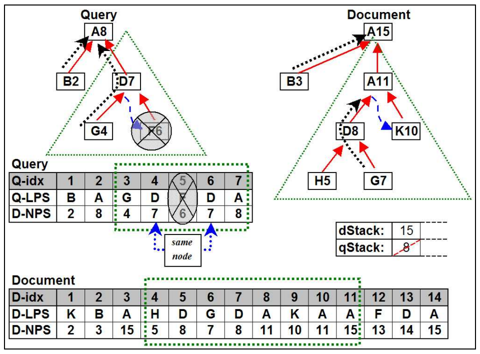
guarantees that a sequence match corresponds to a valid tree structure match. Therefore, the ranking scheme for partial matches is based on the edit cost (the cost of node deletion) between the original query sequence and the modified one.

Example 3.5: In Figure 1, for the seed match $B2$-$A8$ in the query and $B3$-$A15$ in the document, Algorithm 1 retrieves the CCS $B2$-$A8$-$F6$-$D7$-$A8$ and $B3$-$A15$-$F13$-$D14$-$A15$, which represent valid subtree match. This matching sequence has an edit cost of 2 with the original query sequence. Then the control is returned to the outer loop which invokes *pathInit* (Alg 1:line 33) and finds the next seed, query's $G4$-$D7$ and document's $G7$-$D8$. Then another valid subsequence match, $G4$-$D7$-$A8$ and $G7$-$D8$-$A11$, is found with an edit cost of 4, which has a lower rank than the previous CCS. ■

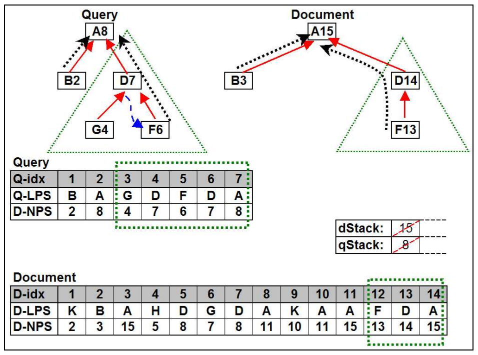
4 Experiments

XFinder is implemented in Java. To evaluate XFinder, we compare it with another two approaches[4]: tree matching approach [13], referred as "Tree-Edit Distance approach" henceforth and a Baseline approach for reducing approximate tree pattern matching to subsequence matching. The Baseline approach first computes all the common subsequences (of length greater than or equal to 2) between XML document and query. It then filters out matching subsequences

[4]Since the implementations of [10, 2, 16] are not available, we didn't make a comparison with them



(a) pathFinder - subsequence finds no path to root A15



(b) pathFinder - subsequence finds path to roots A8/A15

Figure 4. "$pathFinder$" procedure

that do not correspond to valid tree structures by checking connectedness, gap consistency and frequency consistency. Similar as XFinder, valid subsequences are ranked and the top K ones are output. Notice that XFinder returns the same set of top K matches as the Baseline approach. However, XFinder pushes the tree validity check during the common subsequence search to improve efficiency. We do not include XML tree parsing time while clocking the execution time for a fair comparison with Tree-Edit Distance approach[5] and use an in-memory representation for the XML document/query trees.

Both synthetic and real-world datasets are tested of size from 0.5MB-5.0MB, in 0.5MB increments. Synthetic dataset is generated using XMark[6] XML data generator. DBLP[7] is used as real-world dataset.

Ten queries were designed for each dataset with different sizes as well as different numbers of node matches, as listed in [1]. For first five queries with increasing size, the number of nodes in the query that have data matches remains same, while the number of unmatched query nodes are increased. The remaining fives queries are opposite.

4.1 Experimental Results

Figures 5 through 7 show execution time comparison among XFinder, Tree-Edit Distance and Baseline approach on both DBLP and XMark datasets. Here the execution time is presented in log (to the base 10) scale due to the large difference in execution time of these three approaches. Three sets of experiments are performed, with varying document size, varying query size and varying K for returning top K query results, respectively, to test the effect of each parameter.

Increasing Data Size. Figure 5(a), 5(b) shows query execution time of XFinder, Tree-Edit Distance and Baseline approach for DBLP dataset when document size increases for queries Q1 and Q5, respectively, (K=1). Performance comparisons for other queries are similar, as shown in [1]. Both Tree-Edit Distance and Baseline approaches run out of memory after 1.0MB and 3.5MB document size, respectively, for Q1 and Q5. The time required by Tree-Edit Distance approach is more than 2 orders of magnitude than that of XFinder, and slope of its curve is also steeper than that of XFinder. Baseline approach is slower than Tree-Edit Distance and much slower than XFinder, with a higher value of the slope of the curve. This is due to the expensive matching step of computing all possible common subsequences between an XML document and a tree pattern query, followed by the filtering step of checking and removing invalid sequences. Similar behavior is observed when we compared XFinder with Tree-Edit Distance and Baseline approaches for XMark dataset as document size increases for queries Q11 and Q15, as presented in Figure 5(c), 5(d).

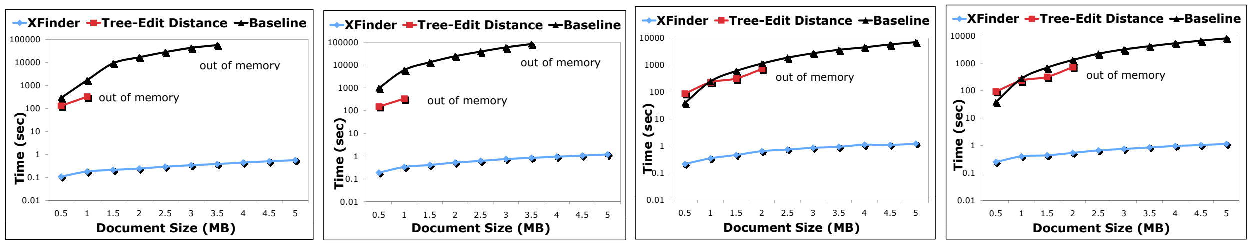
Increasing Query Size. Figures 6(a), 6(b) and 6(c),6(d) show the execution time of XFinder, Tree-Edit Distance and Baseline approaches for DBLP and XMark dataset, respectively, as the query size increases for fixed document size (100KB), K=1. Particularly, in Figure 6(a) and Figure 6(c) as the query size increases, the number of unmatched query nodes increases, while the number of matched query nodes stays the same. While in Figure 6(b) and Figure 6(d), as the query size increases, only the number of matching query nodes increase. The execution time of XFinder and Tree-Edit Distance only increases a little as the query size increases, while the Baseline approach has a larger increase due to a large number of false positive matches. Again, XFinder is much faster than the other two approaches.

Both Tree-Edit Distance and Baseline approach are more sensitive to the increase in XML document size when compared against XFinder. XFinder may look more sensitive to the increase in query size as compared to Tree-Edit Distance. However, the slope of the curve

[5]Tree-Edit Distance approach does not include XML tree parsing time in their execution time.

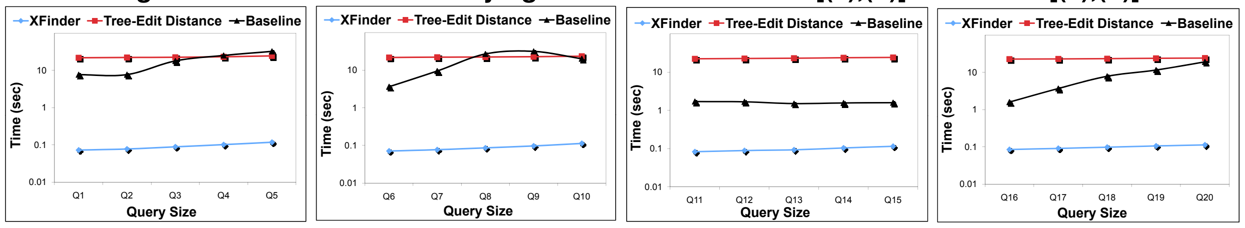
[6]http://monetdb.cwi.nl/xml/index.html, uses auction DTD

[7]http://www.cs.washington.edu/research/xmldatasets/



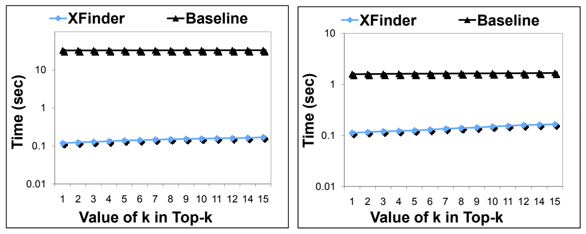
(a) Q1 (b) Q5 (c) Q11 (d) Q15

Figure 5. Performance for Varying Document Size DBLP[(a),(b)] and XMark[(c),(d)]



(a) Q1-Q5 (b) Q6-Q10 (c) Q11-Q15 (d) Q16-Q20

Figure 6. Performance for Varying Query Size on DBLP[(a),(b)] and XMark[(c),(d)]



(a) DBLP, Q5 (b) XMark, Q15

Figure 7. Performance with Varying K

is low and in practice the size of a user query is small.

Increasing the Number of Results. Figure 7(a), 7(b) show the running time when K ranges from 1 to 15 for fixed DBLP and XMark document of size $100KB$, and fixed query $Q5$ and $Q15$, respectively. Since Tree-Edit Distance approach only returns the best match between XML document and query (K=1) by computing the minimum number of operations required to transform the document to the query, we only compared the performance between XFinder and Baseline approach. As we can see, Baseline approach stays constant. This is because irrespective of the value of K, all common subsequences are computed, which is the most expensive task, then top K valid sequences are output. While the execution time of XFinder increases slowly and linearly with respect to K. This behavior is very desirable since K is often small, and computing only the top K sequences without generating all the common subsequences is much more efficient.

Verifying Correctness. We record the number of common subsequences found by Baseline approach and the number of valid sequences among them. We also record the number of matches reported by XFinder. For DBLP dataset, Figure 7(a), Baseline approach generates 623 common subsequences, out of which 15 are valid. XFinder finds all 15 valid subsequences without generating any invalid subsequences if we set $K \geq 15$. For XMark dataset, Figure 7(b), Baseline approach generates 44 common subsequences, out of which 16 are valid. Again, XFinder finds all 16 valid subsequences without generating invalid subsequences if we set $K \geq 16$. This shows that XFinder is very efficient in computing top K query matches by avoiding large intermediate result generation, without compromising search quality.

5 Related Work

Several efforts have been made for supporting approximate XML query processing. [16] proposed FleXPath to search approximate matches by query relaxation. Later, *Whirlpool* [8] was proposed for adaptive join order evaluation. Ranking schemes inspired by *tf-idf* are used to rank join predicates and to adaptively select the best join order. However, the size of the relaxed query can be exponential to the original query, and decomposing a query to sub-queries followed by joins can generate large intermediate results. TWIX [2] assigns a unique label to each node in the tree and uses string alignment for structural matching. A bottom-up breadth-first search is explored using the matched leaves as initial points and the tree matches are ranked based on tree edit distance. On the other hand, XFinder employs a bottom-up depth-first search looking for similar "path-structures" in the query and document. TreeSketch [10] uses the notion of graph synopsis to estimate the selectivity of XML twigs and compute approximate query answers. It finds all possible match results based on the synopsis of XML document. In contrast XFinder produces top K query results from the document itself. TopX [14] computes approximate matches based on pre-computed index lists for individual tag-term content conditions. XFinder reduces approximate tree matching problem to subsequence matching, where tree validity check is pushed inside subsequence matching.

Another line of research finds similarity between two trees of comparable size. [13] proposed approximate tree matching based on edit distance, i.e., the number of insertion, deletion and relabel operations needed to convert one tree to the other. Due to its high computational complexity, a lot of research approximates the tree edit distance metrics. [5] handles approximate matching in terms of joins, such that two trees that are at most τ tree edit distance apart are considered as matches. [7] proposed algorithms to compute lower bound to tree edit distance using a variety of refinements like maximum leaf path histogram, degree of nodes-histogram and content histogram. Top K documents are picked based on these preliminary scores and tree-edit distance is computed for result ranking. [17] proposed a lower bound to tree-edit distance for ordered XML documents by extending q-grams from string edit distance domain. [4] approximates tree edit distance by emphasizing more on the structure of the tree, e.g. deleting a leaf node is less significant than deleting a non-leaf node. The tree similarity approach assigns a score to a tree depending on its similarity with the other tree. This requires the two trees to be of comparable size. Whereas, XFinder searches for similar query tree patterns in one or more, much larger XML tree(s) and assigns a score depending on the extent of match.

6 Conclusions

In this paper, we present XFinder, a system for approximate tree pattern matching on ordered XML data. It reduces the problem of approximate tree pattern matching to a simpler problem: finding common subsequences in the sequences converted from data and query, which can be more efficiently evaluated. However, not all common subsequences represent valid tree matches. XFinder pushes the tree validity checking into sequence matching stage to achieve performance speedup. Experiments show that XFinder has substantial performance improvement over an existing approach [13] and a baseline approach. As a future initiative, we are extending XFinder by exploiting various indexing schemes which are critical for querying very large XML documents.

References

[1] N. Agarwal, M. Galan, and Y. Chen. Approximate Structural Matching over XML Documents. Technical report, Arizona State University, 2006.

[2] S.A. Aghili, H. Li, D. Agrawal, and A. El Abbadi. TWIX: Twig Structure and Content Matching of Selective Queries using Binary Labeling. In *Proceedings of INFOSCALE*, 2006.

[3] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated Ranking of Database Query Results. In *Proceedings of CIDR*, 2003.

[4] N. Augsten, M. Böhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. In *Proceedings of VLDB*, 2005.

[5] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *Proceedings of SIGMOD*, pages 287–298, 2002.

[6] S. Guha, R. Rastogi, and K. Shim. ROCK: A Robust Clustering Algorithm for Categorical Attributes. *Information Systems*, 25(5):345–366, 2000.

[7] K. Kailing, H. P. Kriegel, S. Schfnauer, and T. Seidl. Efficient Similarity Search for Hierarchical Data in Large Databases. In *Proceedings of EDBT*, 2004.

[8] A. Marian, S. A. Yahia, N. Koudas, and D. Srivastava. Adaptive Processing of Top-k Queries in XML. In *Proceedings of ICDE*, 2005.

[9] U. Nambiar and S. Kambhampati. Answering Imprecise Queries over Autonomous Web Databases. In *ICDE*, 2006.

[10] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. In *SIGMOD*, 2004.

[11] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik and Physik*, 27:142–144, 1918.

[12] P. Rao and B. Moon. PRIX: Indexing and querying XML using Prfer sequences. In *Proceedings of ICDE*, 2004.

[13] D. Shasha and K. Zhang. Approximate Tree Pattern Matching. In *Pattern Matching Algorithms*. Oxford University Press, 1997.

[14] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *Proceedings of VLDB*, 2005.

[15] S. A. Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and Content Scoring for XML. In *Proceedings of VLDB*, 2005.

[16] S. A. Yahia, Laks V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-text Querying for XML. In *Proceedings of SIGMOD*, 2004.

[17] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity Evaluation on Tree-Structured Data. In *SIGMOD*, 2005.