Diversifying Query Results on Semi-Structured Data

Mahbub Hasan
UC Riverside
hasanm@cs.ucr.edu

Abdullah Mueen
UC Riverside
mueen@cs.ucr.edu

Vassilis Tsotras
UC Riverside
tsotras@cs.ucr.edu

Eamonn Keogh
UC Riverside
eamonn@cs.ucr.edu

ABSTRACT

Queries on the web can easily result in a large number of results. Result Diversification, a process by which the query provides the k most diverse set of matches, enables the user to better understand/explore such large results. Computing the diverse subset from a large set of results needs a massive number of pair-wise distance computations as well as finding the subset that maximizes the total pair-wise distance, which is NP-hard and requires efficient approximate algorithm.

The problem becomes more difficult when querying semi-structured data, since diversity can occur not only in the document content but also (and more importantly) in the document structure; thus one needs to efficiently measure the structural differences between results. The *tree edit distance* is the standard choice but, is too expensive for large result sets. Moreover, the generalized tree edit distance ignores the context of the query and also the content of the documents resulting in poor diversification. We present a novel algorithm for meaningful diversification that considers both the structural context of the query and the content of the matched results while computing pair-wise distances. Our algorithm is an order of magnitude faster than the tree edit distance with an elegant worst case guarantee.

We also present a novel algorithm to find the top-k diverse subset of matches. Our algorithm skips unnecessary distance computations and works in time linear on the size of the result-set. We experimentally demonstrate the utility of our algorithms as a plug-in for standard query processors without introducing large error and latency to the output.

1. INTRODUCTION

Vast repositories of semi-structured data exist on the web and are accessed by user queries typically using an XML query language (such as XPath [10] and XQuery [8]). It is typical for such queries (especially when searching the web) to return a large answer set, making it quite a challenge for the user to capture/view the whole result space. Result Diversification has been recently introduced for relational datasets [13][23][24], as an approach to ease interpreting a massive result set by returning the k most diverse results. Algorithms for XML result diversification have also been proposed recently [11][21] but, they only consider the data content (i.e. keywords) of the query.

What makes the problem challenging is that diversity can occur not only in the content of the documents but also (and more importantly) in the structure of the documents. Since the XML-based query can contain ancestor-descendent(//) or wildcard(*) relationships, there maybe significant structural differences (e.g., additional nodes in a matched path) among the returned results. Such diversity will not be explored by the content-only based diversification; instead we need an approach that takes into account differences *both* in the structure and content of the results.

To elaborate, let us consider an example document of bibliographic records shown in figure 1. The document has three records: two PhD theses and a paper written by two different authors. An example XPath query (in figure 2) describing "Find all bibliographic entries of Faloutsos", has three exact matches shown by the thick lines in figure 1. Assume instead that we want to present the user with the two most diverse results. We should then provide the pair of matches (among the three possible pairs) that exhibits the highest diversity. Among the three matches, the one on the right (match 3) is structurally different from the other two because it is a record of a paper whereas the others are records for PhD theses. The matches on the left (match 1) and in the middle (match 2) are different because of the contents of the "PhDThesis" records (match 1 is a record for "Michalis" while match 2 is a record for "Christos"). Ideally, we would like to return to the user matches 2 and 3, since they are different both in content and in structure. Therefore, we need a diversification method that combines *both* the structural and content-based differences of the results.

A naive way to find the most diverse k-subset from a set of N returned results, is to take the maximum of the total pair-wise distance as a measure of diversity for all of the $\binom{N}{k}$ subsets. Typically N and k are thousands and tens, respectively. Such an instance of the problem requires 100^2 distance computations. The distance measure, therefore, *must* be very efficient to keep the computation time tolerable. In addition to that, the number of times distances are computed *must* be reduced.

For structural query processing, a popular choice [6] of distance measure is the *tree edit distance* [25]. We focus on extending the tree edit distance in two ways (section 4). First, we consider the contents of the nodes and also the structural context of the query to perform well in presence of both types of differences and thus, provide meaningful diversification. Second, we leverage off the known skeleton (i.e. the query) of the results to compute the distance measure faster. We present a novel algorithm to achieve both of them. Our distance measure is comprehensive and our algorithm is at least an order of magnitude faster than the generalized tree

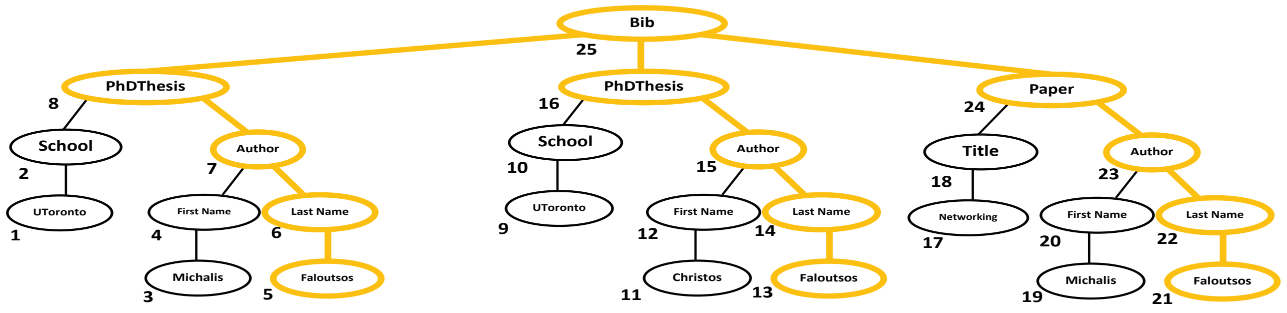


Figure 1: Bib document containing three records, (a) PhDThesis of Michalis Faloutsos, (b) PhDThesis of Christos Faloutsos and (c) a Paper of Michalis Faloutsos.

edit distance with $O(n^2)$ worst case time complexity, where n is the number of nodes in the comparing trees.

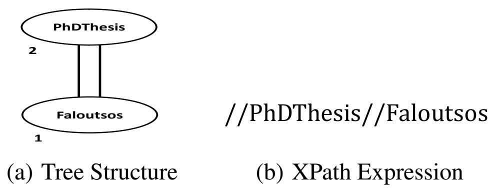


Figure 2: (a) An example query for the documents in figure 1. (b) The XPath expression for the same query.

Diversification is an NP-hard [15] problem. Therefore, enumerating all of the subsets to measure their goodness is necessary for exactness but prohibitive even if we have the best distance measure. For efficiency, we need an approximate algorithm that checks only a tiny fraction of the number of subsets the naive algorithm checks (section 5).

2. PROBLEM FORMULATION

An XML document is an ordered labeled tree T. T is a graph with vertices $V(T)$ and edges $E(T)$. An edge $(u,v) \in E(T)$ represents a parent child relationship where u is the parent of v. Only the root has no parent. A node u can have zero or more children in a strict left to right order. Nodes with zero child are leaves. $anc(u)$ defines the set of ancestors of node u. Every node has a label denoted by $label(u)$. A postorder traversal of a tree visits the children of a node from left to right before visiting the node. For example, the postorder traversals of the trees in figure 1 are shown by the numbers beside each node. We denote the nodes of a tree by the postorder sequence $t_1, t_2, \ldots, t_n$, where $n = |T|$ is the number of nodes in T. The subtree rooted at node t_i is denoted by T_i. The postorder sequence of T_i is a subsequence of T ending at t_i and starting at $l(t_i)$. $l(t_i)$ is the leftmost node of the tree T_i. For example, $l(9) = l(6) = 5$ and $l(14) = 10$ in figure 1(c).

We are given a query Q which is also an ordered labeled tree where edges represent XPath axes. We are also given a set of XML documents $\mathcal{D}$, in which we find matches for the query.

A map between a node s_i in a tree S and a node t_j in a tree T is an ordered pair (s_i, t_j). We define a relation $M : V(S) \to V(T)$ or simply a *mapping* $M : S \to T$ such that M maps some nodes of S to some nodes of T with the following conditions.

1. $1 \leq i \leq |S|$ and $1 \leq j \leq |T|$
2. For any two pairs (s_i, t_j) and (s_u, t_v) in M
   (a) $i = u$ if and only if $j = v$ (One-to-one condition).
   (b) s_i is to the left of s_u if and only if
   t_j is to the left of t_v (Sibling-order condition).
   (c) s_i is an ancestor of s_u if and only if
   t_j is an ancestor of t_v (Ancestor-order condition).

Note that M is not a function and, therefore, is not defined for all nodes in S and T. Nodes mapped by M capture similar structure in both S and T. $M' : T \to S$ is the *inverse* mapping of M such that for all $(s,t) \in M$, $(t,s) \in M'$. If M is defined for every node $s \in S$, then M is a *complete* mapping. If M is complete, M' is not guaranteed to be complete. If both M and M' are complete, they are called *maximal* mappings.

M is called an *outer* mapping if (i) for every leaf s in S, $M(s)$ is a leaf in T and (ii) $root(T) = M(root(S))$. If both M and M' are outer then they are called *minimal* mapping. Figure 4(b) shows an example of minimal mapping between S and T.

An *exact match* of a query Q is another ordered labeled tree T, such that there is a complete and minimal mapping $M : Q \to T$ and for all $(q,t) \in M$, $label(q) = label(t)$. There has been many algorithms proposed for finding all of the exact matches of query Q in $\mathcal{D}$ [2][7][18][22][9]. There is also algorithm [6] that finds *approximate* matches where query Q may not have complete mappings. Each approximate match of the query Q is an exact match of query Q', where Q' is a relaxed version of the original query Q [4]. We consider the matching algorithm $\mathcal{A}$ as given and $\mathcal{A}(\mathcal{D}$,$Q)$ is the set of matches denoted by $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. We denote a distance measure by $d(.,.)$, which computes the dissimilarity between two matches T_i and T_j.

A set $R \subset \mathcal{T}$ of size k is the most diverse if the total pair-wise distance $\sum_{T_i, T_j \in R} d(T_i, T_j)$ is the maximum. The matches in R are said to be the top-k diverse matches for the query Q in the document set $\mathcal{D}$.

[TOP-k DIVERSE MATCHES]. *For a given Q and $\mathcal{D}$, find the k-subset R of the set of matches $\mathcal{T}$ such that the total pair-wise distance of R (i.e. $\sum_{T_i, T_j \in R} d(T_i, T_j)$) is the maximum over all such subsets.*

The optimal algorithm to find the top-k diverse matches requires enumerating all the k-subsets of the set $\mathcal{T}$ and selecting the one with maximum pair-wise distance. This algorithm has $O(|\mathcal{T}|^k)$ time complexity and therefore, too slow for interactive queries. To solve the problem efficiently there are two lines of attack; speeding up the distance measure and considering only a fraction of the subsets heuristically. In section 4, we describe our approach of computing distance very fast by taking both the structure and content of the query into account. In section 5, we describe our heuristic approach to find the diverse subset efficiently.

3. RELATED WORK

Query Processing on Semi-structured Data (i.e. XML documents) has been addressed in several occasions and there are three different types of algorithms have been proposed; path based [2][9], twig based [7][18] and sequence based [22]. In path based methods, the original query is divided into paths from root to leaves and, the matches corresponding to these paths are joined together to construct a complete match. Twig based methods perform better than path based ones by considering the twig as a whole and, therefore, eliminates expensive stitching operations. Sequence based methods first convert the query and document into sequences and perform a search for the query subsequence in the document sequence. It has been recently shown that LCS-Trim [22] outperforms the other approaches. Hence we have chosen it as the query processor for our algorithms in this paper. It should be noted that the choice of query processing algorithm is orthogonal to our problem.

Diversifying search results is also a well addressed area of research. [15] provides a general framework for the result diversification problem. Specialized solutions for relational and web databases are also proposed [1][13][23]. There are rich surveys on diversification [14][17][24] that classify the available algorithms into two principal types, the greedy best first approach and the iterative gain maximization approach. In this paper, we focused on the greedy best first method because it needs few linear scans of the data and does not depend on a large number iterations to produce better quality results.

Despite the wide range of work on diversification, there is little on diversifying XML query results. [11][21] proposed result diversification based on keyword queries instead of classic XPath or XQuery queries and concentrate only on the content information of the documents. None of these methods formally consider the structural diversity among the results. We present the first of such diversification algorithm that treats the results as trees rather than collections of labels.

Computing dissimilarity between trees using the *tree edit distance* (TED) [25] is one of the first methods for comparing tree-like structures. [16] proposed $O(n^2)$ lower and upper bounds of the tree edit distance. [5] provides an $O(n^2)$ algorithm for approximating tree edit distance through string edit distance. None of these methods defined the special case of computing TED in presence of a given seed mapping. We present the first *exact* algorithm to compute the seeded tree edit distance.

4. DISTANCE MEASURE FOR DIVERSIFICATION

To diversify a set of matches for an XML query, we need a distance measure that can compare two trees. The tree edit distance [25] is the most widely used distance measure for tree structures. The idea is to transform one tree to the other such that the total cost of the sequence of edit operations performed for the transformation is minimum and hence the distance between the two trees.

There are three types of edit operations. The *delete* operation removes a node n from the tree and connects the children of n as the children of the n's parent preserving the sibling order of the children. The *insertion* operation on a node n adds an edge from some node p to n and makes a subsequence of children of p the children of n. The *rename* operation changes the label of a node.

For every operation, an associated cost is defined. The cost can depend on the operation, the label of the node(s) being operated on as well as the context at which the operation is being performed. The simplest cost model assumes equal cost for all of the three operations: insertion, deletion and rename. Such cost model makes tree editing distance symmetric i.e. transforming any of the trees to the other yields the same distance.

Any valid mapping $M : S \rightarrow T$ can be translated to a sequence of edit operations to convert one tree to another. The sequence of operations is (i) delete all non-mapped node in S, (ii) rename all mapped nodes that do not have the same label and, (iii) insert all non-mapped nodes in T. Since, M preserves the structural similarity by the three conditions described in the definition of mapping, at any intermediate stage of the sequence of operations M remains valid. The converse is also true. If we are given a sequence of edit operations, there exists a mapping $M : S \rightarrow T$ that has cost no higher than that of the sequence of edit operations[25]. Therefore finding the least costly sequence of edit operations is the same as to finding the least costly mapping as defined below.

DEFINITION 1. *Given a mapping $M : S \rightarrow T$ and a equal cost for the operations, we define the cost(M) as*

$$cost(M) = (|S| - |M|) + (|T| - |M|) + |M_m|$$

where $M_m = \{(s, t) \in M | label(s) \neq label(t)\}$.

The term $|S| - |M|$ denotes the number of non-mapped nodes in the tree S and this is the number of deletions we need to perform. Similarly, $|T| - |M|$ is the number of insertions and $|M_m|$ is the number of rename operations.

DEFINITION 2. *Tree edit distance between S and T, $TED(S, T)$, is the smallest cost over all mappings $M : S \rightarrow T$.*

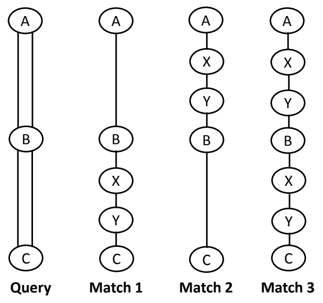
Tree edit distance finds the best possible mapping preserving the structural similarity. But while computing the distance between two matches, TED does not utilize the information that both the matches have complete-minimal mapping from the query. In the next two subsections, we present a new algorithm that uses these two mappings for computing distances. We start with adding the structural context sensitivity in the distance measure and, add the content sensitivity in the later subsection.

4.1 Context Aware Diversity

We first provide a simple example showing that TED fails to capture the desired dissimilarity because of ignoring the structural context of the query.

Consider the example in figure 3 where we have three matches for the query shown on the left. Based on the structure of the query, the two most diverse matches should be match 1 and 2. The reason is the XY segment is located in different parts of matches 1 and 2 while match 3 has some parts common with both match 1 and 2, separately. But according to the tree edit distance, all of the pairwise distances are 2. Therefore, TED cannot distinguish the two most diverse matches (i.e. 1 and 2) in this example.

More precisely, while converting match 1 to match 2, TED needs only two operations: delete B from match 1 and insert B as in match 2. Recall both of the B nodes in match 1 and 2 are mapped from the node B in the query. This implicitly maps the two B nodes of match 1 and 2 together. Therefore, B must not be deleted or inserted while the editing distance between match 1 and 2 is computed in the context of the query. However, as in the above example, the generalized algorithm for tree edit distance does not always preserve this query mapping. If we consider an implied mapping between the matches using the mappings from the query, the distance between match 1 and 2 becomes 4, and thus, makes them the most diverse pair.



Tree Edit Distance

	Match1	Match2	Match3
Match1	0	2	2
Match2	-	0	2
Match3	-	-	0

Tree Edit Distance preserving Query Mapping

	Match1	Match2	Match3
Match1	0	4	2
Match2	-	0	2
Match3	-	-	0

Figure 3: For the query shown on the left there are three matches found. The distance values for the tree edit distance and our proposed variant are shown on the right.

In the next section, we describe our algorithm to compute the modified tree edit distance that considers the query mappings as contextual information. We denote the modified distance measure as *Seeded Tree Edit Distance (STED)*.

Consider the set of matches $\mathcal{T}$ of a given query Q. Let $S, T \in \mathcal{T}$ be any two matches for the query Q as shown in figure 4 and M^S and M^T are the complete minimal mappings from Q to S and T.

We define a new mapping $M : S \to T$ where $(s_i, t_j) \in M$ for all $(q, s_i) \in M^S$ and $(q, t_j) \in M^T$. Note that, M may not be complete but always minimal. We call M a *seed* map. From now, M always refers to a minimal mapping and, therefore, the direction of the map is not important at any point.

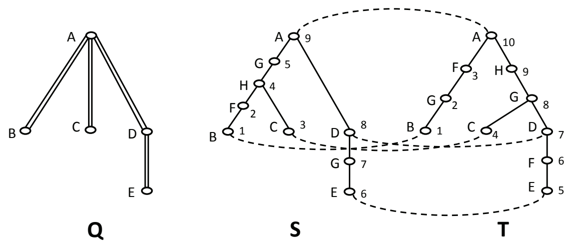


Figure 4: An example query (Q) on the left with two matches S and T. The induced minimal mapping between S and T is shown by the dashed lines.

Note that, if $\hat{M} : S \to T$ is a *super mapping* such that $\hat{M} \supseteq M$ then $cost(\hat{M}) \leq cost(M)$ under equal costs of edit operations. Because we want to preserve the seed mapping M as the context, we modify the tree edit distance to find a super mapping of M that minimizes the total cost instead of *any* mapping.

DEFINITION 3. *The seeded tree edit distance, $STED(S, T, M)$, between S and T given a minimal mapping $M : S \to T$, is the smallest cost over mappings $\hat{M} \supseteq M$.*

To compute STED using existing algorithms for computing tree edit distance, we can just change the cost model trivially. More precisely, if $(s, t) \in M$ then cost of deleting s, inserting t and mapping s (or t) to a different node $x \neq t$ (or s) is raised to infinity. This change in cost model guarantees that (s, t) would be in the optimal mapping.

The classic algorithm for tree edit distance is a dynamic programming algorithm which computes a matrix of size $|S| \times |T|$ where a cell (i, j) denotes the tree edit distance between S_i and T_j. For example, figure 5(a) shows the matrix for trees in figure 4 when the change in the cost model is adopted. Clearly most of the entries are invalid and contribute nothing to the final distance value. This motivates us to develop an efficient algorithm for finding $STED$ for two trees when the seed map is given. The algorithm is described sequentially and is justified with necessary definitions and lemmas as we go along.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
S_1	0	1	2	α	α	α	α	α	α	α
S_2	1	1	1	α	α	α	α	α	α	α
S_3	α	α	α	0	α	α	α	α	α	α
S_4	α	α	α	α	α	α	α	α	α	α
S_5	α	α	α	α	α	α	α	α	α	α
S_6	α	α	α	α	0	1	α	α	α	α
S_7	α	α	α	α	1	1	α	α	α	α
S_8	α	α	α	α	α	α	1	α	α	α
S_9	α	α	α	α	α	α	α	α	α	6

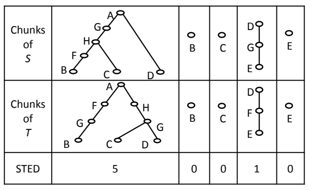


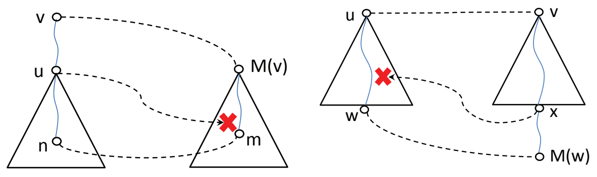
Figure 5: (a) Tree edit distance matrix of S and T (b) Mapped chunks of S, T and corresponding $STED$

Let $U^T = \{x | x \in V(T) \text{ and } \exists_y[(x, y) \in M \text{ or } (y, x) \in M]\}$ be the set of mapped nodes in a tree T. Note that all of the leaves and the root of T are in U^T. If the tree is divided at every node in U^T by keeping two copies in the two halves, we will get $|U^T|$ chunks from T. Let $C(T, M)$ or C^T in short denote the set of chunks found in tree T and C_u^T denote the chunk rooted at a node $u \in U^T$. For example, figure 5(b) shows the chunks of S and T from figure 4[1].

Since $M : S \to T$ is a one-to-one mapping, every chunk C_u^S from tree S has a mapped chunk $C_{M(u)}^T$ in the tree T. The submapping $M_u : C_u^S \to C_{M(u)}^T$ induced from M is minimal by definition. Note that no internal node in C_u^S is mapped by M. Moreover, no internal node in C_u^S will be mapped by the optimal mapping $\hat{M}$ to a node in $C_{M(v)}^T$ where $u \neq v$. The following lemma describes the fact more formally.

LEMMA 1. *Optimal mapping $\hat{M}$ for $STED(S, T, M)$ will not map any node from one chunk C_u^S to another chunk $C_{M(v)}^T$ such that there are $u, v \in U^S$ and $u \neq v$.*

PROOF. Let $n \in C_u^S$ and $m \in C_{M(v)}^T$ are two nodes in S and T (see figure 6(a)). For contradiction, lets assume $(n, m) \in \hat{M}$. Therefore, u is $anc(n)$ in S and $M(v)$ is $anc(m)$ in T. Since v and $M(v)$ are matched so v is $anc(n)$ in S. Now, by construction, v can not be in the path from u to n. Therefore, v is also $anc(u)$. Since u and $M(u)$ are matched, $M(u)$ has to be $anc(m)$ and $desc(M(v))$. Because no internal node in the path from $M(v)$ to m can be a mapped node by M, this is a contradiction. □



(a) lemma 1 (b) lemma 2

Figure 6: Contradictions of Lemma 1 & 2

Using the above lemma, we can now say that finding optimal mappings for every pair of mapped chunks is sufficient. If we only

[1]The reader may wonder why defining the leaves as *tiny* chunks. In reality, they have inconsequential effect on the performance but, helps to simplify the description by far.

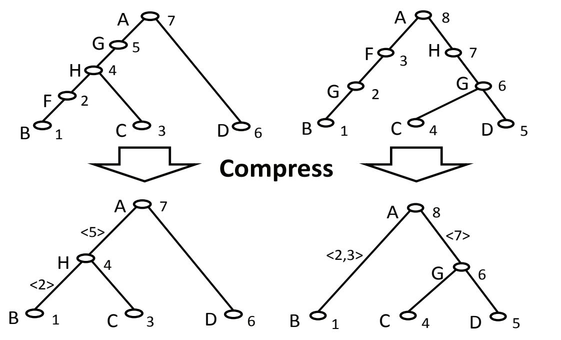
find the mappings for the mapped pairs of chunks, compute the editing distance for these mappings and then, sum these distances for all of the mapped pairs of chunks; it is the same as the optimal editing distance between S and T. Mathematically,

$$cost(\hat{M}) = \sum_{u \in U^S} cost(M_u : C_u^S \to C_{M(u)}^T) \qquad (1)$$

To find the mapping between C_u^S and $C_{M(u)}^T$, we now present an $O(n^2)$ algorithm where n is the number of nodes in C_u^S and $C_{M(u)}^T$. Since there is a one to one mapping between chunks, from now on we denote C_u^S and $C_{M(u)}^T$ by S and T for simplicity of description. Similarly, we denote M_u as M.

DEFINITION 4. *The leaf-sequence $L(u)$ of a node u in a tree T consists of the leaves of T rooted at u in the left to right order.*

For example, in the left tree of figure 7 $L(7) =< 1, 3, 6 >$. We extend the definition of mapping for a leaf-sequence by taking the sequence of the matched nodes in the other tree i.e. $M(L(7)) = M(< 1, 3, 6 >) =< M(1), M(3), M(6) >$.



Post Order Number	Leaf Sequence	Stem Sequence
1	<1>	<1,2>
3	<3>	<3>
4	<1,3>	<4,5>
6	<6>	<6>
7	<1,3,6>	<7>

Post Order Number	Leaf Sequence	Stem Sequence
1	<1>	<1,2,3>
4	<4>	<4>
5	<5>	<5>
6	<4,5>	<6,7>
8	<1,4,5>	<8>

Figure 7: Compressed trees of first mapped pair chunks in figure 5(b) and corresponding leaf and stem sequence

Recall that in a chunk, only root and leaves are mapped by M. No other internal nodes in a chunk will be in M. Our goal is to find the mappings for these internal nodes in $\hat{M}$. The following lemma states the key of our algorithm classifying the internal nodes that will not be mapped by $\hat{M}$ at all. In other words, two internal nodes can be mapped only if their leaf sequences are also mapped by M.

LEMMA 2. *For a node u in S, if there is no node in T with the leaf-sequence $M(L(u))$, then u is not mapped by $\hat{M}$.*

PROOF. For contradiction let u is matched with v in T (see figure 6(b)). Since v has a different leaf-set from $M(L(u))$, there is at least one map $(w, M(w))$ such that either $w \in L(u)$ and $M(w) \notin L(v)$ or the vice versa. Without losing generality, assume $w \in L(u)$. Since u is an $anc(w)$, v has to be an $anc(M(w))$ according to the definition of mapping. Since $M(w) \notin L(v)$, by construction, there is a $x \in L(v)$ which is also $anc(M(w))$. Since x is a leaf node of a chunk, there has to be $(y, x) \in M$ such that y is a $anc(w)$ and a $desc(u)$. This leads to contradiction since no chunk has an internal node mapped by M. $\square$

There can be multiple nodes in the same tree having the same leaf-sequence and nodes with the same leaf-sequence form a path in a tree. Based on this observation, we can compress S and T by collapsing paths to single nodes. Figure 7 shows the compressed trees of the first pair of chunks in figure 5(b), where nodes in a single path with the same leaf sequence are shown as the label of the corresponding edge. We call a collapsed path a *stem*.

DEFINITION 5. *A stem is a subsequence P of the nodes in the postorder sequence of a tree such that $\forall_i L(P_i) = L(P_1)$ where P_i is the ith node in P. $L(P)$ is defined to equal $L(P_i)$.*

The table in figure 7 shows the stem and leaf sequence of all the nodes of the two compressed trees. We are now required to find the pairs of stems from the two trees having leaf sequences mapped from one to the other. We need to do it efficiently without checking all possible pairs of stems. We argue that, one parallel scan through S and T in post-order is sufficient.

Our algorithm parallely scans the nodes in trees S and T in post-order. Assume the algorithm is currently looking at two nodes u and v from S and T, respectively. If their leaf-sequences are mapped by M, we compute the mapping between their stems in a way described later. If their leaf-sequences are not mapped by M then, we can skip *either u or v* and advance the scan with the confidence that the skipped node will never be mapped by an $\hat{M}$. The lemma 3 justifies this decision. Note that the parallel scan requires at most $|S| + |T| - 1$ checks for pairs of stems.

LEMMA 3. *Let u be a node in S. Let p_l and p_r be the leftmost and the rightmost nodes in $M(L(u))$ in the tree T, respectively. Also let v be a node in T and, q_l and q_r are the leftmost and rightmost nodes in $L(v)$ in the tree T. If $M(L(u)) \neq L(v)$ then*

1. *if $p_r > q_r$ or ($p_r = q_r$ and $p_l < q_l$), then for no node $x > u$, $M(L(x)) = L(v)$.*
2. *If $q_r > p_r$ or ($q_r = p_r$ and $q_l < p_l$), then for no node $x > v$, $M(L(u)) = L(x)$.*
3. *no other case occur.*

PROOF. If x and y are any two nodes and $x > y$ (i.e. x is to the right of y) then only one of the following is true[2].

$\star$ x is an ancestor of y and, therefore, $L(y)$ is a subsequence of $L(x)$. $L(y) \subseteq L(x)$

$\star$ At their least common ancestor, x is in a right subtree to y and, therefore, they have no common subsequence. $L(x) \cap L(y) = \emptyset$.

1. Note that $M(L(u)) \neq L(v)$. The given condition essentially describes three possible scenarios as shown in figure 8(a-c). For any node $x > u$, there can be two cases.

   $\star$ If $L(x) = L(u)$ then trivially $M(L(x)) \neq L(v)$.

   $\star$ If $L(u) \subset L(x)$ or $L(u) \cap L(x) = \emptyset$ then there is a leaf $t \in L(x)$ where $M(t) \notin M(L(u))$. Now $M(t)$ can be in two possible places.

   - $M(t) < p_l$: Definitely $p_r \in M(L(x))$. Since $M(t) \notin L(v)$ (in figure 8(b-c)) and $p_r \notin L(v)$ (in figure 8(a)), therefore $M(L(x)) \neq L(v)$.
   - $M(t) > p_r$: Trivially $M(t) \notin L(v)$, therefore $M(L(x)) \neq L(v)$.
2. Since M is minimal, using the inverse mapping M' and similar arguments as above we can prove that for no node $x > v$, $M(L(u)) = L(x)$.

[2]We are abusing the set operations for sequences. We believe the context clarifies the intended meaning.

3. For any two nodes in a tree it is not possible to have both $L(x) \cap L(y) \neq \emptyset$ and $L(y) \nsubseteq L(x)$. Therefore, the remaining cases as shown in the figure 8(d) cannot occur.

□

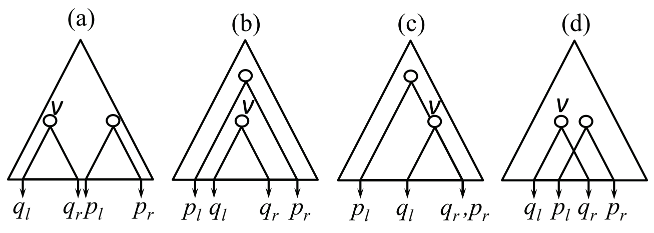


Figure 8: The tree T. (a-c) Three possible scenarios for case 1. (d) An impossible scenario which cannot occur as described in case 3 of lemma 3.

The remaining piece of the puzzle is to find the optimal mapping between the stems having leaf-sequences mapped from one to the other. The following lemma describes how we compute the optimal mapping and the distance as well. Here, SED stands for string edit distance [19]; The proof of this lemma is skipped for brevity as it is a straight forward specialization of the tree edit distance for paths (see [25] for details).

LEMMA 4. *If P^S and P^T are two stems of S and T, respectively, such that $M(L(P_1^S)) = L(P_1^T)$, then $TED(P^S, P^T) = SED(P^S, P^T)$.*

Using the above lemmas 1 to 4 we have designed our algorithm 1 for computing STED. $STED(S, T, M)$ takes in two trees S and T and divides them into chunks. For each mapped pair of chunks, the algorithm parallelly scans to see if there is any pair of stems with mapped leaf sequence. For mapped stems, the algorithm computes the string edit distance of the stems and add the value to the total cost. For non-mapped pair of stems, the algorithm adds the length of one of the stems that is guaranteed to remain unmapped in $\hat{M}$. Note that, adding length of the stem is equivalent to insertion/deletion of all of the nodes in the stem.

The running time of the proposed algorithm is $O(n^2)$ with the requirement of a minimal seed mapping M. In the worst case, when both of the trees are simple paths the algorithm costs exactly $O(n^2)$ time to compute the string edit distance where n is the number of nodes in the trees. Note that, the standard tree edit distance is at least $O(n^3)$ [6] and, therefore, our algorithm is faster than the generalized tree edit distance by at least an order of magnitude (i.e. a factor of n) while being more meaningful as well.

4.2 Content Based Diversity

In the previous section, we have described how the mapping induced by the query can be used to compute accurate and efficient distances for diversification. If we use STED for the matches in figure 1, both (1, 3) and (2, 3) produce distance values of 1 and, consequently, result in a tie. Because (1,3) involves the same person (i.e. "Michalis") while (2,3) does not, the obvious choice for the diverse pair is (2,3). Now the question is, how we can modify STED to capture true diversity by breaking such tied situation.

The answer is, by taking the contents (i.e. nodes in the document that are structurally unrelated to the query) into consideration. Contents can create different levels of differences between matches even if the matches are structurally similar to each other. For example, in figure 1 the first two matches are PhDThesis records linked to Faloutsos, but their authors are different.

When two nodes of a map (s_i, t_j) have the same label (i.e. $label(s_i) = label(t_j)$), under the equal cost model no cost is added to the total. There can be differentiating features in the branches of the subtrees S_i and T_j that are not matched to the query and, hence are ignored by STED. For example, First Name/Michalis is a branch of Author in match 1 which is not matched to any part of the query. Let S_i^R and T_j^R are the two trees rooted at s_i and t_j that contain the remaining branches unmatched to the query. We add a $correction$ cost $c \in [0, 1]$ as a cost of the map (s_i, t_j) to capture the amount of mismatch present in S_i^R and T_j^R.

The correction cost c can trivially be computed by simply taking the $TED(S_i^R, T_j^R)$ and normalizing by the maximum possible distance between a pair of matches. However, TED is too costly to use for computing the fractional contributions from the contents just to break the ties. We develop a novel approach to obtain the correction cost c efficiently.

At first, we classify nodes of an XML document in one of the four categories: *value, attribute, entity* and *connector*.

⋆ All leaf nodes are *Value* nodes.
⋆ A parent of a value is an *Attribute*
⋆ A parent of an attribute is an *Entity* if it is not an attribute itself.
⋆ A node other than the above three is a *Connector*.

Similar classification has been proposed in [20] when the *Document Tree Descriptor (DTD)* is not available. We scan the documents once to identify the type of every node. For example in figure 1, there are four attributes; School, First Name and Last Name, Title and, three entities; PhDThesis, Author and Paper.

The four classes of nodes are defined keeping the usual structure of an XML document in mind. In general, an attribute (similar to a "variable" in programming languages) has exactly one value and no other child. Therefore, attributes do not require the above mentioned correction cost as their values are always compared. In contrast, entities generally have multiple attributes and may need some correction cost. Since connectors have no attribute/value, having correction cost for them is not meaningful.

To compute the correction cost for entities, we only consider the number of mismatched attributes. Two attributes are mismatched if they have the same label but different values. For example, in figure 1 the entity Author in all the documents has two attributes First Name and Last Name. While comparing the two Author nodes in 1 and 2, the number of mismatched attributes is 1 because of the different first names. If an attribute is present in only one entity and absent in the other, it does not confirm any difference between the entities and, therefore, these attributes are not counted as mismatch [21]. For example, had there be a Middle Name attribute for the Author entity in match 1, the number of mismatched attributes would still be 1.

We define the correction cost for entities as below.

$$c_e = \frac{\text{Number of mismatched attributes}}{\text{Total number of distinct attributes}} \qquad (2)$$

Here, the total number of attributes is a normalization constant. Examples of correction costs for entities: $c_{Author} = 0.5$ for the match pairs (1, 2) and (2, 3).

Let us revisit the problem of breaking ties for the matches in figure 1. (1, 3) has a distance of 1 and (2, 3) has a distance of 1.5 when the above defined correction costs are used with the equal cost model. Thus, adding content awareness breaks the tie meaningfully in favor of the true diverse set of matches.

4.3 Algorithm for STED

Algorithm 1 shows the pseudocode for finding STED between two matches S and T when the minimal mapping M is given. Consider two matches S and T of figure 4 as the inputs of algorithm 1. The algorithm creates all the chunks as shown in figure 5(b) at lines 1 and 2 using the algorithm 3 . For each pair of mapped chunks, we initiate two pointers n and m (lines 5-6) that iterate through the chunks in their respective postorder sequence. The algorithm also computes (using the algorithm 5 at line 7-8) two sequences (i.e. arrays), B and E, that store the beginning and ending leaves of the leaf-sequences. For example, B_i and E_i are the beginning and ending leaves of $L(i)$.

At every iteration, the stems of the of nodes n and m are found (lines 11-12) by the algorithm 2. Algorithm 2 creates and returns the stem of node n by concatenating nodes with the same leaf-sequence as $L(n)$ in the post-order of C. The algorithm also returns the first node after n with different leaf-sequence.

When the stems are ready, the algorithm 1 checks to see if the stems have mapped leaf-sequences (i.e. the beginning and ending leaves are same). The algorithm handles the pair of stems with mapped leaf-sequences in two different ways (lines 13 and 15) based on the first node of the stem. If the first nodes (i and j) are leaves, by the definition of chunks they are matched to the query nodes by the query processor and we want to preserve their mapping. Note that, if i is a leaf, so is j and vice versa. To preserve the mapping between the leaves, the algorithm computes string edit distance for the rests of the stems and add the cost for the mapping of the leaves (line 16). When i and j are not leaves, the algorithm simply takes the string edit distance between the stems.

When the leaf-sequences are not mapped, there can be two cases as described in the lemma 3. In the first case, the node i remains active for the next iteration but stem the P^T of the node j is inserted/deleted (line 18). In the remaining case, the node j remains active and P^S is inserted/deleted (line 20).

Figure 9 shows the iterations of the loop at line 4 for the first pair of chunks in figure 5(b). The final $STED(S, T, M)$ is 6 which is equal to the last entry of the tree matrix in figure 5(a).

Algorithm 1 $SeededTreeEditDistance(S, T, M)$

```
Require: S and T are two trees, $M : S \rightarrow T$ is a minimal mapping
Ensure: Return the seeded tree edit distance
 1: $C^S \leftarrow Chunks(S, M)$//algorithm 3
 2: $C^T \leftarrow Chunks(T, M)$
 3: $sum \leftarrow 0$
 4: for each pair $(C^S_u, C^T_{M(u)})$ do
 5:     $n \leftarrow$ first node of $C^S_u$ in post-order
 6:     $m \leftarrow$ first node of $C^T_{M(u)}$ in post-order
 7:     $B^S, E^S \leftarrow LeafSequences(C^S_u)$
 8:     $B^T, E^T \leftarrow LeafSequences(C^T_{M(u)})$
 9:     while $n$ and $m$ are not nil do
10:         $i \leftarrow n, j \leftarrow m$
11:         $P^S, n \leftarrow FindStem(n, C^S_u)$
12:         $P^T, m \leftarrow FindStem(m, C^T_{M(u)})$
13:         if $B^T_j = M(B^S_i), E^T_j = M(E^S_i)$ and $i,j$ are not leaves then
14:             $sum \leftarrow sum + SED(P^S, P^T)$
15:         else if $B^T_j = M(B^S_i), E^T_j = M(E^S_i)$ and $i,j$ are leaves then
16:             $sum \leftarrow sum + SED(P^S - i, P^T - j) + cost(i, j)$
17:         else if $M(E^S_i) > E^T_j$ or $(M(E^S_i) = E^T_j$ and $M(B^S_i) < B^T_j)$
            then
18:             $sum \leftarrow sum + |P^T|, n \leftarrow$ i
19:         else
20:             $sum \leftarrow sum + |P^S|, m \leftarrow$ j
```

Algorithm 2 $FindStem(n, C)$

```
Require: A chunk $C$ and a node $n$ in $C$
Ensure: Return the stem $P$ and the next $n$ after the stem
 1: $i \leftarrow n, P \leftarrow \epsilon$
 2: while $L(i) = L(n)$ and $n$ is not nil do
 3:     $P \leftarrow Concatenate(P, n)$
 4:     $n \leftarrow$ next node of $C$ in post-order
```

Algorithm 3 $Chunks(S, M)$

```
Require: A tree $S$ and a minimal $M$ mapping to or from $S$
Ensure: Return $C$, a set of chunks of $S$
 1: $C \leftarrow \emptyset, Q = \{x | x \in V(S)$ and $\exists_y[(y, x) or (x, y) \in M]\}$
 2: for each $u$ in $Q$ do
 3:     $C_u \leftarrow FindChunk(u, \epsilon, Q)$
 4:     add $C_u$ to $C$
```

4.4 Properties of the Distance Measure

The seeded tree edit distance (STED) has some elegant properties: triangular inequality, fast lower bound and upper bound.

4.4.1 Triangular Inequality

The original tree edit distance holds the triangular inequality. STED also holds the triangular inequality as long as the seed mapping is same.

$$STED(T_1, T_2, M) + STED(T_2, T_3, M) \geq STED(T_1, T_3, M)$$

PROOF. The STED can be obtained by creating a cost model from M. Since, M is fixed, and TED holds triangular inequality, STED also holds triangular inequality. □

When we add content awareness in STED, it becomes a bit complicated. If we choose to use the described definition for "mismatch" in the previous section, the triangular inequality *does not* hold. But there is a way around for performance critical applications where triangular inequality is the key for performance. If we consider absent attributes as *mismatched* attributes, triangular inequality *holds* with the given definition of correction cost.

4.4.2 Lower Bounds

Another desirable property of a distance measure is the availability of low cost lower bounds for fast similarity search. There is a simple lower bound for STED that requires $O(n)$ time for computation while STED itself requires $O(n^2)$.

STED requires computing the string edit distance for stems with mapped leaf-sequences. A trivial lower bound for String edit distance is the absolute difference between the lengths of the strings being compared. If we use such trivial bounds whenever STED needs a string edit distance, the resulting distance value is a lower bound to the original STED.

4.4.3 Upper Bounds

Similar to lower bound, an upper bound of STED can be computed by taking the sum of the lengths of the two stems used in string edit distance computations. Such an upper bound also requires $O(n)$ time for computation.

Algorithm 4 $FindChunk(n, C_u, Q)$

```
Require: A node $n$, the list of mapped nodes $Q$ and the current chunk $C_u$
    to add in
Ensure: Return the modified current chunk $C_u$
 1: add $n$ to $C_u$
 2: if $C_u = \epsilon$ or $n \notin Q$ then
 3:     for each child $v$ of $n$ in left to right order do
 4:         $C_u \leftarrow FindChunk(v, C_u, Q)$
```

Algorithm 5 $LeafSequences(S)$

```
Require: A tree $S$
Ensure: Return two arrays $B$ and $E$ containing the start and end nodes of
    the leaf sequences of every node in $S$
1: $B \leftarrow \epsilon, E \leftarrow \epsilon$
2: $FindLeafSequence(Root(S), B, E)$
```

Algorithm 6 $FindLeafSequence(u, B, E)$

```
Require: A node $u$ and two arrays $B$ and $E$ to store the start and end of
    $L(u)$
1: if $u$ is a leaf then
2:     $B_u \leftarrow u, E_u \leftarrow u$
3: else
4:     for each child $v$ of $u$ do
5:         $FindLeafSequence(v, B, E)$
6:     $i \leftarrow$ leftmost child of $u$
7:     $j \leftarrow$ rightmost child of $u$
8:     $B_u \leftarrow B_i, E_u \leftarrow E_j$
```

5. DIVERSIFICATION

Result set diversification is an NP-hard problem. Many heuristics [12] have been proposed to find approximate diverse result set (greedy heuristic, interchange heuristic, clustering heuristic, etc.). In this paper we utilize the greedy heuristic algorithm [13] (see algorithm 7) which selects a seed of one or two matches (line 1). Once the seed is selected, the algorithm finds the next object to add in the final result set (line 4-5). To do that, the algorithm compares each of the remaining matches to the already added matches in the result set and add the one that has the maximum total distance to the current result set. The algorithm stops once k matches are added to the result set (line 3). The algorithm computes linear number of editing distances on the number of matches ($|(\mathcal{T})|$) as $k << |\mathcal{T}|$. We consider three methods *Diameter Seed* [13], *Lower Bound Seed* and *Random Seed* for selecting the seeds.

- ***Diameter Seed:*** Select the farthest pair of points (diameter) in the set of matches (i.e. $\mathcal{T}$) as the seed. Finding the diameter is inherently quadratic in time complexity for high dimensional data.
- ***Lower Bound Seed:*** Select the farthest pair of points by using the lower bound (as described in section 4.4) instead of the true tree edit distance. This approach is also quadratic but promises to be faster.
- ***Random Seed:*** Select one match as the seed at random. This approach is efficient but suffers degradation in quality (see section 6).

Algorithm 7 $Greedy-Diversification(\mathcal{T}, K, Algo)$

```
Require: A set of matches $\mathcal{T}$, the final result set size $k$
Ensure: Return the set $R$ of top-$k$ diverse matches from $\mathcal{T}$
1: $R \leftarrow$ initial seed(s)
2: $\mathcal{T} \leftarrow \mathcal{T}$-$R$
3: while $|R| < k$ do
4:     find $T_i$ in $\mathcal{T}$ such that the total pair-wise distance of
       $R \cup T_i$ is maximum for all $T_i \in \mathcal{T}$
5:     $R \leftarrow R \cup T_i, \mathcal{T} \leftarrow \mathcal{T}$ - $T_i$
```

In figure 10, we demonstrate the trends of the seed selection algorithms as the number of matches increases. We compare the running time of the algorithms with that of a standard query processor (LCS-Trim [22]). As the figure suggests, the curves are diverging and therefore, the motivation of having a diverse result set no longer worth the waiting time after the matches are available

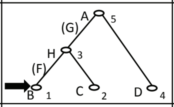
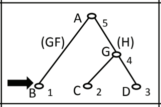
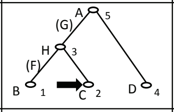
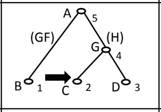
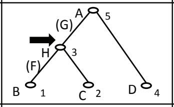
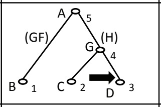
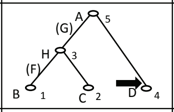
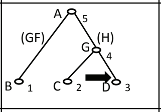
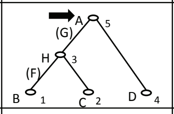
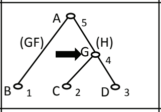
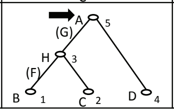
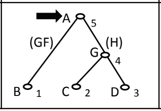
Iteration	C_1^S	C_1^T	Comment	STED
1			• Nodes are leaves and their leaf sequences are mapped •Find SED of stems (line 15) •Advance the pointers	1
2			Same as iteration 1	0
3			• Leaf Sequences are not mapped • Delete HG from C_1^S (line 20) • Advance the pointer of C_1^S only.	2
4			Same as iteration 1	0
5			• Leaf Sequences are not the mapped • Delete GH from C_1^T (line 18) • Advance the pointer of C_1^T	2
6			• Leaf Sequences are mapped but nodes are not leaves. •Find SED of stems (line 14) •Advance the pointers	0

Figure 9: Iterations performed by the algorithm 1 for the first pair of chunks in figure 5(b).

from the query processor. Clearly we need an efficient diversification algorithm taking sublinear time with the increasing number of matches.

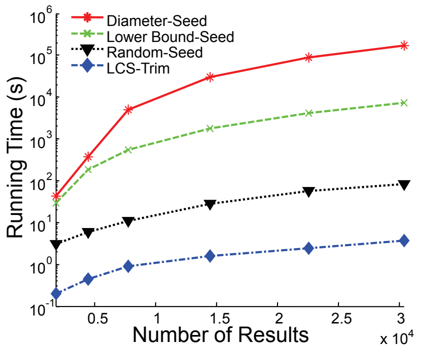


Figure 10: Comparison of running times of different diversification algorithms with a sample query processor, LCS-Trim.

5.1 Novel Heuristic for Seed Selection

As we have discussed random-seed linear time diversification algorithm improves the running time but degrades the quality, which motivates to propose a new and fast heuristic for seed selection, so to have similar time complexity as random-seed while improve the overall quality of diverse result set.

We propose a new scoring technique for selecting the initial seed. Instead of a random seed, we want to start from one of the matches which have an extreme value for a relevant but low cost feature. One such feature is the count of nodes in a match. Counting nodes for every match and selecting the one with the maximum count takes one linear scan over the matches. Note that, this process does not require any distance computation. We name this selection method as *QMax*.

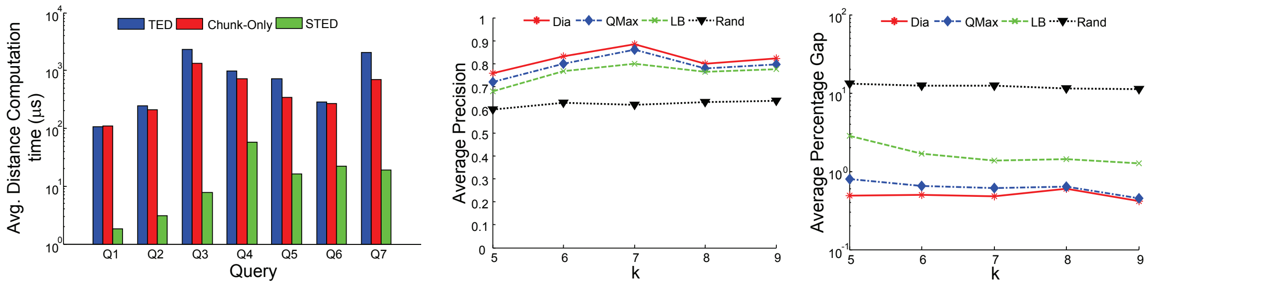
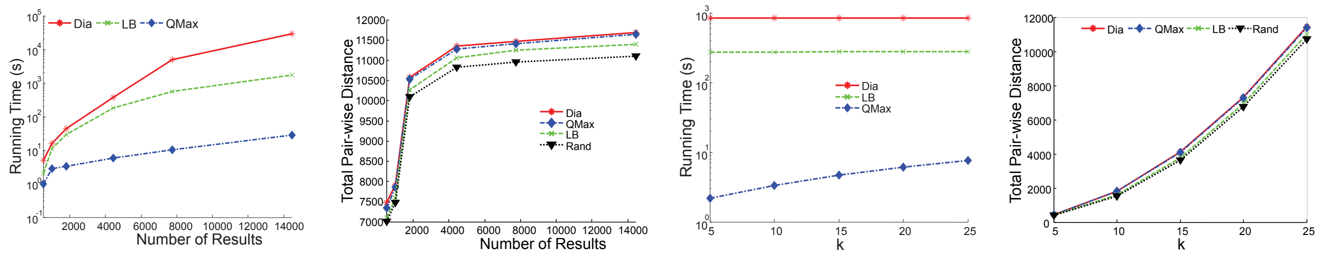


Figure 11: (a) Average distance (structure and content) computation time between two results, and (b) Average Precision vs k (c) Average Distance Gap vs k for different diversification algorithms



(a) $k = 25$

(b) $k = 25$

(c) $|(\mathcal{T})| = 5000$

(d) $|(\mathcal{T})| = 5000$

Figure 12: Experimental Results for the Query Q_3

6. PERFORMANCE EVALUATION

To experimentally demonstrate the utility of our algorithms, we have used the *Treebank* dataset[3] because of its rich structural variations. We have selected seven queries (table 1). The queries are structurally different from each other to cover several extreme cases. The experiments are performed in a standard unix system on a 2.10 GHz processor and 4GB of RAM.

Query	XPath Expression	Matches
Q_1	$//S[/VP/NP][/VP]$	11752
Q_2	$//EMPTY//X/VP/PP//NP$	1412
Q_3	$//S[//NNS][//JJ][//VP//NNP]$	30349
Q_4	$//EMPTY//S/VP/*/SBAR//PP//NN$	28463
Q_5	$//S[//DT][/VP//PRP\_DOLLAR\_]$	4559
Q_6	$//S[/NP][/VP/*/PP//NNP]$	35326
Q_7	$//S[//NP/NNP][//CC][/*/VP[/VBZ]$ $[//NP/\_NONE\_]]$	906

Table 1: Query Set

6.1 Speedup of STED

Our first experiment is to evaluate the performance of STED in comparison with the generalized tree edit distance that uses a modified cost model to preserve the seed map. We also use an intermediate algorithm which divides the trees into chunks as STED but, computes regular tree edit distances (with the modified cost model) for every pair of chunks. Figure 11(a) shows the average time taken to compute the distance between two results for the queries in table 1. Note that all three methods compute the same distance as output, thus the quality of the distance measures are equal. However, STED performs at least two orders of magnitude faster than the tree edit distance while the *chunk-only* version achieved notable amount of speedup demonstrating the importance of the our chunking approach.

6.2 Evaluation of Diversification Algorithms

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

We compare the seed selection algorithms, Diameter Seed (***Dia***), Lower Bound Seed (***LB***) and Random Seed (***Rand***), against our proposed heuristic, ***QMax***. Note that, Dia and LB require quadratic number of distance computations for seed selection, while the $Rand$ and $QMax$ need no distance computation for seed selection.

6.2.1 *Qualitative Analysis*

For qualitative analysis, we compare the final result sets returned by different algorithms with the result set generated by the optimal (brute force) algorithm for the same query and input parameters. We use two criteria to measure the quality, *precision* and *percentage gap*. Precision of an algorithm is the fraction of the optimal top-k matches that the algorithm returns. Percentage gap is the percentage of the deviation of the total pair-wise distances of an algorithm from that of the optimal algorithm. For the efficiency of the brute force algorithm, the number of candidate results (N) is fixed to be 100 (figure 11(b),11(c)). In both the measures $QMax$ performs better than LB and $Rand$, and very close to Dia.

6.2.2 *Scalability Analysis*

Our next experiment is to evaluate the scalability of the diversification algorithms as the number of candidate results and k increase. We have shown the experiments for the query Q_3, since it is complicated in structure and can generate a wide range of structurally diverse results.

In figure 12(a), we show the running times of the algorithms to produce top-25 diverse results for different sizes of result sets. Clearly $QMax$ outperforms the quadratic algorithms Dia and LB (figure 12(a)). The curve for Rand is skipped for visual clarity as it overlaps the curve for $QMax$.

In figure 12(b), the total pair-wise distances of the top-25 diverse matches are shown for different algorithms. In both figures 11(c) and 12(b), $QMax$ achieves insignificantly less accurate results compared to Dia. Reader may interpret this little loss on accuracy as the price paid for the huge speedup shown in figure 12(a). In practice, the small difference in the total distance does not add subjectively noticeable changes in the reported output.

We have also studied the running time and the total pair-wise distance of the algorithms for different values of k for a fixed result set size (figure 12(c) and 12(d)). Quadratic seed selection methods (Dia and LB), need so large an amount of time for selecting the seed that the rests of the algorithms (with complexity $O(k|(\mathcal{T})|)$) negligibly increase the total time (figure 12(c)). In contrast, $QMax$ selects the seed very fast and therefore, the running time for $QMax$ linearly increases with k. This ensures a possible adaptation of our algorithm as an *anytime* algorithm, where the user can preemptively stop the computation at any time with the best answers she could get in the elapsed amount of time.

7. CONCLUSION

We have modified the standard tree edit distance to consider the query-context and the contents for XML result diversification. We have also given a novel heuristic technique for speeding up the existing algorithms. We have experimentally validated our contributions.

Our focus has been to develop methods to diversify *exact* matches to the query. As for future research, we note that our algorithm can be extended to work on approximate matches.

In a typical approximate query matching system [3][4], the query is perturbed to generate few approximate queries. Later, *exact* matches for these approximate queries are found. Let S_Q denotes the set of approximate queries of Q. Also let the exact result set for a query $q \in S_Q$ be $\mathcal{T}_q$. Finding the top-k diverse results for Q can then be done in two steps. In the first step, we compute top-k diverse results for each $\mathcal{T}_q$ using our efficient STED and diversification algorithm. In the second step, we find the top-k diverse results from the $k|S_Q|$ matches of the first step. As two matches for two different queries in S_Q may not have a complete mapping between them, we use the modified tree edit distance described in 4 instead of STED. The number of distances computed in the second step is not significant as $k|S_Q|$ is in the order of hundreds.

All the approximate matches are not equally relevant to the query, which necessitates to use relevance score in diversity calculation. We can use relevance score in [4] or any other function suitable for this purpose.

8. REFERENCES

[1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *Proc. ACM WSDM*, 2009.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.

[3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.

[4] S. Amer-Yahia, L. V. Lakshmanan, and S. Pandit. Flexpath: Flexible structure and full-text querying for xml. In *SIGMOD*, pages 83–94, 2004.

[5] T. Aratsu, K. Hirata, and T. Kuboyama. Approximating tree edit distance through string edit distance for binary tree codes. In *SOFSEM*, 2009.

[6] N. Augsten, D. Barbosa, M. Bohlen, and T. Palpanas. Tasm: Top-k approximate subtree matching. In *ICDE*, 2010.

[7] N. Bruno. Holistic twig joins: Optimal xml pattern matching, 2002.

[8] D. Chamberlin. Xquery 1.0: An xml query language.

[9] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[10] J. Clark. Xml path language (xpath) version 1.0.

[11] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. Divq: Diversification for keyword search over structured databases. In *SIGIR*, 2010.

[12] M. Drosou and E. Pitoura. Comparing diversity heuristics. In *Technical Report. Computer Science Department, University of Ioannina*, 2009.

[13] M. Drosou and E. Pitoura. Diversity over continuous data. *IEEE Data(base) Engineering Bulletin*, 32:49–56, 2009.

[14] M. Drosou and E. Pitoura. Search result diversification. In *ACM SIGMOD Record*, 2010.

[15] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[16] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate xml joins. In *SIGMOD*. ACM, 2002.

[17] M. Hadjieleftheriou and V. J. Tsotras. (eds.) Result Diversity. *IEEE Data Engineering Bulletin*, 32(4), 2009.

[18] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed xml documents. In *Proc. of VLDB*, pages 273–284, 2003.

[19] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–10, 1966.

[20] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, pages 329–340. ACM, 2007.

[21] Z. Liu, P. Sun, and Y. Chen. Structured search result differentiation. In *VLDB*, 2009.

[22] S. Tatikonda, S. Parthasarathy, and M. Goyder. Lcstrim: Dynamic programming meets xml indexing and querying. In *VLDB*, 2007.

[23] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[24] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[25] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *Siam Journal on Computing*, 18:1245–1262, 1989.