An Efficient Method for Extracting Subtrees Against Forest Query

Shafaet Ashraf
Department of Computer Science and Engineering
University of Dhaka
Bangladesh
shafaet.csedu@gmail.com

Sheikh Muhammad Sarwar
Institute of Information Technology
University of Dhaka
Bangladesh
smsarwar@du.ac.bd

Md. Abeed Hassan
Department of Computer Science and Engineering
University of Dhaka
Bangladesh
abeedcsedu@gmail.com

Dr. Saifuddin Md. Tareeq
Department of Computer Science and Engineering
University of Dhaka
Bangladesh
smtareeq@cse.univdhaka.edu

Anna Fariha
Department of Computer Science and Engineering
University of Dhaka
Bangladesh
anna@cse.univdhaka.edu

ABSTRACT

In this paper, we present an algorithm to search and rank *top-k* approximately matched subtrees from a tree database, where the query is a collection of trees *i.e.* a forest. Even though existing algorithms can handle a single tree query, we argue that forest query would be significantly useful in some real life applications including biological domain. To address the issue we have proposed a method to find relevant subtrees and rank those given a tree database and a forest query. Tree edit distance is used to find and rank a set of subtrees with a pruning technique to improve the performance of the algorithm. We have tested our algorithm on different data sets and the efficiency of the searching and ranking process show promising results. Experimental results suggest that our algorithm improve run time at this stage and in future we would like to make it more useful for practical large data set.

IMCOM '15, January 8-10, 2015, Bali, Indonesia

ACM 978-1-4503-3377-1/15/01$15.00
http://dx.doi.org/10.1145/2701126.2701195 .

Categories and Subject Descriptors

H.3.3 [[**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithm, Data Structure

Keywords

XML Retrieval, Forest Query

1. INTRODUCTION

There is a growing need of subtree matching algorithms in various fields. It is particularly useful for XML documents [1]. It is used to join heterogeneous XML documents when two documents with different structures convey approximately the same information [2, 3]. Tree matching algorithms can be used for matching diverse data structures like molecular networks or SQL data types modelled as a graph [4, 5]. In many of the cases finding exact matching is not possible but approximate matching can serve the purpose. Thus, an efficient algorithm for finding approximate matching is important [6]. State-of-the-art algorithms can find *top-k* subtrees from tree database in polynomial time [1, 6], but handling query as a forest is still an issue that haven't been addressed yet.

In any kind of text search engine, query usually takes the form of a list of keywords. The search engine finds a set of documents that matches all or a subset of the keywords. After that it ranks those set of documents under certain criteria. If query keywords appear more closely in a document, it is considered more relevant and hence ranked higher in

the result set. Analogous to that searching inside an XML document it is very likely that the query can come as a forest of trees instead of multiple keywords. For example, a user may search for some books in a book store that categorizes and stores their content as an XML file. The XML database may look like the one shown in figure 1. Now, suppose that a user wants to search for two books and naturally the query may take the form of a forest as shown in figure 2.

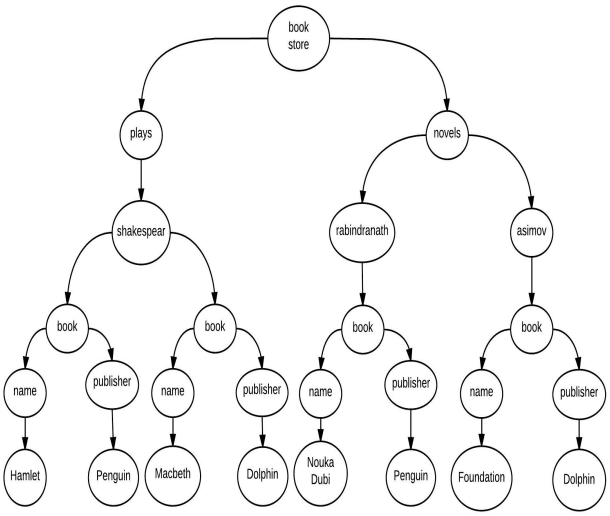

Figure 1: Sample Database

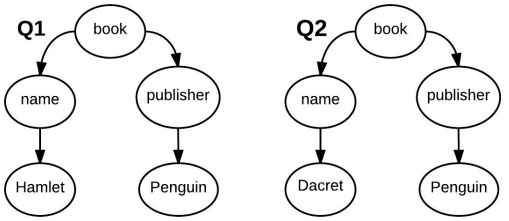

Figure 2: Sample Forest Query

In this scenario, exact match is not enough, approximate matching can be useful to the user too. Existing algorithms can find approximate matching only when query comes as a tree but they don't support approximate matching against forest query. In this paper, we present an algorithm that takes forest query as its input and returns a collection of sub-trees extracted from a tree structured document. We rank those collection of sub-trees on the basis of *tree edit distance* and *compactness*. Tree edit distance reflects how closely each tree component of the query matched a sub-tree in the tree structured document. Compactness reflects closeness of resulting subtrees in the document and it is analogues to proximity in keyword search.

2. BACKGROUND AND RELATED WORK

2.1 Basic Definitions

2.1.1 Tree

A tree is a connected acyclic graph. Each two vertices of the graph are connected by exactly one path. A tree contains V vertices and E edges where $E = V - 1$. A tree is called a rooted tree if one vertex has been designated as the root. All other nodes have orientation away from the root. In this section, each reference to tree will refer to a rooted ordered connected acyclic tree.

2.1.2 Forest

A forest is an acyclic graph. Forests therefore consist only of (possibly disconnected) trees and hence named as forest. A forest $F = (T_1, T_2, ..., T_n)$ is a disjoint ordered union of trees. A forest is ordered and rooted when all its trees are ordered and rooted.

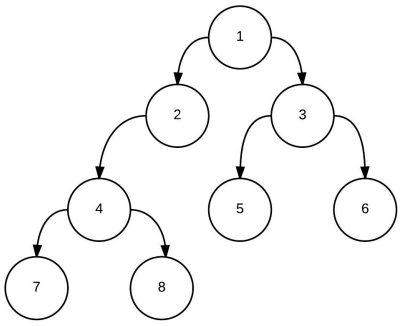

Figure 3: Rooted Ordered Tree

2.1.3 Lowest Common Ancestor (LCA)

Depth of a vertex v is defined by number of edges in the path between root vertex and v. Root vertex has zero depth by the aforementioned definition. Height of a tree T is defined by the depth of the deepest vertex. Ancestor of a node n is defined as set of nodes S_w such that each element $w \in S_w$ lies in the path from root to u. In figure 3, ancestors of vertex 4 are 2 and 1. Common Ancestor for two vertex u and v is any vertex w such that $w \in ancestor(u)$ and $w \in ancestor(v)$. **Lowest Common Ancestor** for two vertex u and v is the node w such that it has lowest depth among all the common ancestors of u and v.

2.2 Tree Edit Distance (TED) Problem

The Tree Edit Distance is defined as the minimum-cost sequence of node edit operations that transform one tree into another. Lets assume S is a sequence of edit operations $s_1, s_2..., s_k$. Lets define a cost function $\gamma(S)$ which returns the cost of the edit sequence S. Then the edit distance between T_1 and T_2 is defined by:

$\delta(T_1, T_2)$=min$\{\gamma(S) \mid S$ is an edit operation sequence taking T_1 to $T_2\}$

2.2.1 Existing Tree edit distance Algorithms

TED is an extension of string edit distance problem. String edit distance problem finds minimum number of steps to transfer a string to another string. Tree edit distance algorithm does the same for a tree. In a word it is a similarity measurement between two trees. The problem was first

introduced in 1979 by Kuo-Chung-Tai [7]. Later in 1989 Zhang-Shasha [8] proposed a dynamic programming algorithm to solve this problem. In 2012 Augusten *et al.* [9] proposed RTED algorithm which works good for any tree shape. In this work, we have used Zhang-Shasha algorithm as a subroutine and this section will be focused on that algorithm.

2.3 Top-k Approximate Subtree Matching Problem

Let T is a tree database and Q is the query. t_i denotes a subtree of the tree whose root is t_i. The problem is to find a set of subtree $S_t = t_1, t_2, ..., t_k$ such that $\text{TED}(T, t_1) \leq \text{TED}(T, t_2) \leq ... \leq \text{TED}(T, t_k)$. That means the sub-trees in the set are sorted in ascending order with respect to edit distance with the query. Figure 4 shows an example of top-k approximate subtree matching algorithm.

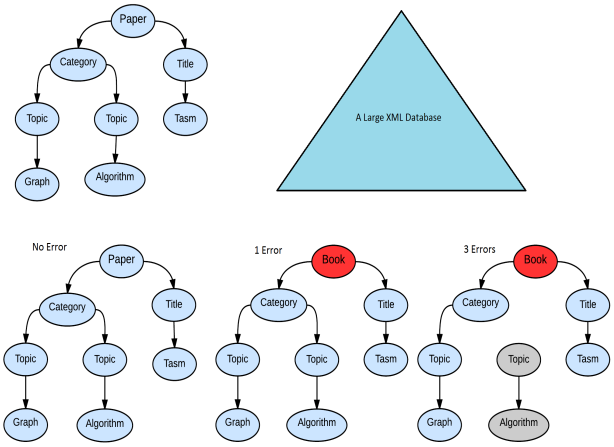

Figure 4: Top-k Approximate Subtree Matching Example

2.3.1 *Existing Top-k Approximate Subtree Matching Algorithm*

When we need to find the tree edit distance from a large XML trees it becomes difficult. Memory efficiency and scalability are very important for the edit distance. In this section, we will discuss how to extract Top-k best matched subtree from a tree database. Augsten *et al.* proposed an algorithm to compute top-k subtree [10]and Agarwal *et al.* [6] presented an algorithm which uses which establishes a one-to-one correspondence between trees and sequences. In this work, we have used the algorithm proposed by Augsten *et al.* as a subroutine of our algorithm [10].

3. PROPOSED ALGORITHM FOR PROCESSING FOREST QUERY

In this section, we formally define forest query and present an algorithm to handle forest query. Our algorithm finds at most k subtrees, which covers the forest fully or partially. We also propose a heuristic to rank the subtrees.

3.1 Definition of Forest Query

A forest query is a set Q=$\{T_1, T_2, ..., T_n\}$ where $n \geq 1$ and each $T_i \in Q$ is a ordered rooted tree. Existing algorithms can handle the query when n=1; so tree query is just a special case of forest query.

3.2 Steps of Proposed Method

Lets assume we have a tree database D, which is a rooted ordered tree. TASM algorithm [10], which takes a tree database and a query tree as parameter and returns a set consists of k trees sorted in ascending order of relevancy.

3.2.1 *Extraction of Subtrees*

Lets assume, that the input to our algorithm is a forest query Q contains n subtrees *i.e.* n components. At the first stage of our algorithm, we apply a method *getdata*, which extracts subtrees using the TASM algorithm. For each tree T_i in the query Q, *getdata* extracts subtrees using the following steps:

1. Let FS be an empty set, FS= ϕ.
2. For each $T_i \in Q$ use top-k function to get top-k relevant subtrees and append the set in FS.
3. Return the set FS.

Therefore *getdata* will return a set FS where each element of FS is also a set. Therefore, i^{th} element $fs_i \in FS$ will contain top-k relevant subtrees for $T_i \in Q$.

3.3 Finding Candidate Subtrees

As we have n trees in our query the cardinality of set FS will be n. Now, we generate all possible candidate forest from the set FS. A candidate forest is a set $R = \{T_1, T_2, T_3, ..., T_n\}$ such that $n \geq 1$ and $T_i \in FS_i$. So R is a set consisting of one subtree from each element of FS and there are k^n candidate forests as there are k elements in each element in FS. Finally, for each candidate forest, we find the candidate subtree that contains the whole forest and that subtree is the smallest subtree covering the forest is actually the lowest common ancestor of the root nodes for each subtree $T_i \in R$. Formally we can say that a candidate subtree generated from a candidate forest $R = \{T_1, T_2, T_3, ..., T_n\}$ is $LCA(r_1, r_2, ..., r_n)$ where r_i is the root node of T_i. After finding all candidate subtrees we need to rank them to find the top-k candidate subtrees and for that reason we need to compare and sort candidate subtrees.

3.4 Scoring Candidate Subtrees

In this section we define a *score* to compare two candidate subtrees to sort them. For each candidate subtree we assign it a score based on following criteria:

Similarity Measure: Each candidate subtree is generated from a candidate forest R. We use R to measure similarity. Let Q_i be the query tree, which matched i^{th} component T_i of R. Then similarity score s for a candidate forest R can be defined as below:

$$s = \sum_{i=1}^{n} TED(Q_i, T_i)$$

In the above equation TED is the Tree Edit Distance function, which is defined in 2.2. Similarity measure indicates how similar a candidate subtree is with the forest query.

Lower value of s suggests close similarity and s=0 denotes exact match.

Compactness: With similarity measure, we also consider the notion of compactness into account to assess the quality of the results. Compactness depends on root of the candidate subtree and indicates how closely each component of the candidate subtree are attached. Let R be the candidate subforest and r is the root of the candidate subtree. Compactness is calculated by following formula:

$$c = \sum_{i=1}^{n} distance(r, T_i)$$

Based on the above criteria, we have defined a total score function, which is the result of a simple addition of the similarity measure and compactness. Total Score for a subtree is simply sum of both measures.

$$score = s + c.$$

Less score means a better match while higher score means a worse match.

3.5 Search Space Pruning

A simple but effecting pruning technique is applied when we obtain k results. We keep track of the worst result found so far. When we build a new subtree we check if the score for it is worse than the worst result found so far. If it is worse, then there is no need to go in that branch anymore because that will not yield any better results. In performance analysis section we have shown that this branch and bound technique can reduce lots of branches and thus reduce running time.

3.6 An Example Showing the Execution of the Proposed Algorithm

In this section, we show a sample execution of our algorithm. We have implemented the algorithm in Python and applied the top-k approximate subtree matching algorithm (TASM) as a subroutine. TASM algorithm uses the tree edit distance algorithm by Zhang-Shasha to find approximate matching.

The sample XML document, on which we are going to search with a forest query is shown in figure 5. In the figure, the xml document is represented as a tree. Now, we show a sample forest query in figure 6 and the forest contains two trees, Q1 and Q2. For this example, we assume k=2 *i.e.* two subtrees will be returned for each component of the forest query. In figure 6, it can be seen that according to tree edit distance algorithm, best matched subtrees from the XML document for tree Q1 is M1 and M2 , while for tree Q2, P1 and P2 are the best matches. As a consequence, in all combination we would have a set R containing four results (M1,P1), (M1,P2), (M2,P1), (M2,P2). Now we will find and rank subtrees that contains both queries.

Figure 7 shows position of the matched subtrees in the XML tree database. From figure 6, it is easily observable that the subtree that contains (M1,P1) pair would have the root node shakespear, which is the LCA of M1 and P1. It is actually the candidate subtree that we have defined in section 3.4. Now, for all the elements of set R we will have to find the candidate subtrees containing them and finally rank those subtrees according to the measure we have defined in 3.4.

Figure 8 shows what happens if we choose subtree P1 and M2. Two nodes have mismatched label; one for P1 and one for M2. So, from similarity measure we find s = 2. The LCA of P1 and M2 is the root node, which is labelled as bookstore. The distance of P1 from LCA(P1,M2) is 3 and from M2 it is 3 too. Hence, the sum of distances c = 3+3 = 6. As a result, the score for pair (P1,M2) is s+c = 2+6 = 8. Now, for all the pairs in set R we calculate the scores and rank the pairs in ascending order of scores. We show the calculation process using figure 8, figure 9 and figure 10. We can also conclude that the subtree for (P1,M1) is the best match as it has the lowest score 3 (shown in figure 10).

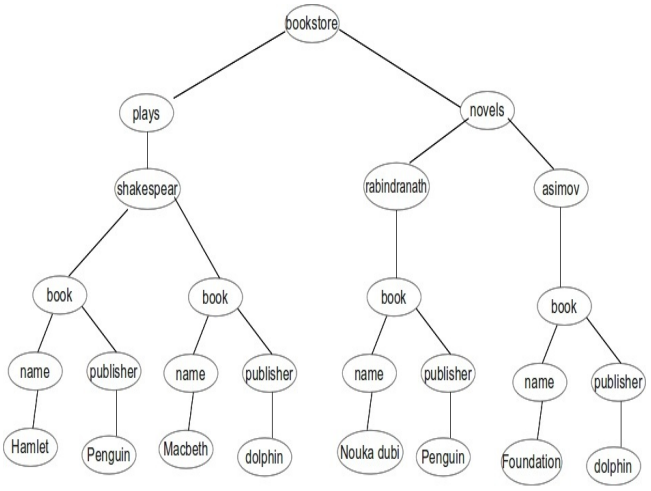

Figure 5: XML Database

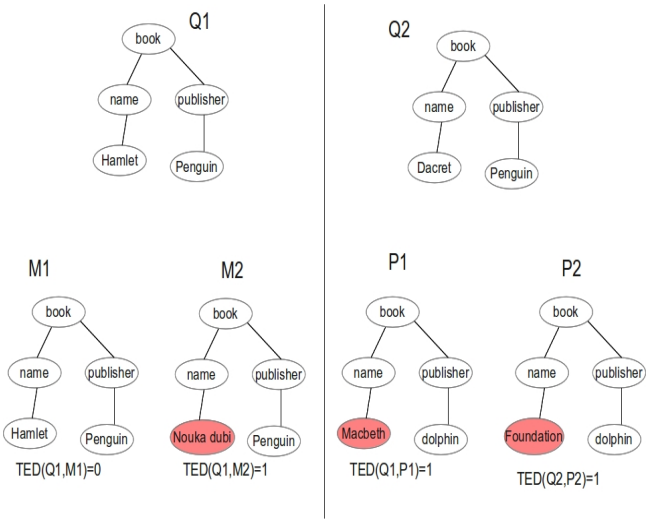

Figure 6: Query Forest and Top-k Approximate Subtree Matching for Each Query

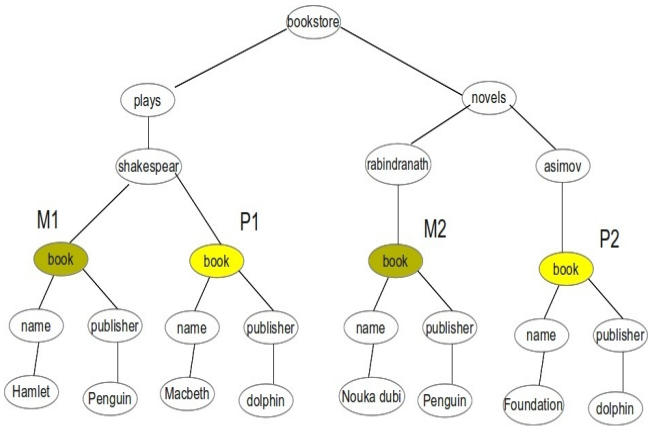

Figure 7: Position of Matched Subtrees in the XML Database

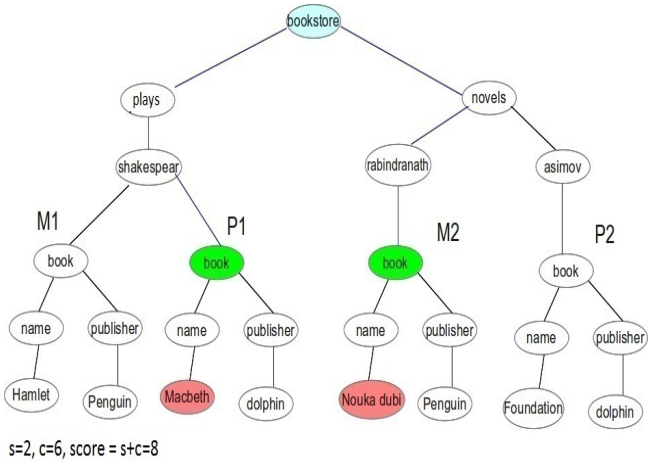

Figure 8: Selecting subtree P1 and M2 as a candidate forest.

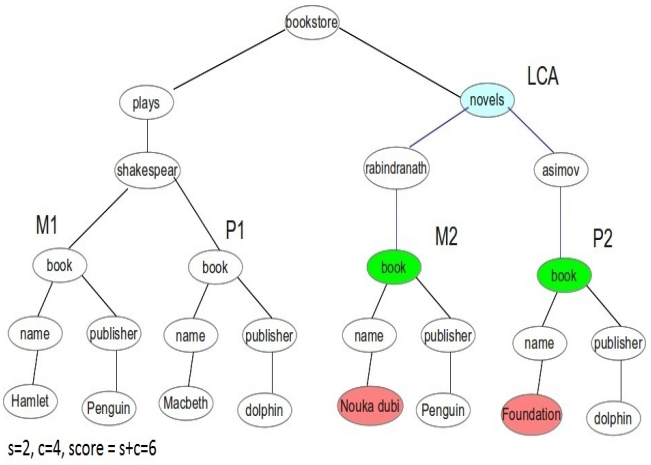

Figure 9: Selecting subtree P2 and M2 as a candidate forest.

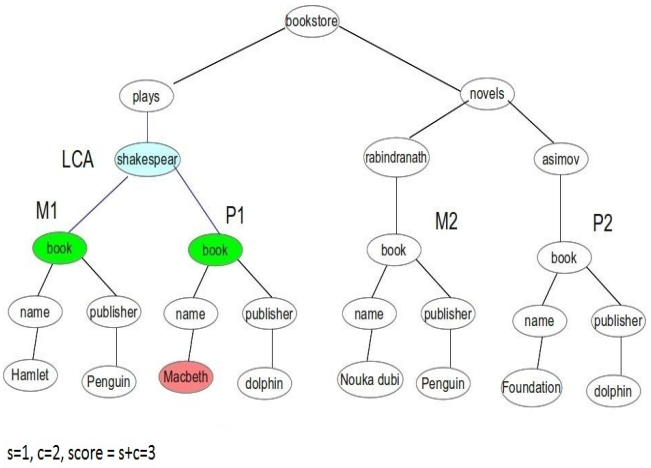

Figure 10: Selecting subtree P1 and M1 as a candidate forest.

4. PERFORMANCE ANALYSIS

In section 3.5, we have mentioned that our pruning technique will be able to reduce lots of branches and hence reduce the execution time. After implementation and experimentation with generated synthetic database we obtain results and it proves the aforementioned claim empirically. We mainly worked on two randomly generated databases, one have only 64 nodes, another have 8890 nodes. According to our experiment, total number of branches largely depend on the number of trees in the forest. We have experimented for different size trees and found that number of pruned branches increase rapidly with forest size. Note that performance doesn't depend on individual tree size of the query forest, rather it depends heavily on the number of trees in the forest as the number of candidate sub-trees increases exponentially with an increase in the number of trees in the forest.

In order to show the efficiency of our algorithm, at first, we show performance dependency on the number of trees in the query forest. Performance statistics show the number of branches pruned and the run-time analysis.

Branch Pruning: Figure 11 shows that the number of pruned branches is significantly large for a database consisting of 64 nodes and that implies the pruning is fairly efficient. In figure 12, we show the result for branch pruning on a larger database containing 8890 nodes and k = 4. In figure 13, we show the efficiency of pruning by varying the size of result set obtained for a forest query while keeping the number of forests constant.

Runtime: As this is a complete search algorithm, run time increases exponentially with number of forests in query tree. But the branch pruning has significant impact on run time. Figure 14 shows run time with and without pruning for k = 2 and 64 node database. Run time for k = 4 and 8890 node database is shown in figure 15. In figure 16, we show the run time with and without pruning by varying the

size of result set obtained for a forest query while keeping the number of forests constant.

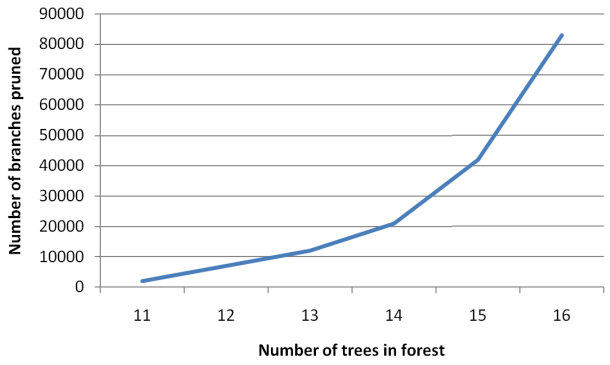

Figure 11: Branch Pruning for different query forest size. k=4, database size=64 nodes

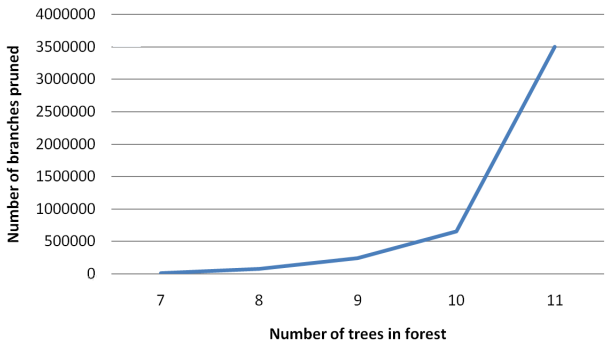

Figure 12: Branch Pruning for different query forest size. k=4, database size=8890 nodes

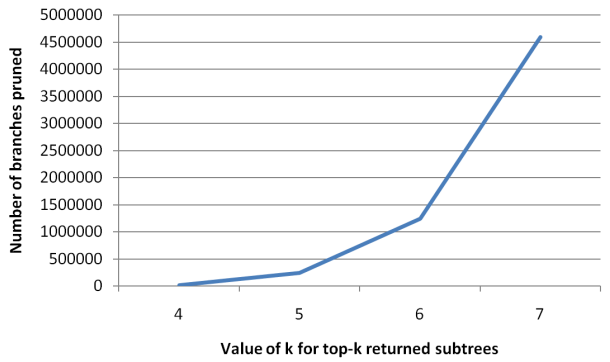

Figure 13: Branch Pruning for different k, database size = 8890 nodes, number of trees in forest query = 8

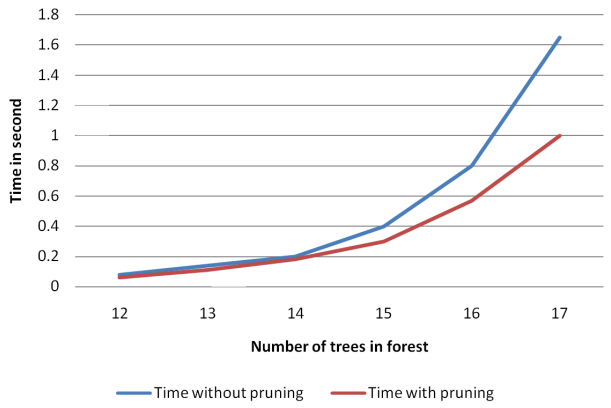

Figure 14: Runtime with and without pruning for k=2 and database size = 64 nodes

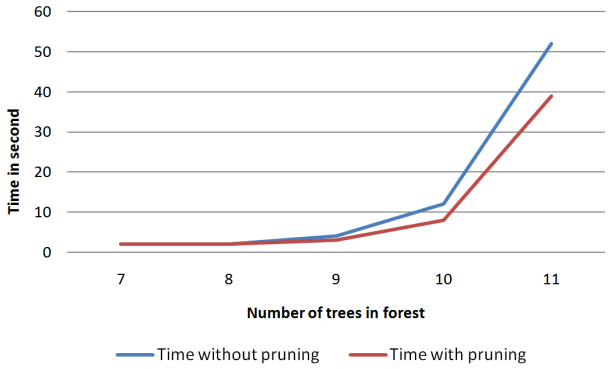

Figure 15: Runtime with and without pruning for k=4 and database size = 8890 nodes

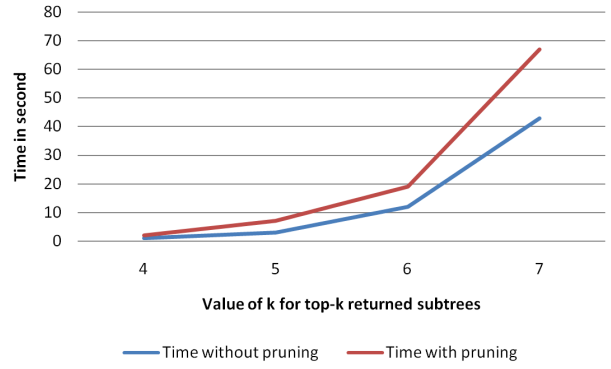

Figure 16: Runtime with and without pruning for different k. database size = 8890 nodes and number of trees in forest query = 8

5. CONCLUSION

In this paper we have introduced forest query problem in the context of top-k approximate matching. We proposed

a method to rank the subtrees according to similarity and compactness. We also proposed a pruning technique to reduce search space. In performance analysis we have shown that our pruning technique can reduce a significant amount of branch and improve run time. However, our algorithm largely depends on tree edit distance algorithm which does not provide a perfect similarity measure. Future work will be to find a better similarity measure algorithm with better pruning strategy.

6. REFERENCES

[1] Nikolaus Augsten, Denilson Barbosa, Michael H. Böhlen, and Themis Palpanas. Efficient top-k approximate subtree matching in small memory. *IEEE Trans. Knowl. Data Eng.*, 23(8):1123–1137, 2011.

[2] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 287–298, 2002.

[3] Nikolaus Augsten, Michael H. Böhlen, Curtis E. Dyreson, and Johann Gamper. Approximate joins for data-centric XML. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 814–823, 2008.

[4] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 201–212. ACM Press, 1998.

[5] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 117–128, 2002.

[6] Nitin Agarwal, Magdiel Galan Oliveras, and Yi Chen. Approximate structural matching over ordered XML documents. In *Eleventh International Database Engineering and Applications Symposium (IDEAS 2007), September 6-8, 2007, Banff, Alberta, Canada*, pages 54–62, 2007.

[7] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[8] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[9] Mateusz Pawlik and Nikolaus Augsten. RTED: A robust algorithm for the tree edit distance. *CoRR*, abs/1201.0230, 2012.

[10] Nikolaus Augsten, Denilson Barbosa, Michael H. Böhlen, and Themis Palpanas. TASM: top-k approximate subtree matching. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 353–364, 2010.