Semantic-Distance Based Clustering for XML Keyword Search

Weidong Yang and Hao Zhu

School of Computer Science, Fudan University,
Handan Road 220, Shanghai, China

Abstract. XML Keyword Search is a user-friendly information discovery technique, which is well-suited to schema-free XML documents. We propose a novel scheme for XML keyword search called XKLUSTER, in which a novel semantic-distance model is proposed to specify the set of nodes contained in a result. Based on this model, we use clustering approaches to generate all meaningful results in XML keyword search. A ranking mechanism is also presented to sort the results.

Keywords: XML, Keyword Search, Clustering.

1 Introduction

Keyword search in database is gaining a lot of attentions [1-11]. Many existing methods about XML keyword search are based on LCA (lowest common ancestor) model. A typical one of them is the SLCA (smallest LCA) method [2, 3, 5], which returns a group of smallest answer subtrees of an XML tree. A smallest answer subtree is defined as: a subtree which includes all the keywords and any subtree of it doesn't contain all the keywords. The root of a smallest answer subtree is called a SLCA. However, SLCA method probably omits some meaningful results. Example 1.1 shown in Fig. 1 illustrates a tree structure of an XML document, in which every node is denoted as its label and the number below it is the Dewey number.

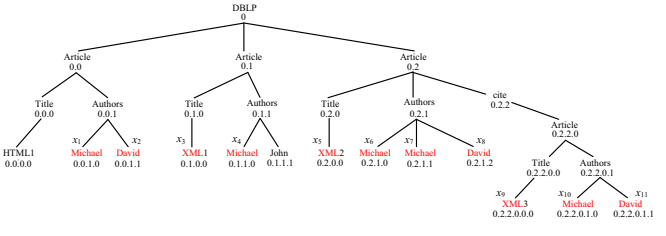

Fig. 1. Tree Structure of an XML document (Example 1.1)

Suppose that keywords are "XML", "Michael" and "David" are submitted to search in the document, and the nodes containing keywords are marked in red. If SLCA method is applied, the result would be a subtree rooted at node

M.J. Zaki et al. (Eds.): PAKDD 2010, Part II, LNAI 6119, pp. 398–409, 2010.

Article (0.2.2.0). The searching semantics is most likely to be "which papers about XML have been written by Michael and David", and obviously it finds the paper "XML3" while omitting the paper "XML2". Moreover, users may have other reasonable searching intentions, like "which papers are written by Michael and David" and "which papers about XML are written by Michael OR David". For these two requests, paper "HTML1" and "XML1" are both contented results but also omitted. If we regard any node set which contains all the keywords as a candidate results, actually what SLCA method does is picking out some "optimal" ones from all the candidate results, and it considers a candidate result to be "optimal" when its LCA is relatively lowest, so those candidate results with higher LCAs are discarded, this is the essential reason for SLCA method losing some meaningful results. Besides, SLCA method still has some other drawbacks: (1) all the results are isolated, which means any node only can exist in one result, and apparently it is inappropriate in some situations; (2) each result is demanded to contain all the keywords, means that SLCA methods only support *"AND"* logic between all keywords.

This paper presents a novel approach based on clustering techniques for XML keyword search, which can solve all above problems. Main contributions include: We propose a novel semantic distance model for XML keyword search in section 2; three clustering algorithms are designed in section 3, which are graph-based (GC), core-driven (CC) and loosened core-driven (LCC) algorithms; a ranking mechanism is proposed to sort all the searching results in section 4.

2 Search Semantics and Results

We define the keywords submitted by users as a set containing t keywords $L = \{k_i | i = 1, \ldots, t\}$,and the XML document as an XML document tree.

Definition 2.1 (Search Semantics and Results). An XML document tree is a tree coded by Dewey numbers, and denoted as:$d = (V, E, X, label(id), pl(id_1, id_2), depth(id), dwcode(id), lca(V'))$, in which: (1)$V$ is the set of all the nodes, in which each node corresponds to an element or an attribute or a value in the document and has a unique identifier as well as a Dewey number; (2) $E \subseteq V \times V$, is the set of edges in the tree; (3) *label(id)*, is a function which can get the label of the node whose identifier is id;(4)$X \subseteq V$, is the set of keyword nodes in the tree, and a keyword node is a node whose label contains any keyword; (5)$pl(id_1, id_2)$, is a function to obtain the path length between two nodes, in which id_1 and id_2 must have an ancestor-descendant relation, and the function returns the number of edges between them; (6) $depth(id)$ is a function to get the depth of the node whose identifier is id (the depth of the root is 1); (7) $dwcode(id)$, is a function to get the Dewey number of the node whose identifier is id; (8) $lca(V')$, is a function to get the LCA of all nodes in V', and $V' \subseteq V$.

Definition 2.2 (Shortest Path). The shortest path between two nodes The shortest path between two nodes x_i and x_j is the sum of the path between x_i and $lca(x_i, x_j)$ and the path between x_j and $lca(x_i, x_j)$. Meanwhile, the

function $spl(x_i, x_j)$ is used to obtain the length of the shortest path. Apparently, $spl(x_i, x_j) = pl(lca(x_i, x_j), x_i) + pl(lca(x_i, x_j), x_j)$.

Definition 2.3 (Semantic Distance). The semantic distance between any two keyword nodes x_i and x_j is defined as a function $dis(x_i, x_j)$: $dis(x_i, x_j) = \frac{spl(x_i, x_j)}{depth(lca(x_i, x_j))}$.

In the rest, semantic distance is called distance for short. In the formula of the distance function, the numerator and the denominator are the length of shortest path and the depth of LCA respectively. Obviously, the semantic distance of two keyword nodes is smaller when their shortest path is shorter or the depth of their LCA is lower. Assume the height (maximum depth) of the tree is h, then the range of $spl(x_i, x_j)$ is $[0, 2h]$ and the range of $\text{depth}(lca(x_i, x_j))$ is $[1, h]$, so the range of $dis(x_i, x_j)$ is $[0, 2h]$. In Example 1.1, evaluating all the distances between any two keyword nodes can get a semantic distance matrix (Fig. 2).

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
x_1	0										
x_2	0.67	0									
x_3	6.00	6.00	0								
x_4	6.00	6.00	2.00	0							
x_5	6.00	6.00	6.00	6.00	0						
x_6	6.00	6.00	6.00	6.00	2.00	0					
x_7	6.00	6.00	6.00	6.00	2.00	0.67	0				
x_8	6.00	6.00	6.00	6.00	2.00	0.67	0.67	0			
x_9	8.00	8.00	8.00	8.00	3.00	3.00	3.00	3.00	0		
x_{10}	8.00	8.00	8.00	8.00	3.00	3.00	3.00	3.00	1.00	0	
x_{11}	8.00	8.00	8.00	8.00	3.00	3.00	3.00	3.00	1.00	0.40	0

Fig. 2. Semantic Distance Matrix (Example 1.1)

From Fig. 2, it can be found that the few smallest distances (0.40 and 0.67) are between two authors of a paper, and any largest distances (8.00) is between titles or authors of different papers which have no citation relation. The values of the distances match the practical meanings. Strictly speaking, the searching intentions of users can never be confirmed accurately; so different than existing researches, we suggest that all keyword nodes are useful more or less and should be included in results. Based on the semantic distance model, we divide the set of keyword nodes X into a group of smaller sets, and each of them is called a "cluster".

Definition 2.4 (Cluster). A cluster C_i is a set of keyword nodes, which means $C_i \subseteq X$; and if $C = \{C_i | i = 1, \ldots, m\}$ is the set of all clusters generated, then $\bigcup_{i=1}^{m} C_i = X$.

We set a distance threshold to confine the size of clusters that the distance of any two keyword nodes in a cluster must be less equal to ω. It actually means, two keyword nodes are regarded semantically related if their semantic distance is less than to ω. Besides, there isn't any restriction for which keywords can exist in one cluster, which means both of the *"AND"* and *"OR"* logic between keywords can be supported. Nevertheless, given a distance threshold, there will be massive clusters satisfying the request, and many of them have inclusion relations with each other, so a concept of *"optimal cluster"* is proposed.

Definition 2.5 (Optimal Cluster). Given a distance threshold ω, a set of keyword nodes $C_i \subseteq X$ is called an *optimal cluster* iff: (1) $\forall x_i, x_j \in C_i, s.t.\ dis(x_i, x_j) \leq \omega$; (2) $\forall x_a, (x_a \in X) and (x_a \notin C_i), \exists x_b \in C_i\ s.t.\ dis(x_a, x_b) > \omega$.

Example 2.1: Assume the value of distance threshold is 2.0, then there are only part of the distances can be retained in the table shown in Fig. 2 (marked in blue), four optimal clusters is easily to be obtained: $C_1 = x_1, x_2, C_2 = x_3, x_4, C_3 = x_5, x_6, x_7, x_8 and C_4 = x_9, x_{10}, x_{11}$, which actually correspond to the contents of four papers respectively in Fig. 1.

Finding the optimal clusters is definitely not the last step. Actually a cluster is only a set of keyword nodes, how to generate enough useful information from clusters is another point. A simple approach is: for each cluster C_i, return the whole subtree rooted at $lca(C_i)$ to users. However many these subtrees will have inclusion relations (for example the subtree rooted at $lca(C_3)$ and the subtree rooted at $lca(C_4)$ in Example 2.1), especially when the depth of some $lca(C_i)$ is high, this kind of problems become serious. Moreover, it would cause some results being too large to be returned and for users to obtain useful information. So, for each *(optimal) cluster*, we generate and return a *"minimum building tree"* of it.

Definition 2.6 (Minimum Building Tree). A minimum building tree (MBT) of any cluster C_i is a tree whose root is $lca(C_i)$ and leaves are all the nodes in $descendants(C_i)$. The function $descendants(C_i)$ returns all the nodes in C_i which doesn't have any descendant node in C_i.

Two extra operations are provided to expand the MBTs: (1) the "expansion" of any node, which would add the subtree rooted at the node into the MBT. For example for the MBT of C_2 in Example 2.1, users can expand the node *Authors(0.1.1)* to get another author of the paper *John (0.1.1.1)*; (2) the "elevation" of the root, which will find the parent node of the MBT root and add it into the MBT, for example for the MBT of C_1 in Example 2.1, users can elevate the root *Authors(0.0.1)* and get its parent node *Article(0.0)*. Thus, users can expand each MBT optionally according to their needs until satisfactory information is found.

Definition 2.7 (XML Keyword Search). XML keyword search is a process that: for a set of keywords L submitted by users and a distance threshold ω set in advance, a set of MBTs are obtained by searching an XML document d; moreover, each MBT could be expanded according to the demands of users.

3 Clustering Algorithms

Since our goal is to divide a set of keyword nodes into groups according to the distances between keyword nodes, if each keyword node is regarded as an object, it is very natural to achieve our goal through clustering techniques, so we develop three clustering algorithms for XML keyword search: GC, CC, and LCC.

Lemma 3.1. For an XML document tree *d*, any number of nodes which have the same depth are picked out, and sorted in preorder to form a list *I*. For any

node x_i in I, traverse leftwards (rightwards) from it, then the distance between x_i and the node each time met is non-decreasing.

Lemma 3.2. The list I is defined as the same as in Lemma 3.1, also a similar list I' is defined. Assume that the depth of the nodes in I' is higher (lower) than that of the nodes in I, x_i is an arbitrary node in I, and x'_i is the ancestor (descendant) node of x_i in I' (x'_i doesn't have to exist in I' or even in the tree, we can add one in the corresponding position of I' if it's inexistent). When traverse leftwards (rightwards) from x'_i in I', the distance between x_i and the node each time met is non-decreasing.

Proof: For any node x_j in I', equals to $spl(x'_i, x_j)$ plus $spl(x_i, x'_i)(spl(x'_i, x_j)$ minus $spl(x_i, x'_i))$, and since $spl(x_i, x'_i)$ is invariable, $spl(x_i, x_i)$ will become greater along with spl(xi', xj) (spl(xi', xj) is non-decreasing). Otherwise, it is obvious that depth(lca(xi, xj)) is non-increasing, according to the formula of semantic distance, is non-decreasing.

After the ω is set and the keywords are submitted, we traverse the XML tree in preorder, find all the keyword nodes and put them into a hierarchical data structure.

Definition 3.1 (Hierarchical Data Structure). Assume the height of the XML document tree is h; the hierarchical data structure H is a data structure which contains h ordered lists of keyword nodes, and the depth of any node in the ith list is i.

When we traverse the XML tree in preorder, each node met would be added into the end of corresponding list in H if its label contains some keyword. Consequently, after the traversal is finished, H will contain all keyword nodes, and because in any list of it all nodes are added in preorder, Lemma 3.1 and Lemma 3.2 hold true in H.

3.1 GC: Graph-Based Clustering Algorithm for XML Keyword Search

In algorithm GC, firstly we traverse H and connect any two nodes between which the distance is less equal to the distance threshold ω with a link. After that a weighted undirected graph whose vertices are all the keyword nodes is obtained. Afterwards, a graph-partition algorithm is used to find all the maximal complete subgraphs (cliques); apparently the vertex set of each clique is an *optimal cluster*. H is accessed from top to bottom, and for each list, nodes are traversed from left to right. Assume x_i is the node currently being accessed, we check the distances between x_i and some neighbors which are right of or below x_i, and because of the accessing order, the nodes left of or above x_i needn't be considered. Assume x_j is a node right of x_i in the same list, according to Lemma 3.1, $dis(x_i, x_j)$ is non-decreasing when the position of x_j moves rightwards, so if $dis(x_i, x_j)$ is greater than ω, all nodes right of x_j in the same list needn't be regarded.

For the nodes in lower lists, firstly it should be confirm that how many lists should be taken into account. For any node x_j below x_i, $dis(x_i,x_j)=\frac{spl(x_i,x_j)}{depth(lca((x_i,x_j)))} \leq \omega$,

also because that $depth(x_i) \geq depth(lca((x_i, x_j)))$, then we can easily get that $\frac{spl(x_i,x_j)}{depth(x_i)} \leq \frac{spl(x_i,x_j)}{depth(lca((x_i,x_j)))} \leq \omega$, which indicates that $spl(x_i, x_j) \leq depth(x_i) \times \omega$, so only lists below x_i need to be taken into account. Obviously, even there exists a descendant of x_i in the $(floor(depth(x_i) \times \omega) + 1)$th lower list, the distance overflows.

For each of these $(floor(depth(x_i) \times \omega)$ lists, first we find the position of the descendant node of x_i in it (the descendant node of x_i doesn't have to exist), then traverse leftwards (rightwards) from the position until the distance overflows, and according to Lemma 3.2, all other nodes in the list needn't be considered.

Each time the distance between two nodes is found to be less equal to ω, these two nodes are linked (by adding cursors point to each other) to build an edge, and the value of distance is recorded as the edge's weight. A weighted undirected graph is obtained, and then a simple graph-partition algorithm is used to get all the cliques. The pseudocode of GC is given in Fig. 3. The graph-partition algorithm is not given here for the sake of space of this paper. We can show that the total cost of finding descendant's position is $h^2 \cdot O(\log n) + h \cdot O(n)$.

It is easy to find that the time complexity of GC not only depends on the total number of keyword nodes, but also strongly relies on the distance threshold. The disadvantage of GC is the uncontrollable efficiency, especially when the distance threshold is large. So, we propose two other algorithms: CC and LCC.

ALGORITHM 1 (GRAPH-BASED CLUSTERING)
Input: a hierarchical structure H
Output : a set of optimal clusters C
1. for (every list l in H) // top-down
2. for (every node x_i in l) // left-right
3. find a rightward neighbor x_j
4. while ($dis(x_i, x_j) <= \omega$)
5. *link*(x_i, x_j); // link two nodes
6. find next rightward neighbor x_j;
7. for (every list l' in *floor*(*depth*(x_i)·ω) layers below l)
8. $p \leftarrow$ *findDescPosition*(x_i, l');
9. traverse leftward and rightward from p until distance overflows and link x_i with neighbors close enough;
10. use graph partition algorithm to get optimal cluster set C;

findDescPosition(x_i, l') // find the position of a descendant of x_i in l'
1. get x_i's descendant x_j in l';
2. if (x_i is the first element of l)
3. use binary search to find p as the *position* of x_j in l';
4. else
5. search rightwards from l'.*cursor* to find p as the position of x_j;
6. l'.*cursor* $\leftarrow p$;
7. return p;

Fig. 3. Algorithm GC

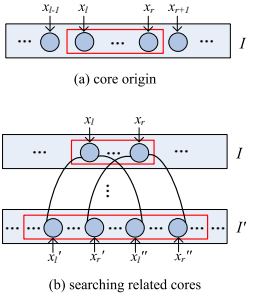

Fig. 4. Illustration of Cores

3.2 CC: Core-Driven Clustering Algorithm for XML Keyword Search

GC is node-driven, which gathers some keyword nodes close enough to each other together. While the idea of algorithm CC is "divide-and-conquer": finds some keyword node sets which are called "cores" in the first place, all the nodes in a

core can definitely be clustered together; afterwards, each of which affirmatively contains at least one optimal cluster; finally *optimal clusters* are obtained from these core sets.

Definition 3.2 (Core). Given a distance threshold ω, a keyword node set $R \subseteq X$ is called a core iff the distance between any two nodes in R is less equals to ω.

An optimal cluster is a core, but a core is not necessarily an optimal cluster.

Lemma 3.3. Assume x_l and x_r are any two nodes in the same list of H, and x_l is left of x_r. Given a distance threshold ω and $dis(x_l, x_r) \leq \omega$, then the set of nodes from x_l to x_r (including them) is a core.

Proof: Take any two nodes x_a and x_b from the $(r - l + 1)$ nodes from x_l to x_r, and let x_a be left of x_b. According to Lemma 3.1, $dis(x_a, x_b) \leq dis(x_a, x_r) \leq dis(x_l, x_r) \leq \omega$, so the distance between any two nodes isn't greater than ω. At the beginning of algorithm CC, we traverse H and divide all keyword nodes in H into a number of cores, each of which is called a "core origin".

Definition 3.3 (Core Origin). O_i is a set of certain keyword nodes in H; it is called a core origin iff: (1) O_i is a core; (2) all the nodes in O_i are in the same list I of H; (3) there doesn't exist a core O_i' which satisfies $O_i \subset O_i'$ and all the nodes in O_i' are in I.

A core origin is actually an *optimal cluster* in one list. As illustrated in Fig. 4 (a): assume I is a list in H, x_{l-1}, x_l, x_r and x_{r+1} are four nodes in I and their positions are just as the same as illustrated in the figure, also $dis(x_l, x_r) \leq \omega$, $dis(x_{l-1}, x_l) \geq \omega$, and $dis(x_r, x_{r+1}) \geq \omega$. According to Lemma 3.3, $x_l, \ldots, x_r$ is a core, and easily find: for any node x_a in it, $dis(x_{l-1}, x_a) > dis(_{xl-1}, x_l) > \omega$, and $dis(x_a, x_{r+1}) > dis(x_r, x_{r+1}) > \omega$. On account of Lemma 3.1, the distance between x_a and any node left of x_{l-1} or right of x_{r+1} is greater than ω, consequently $x_l, \ldots, x_r$ is a core origin. It denotes that the positions of a core origin's nodes are continuous in the list.

After getting all the cores origin from H, some cores around each core origin O_i are considered for the purpose of finding optimal clusters which contain all the nodes in O_i. As illustrated in Fig.4 (b), $x_l, \ldots, x_r$ is a core origin, and according to previous discussions, all other nodes in the same list needn't be regarded. Assume I' is a lower list, $x_l', \ldots, x_l$" is a set of nodes in I' whose distances to x_l are all less equal to ω, and $x_r', \ldots, x_r$" is a set in I' of nodes whose distances to x_r are all less equal to ω. If the intersection of $x_l', \ldots, x_l''$ and $x_r', \ldots, x_r''$ is null, affirmatively there isn't any node in I' could be added into $x_r', \ldots, x_r''$ to form a core; if the intersection isn't null, it must be $x_r', \ldots, x_l''$ (it is easy to prove that x_l' can't be right of x_r' and x_r'' can't be left of x_l''), therefore we can get a lemma as follows.

Lemma 3.4. If $x_r', \ldots, x_l''$ is a core, then: (1) the union of $x_l, \ldots, x_r$ and $x_r', \ldots, x_l''$ is a core; (2) any node in I' other than the nodes of $x_r', \ldots, x_l''$ cannot be added into $x_l, \ldots, x_r$ to build a core.

Proof: For the first thesis, $x_l, \ldots, x_r$ and $x_r', \ldots, x_l''$ are both cores, so the only thing needs to be proved is the distance between any two nodes from different

sets is less equal to ω. Assume x_a is an arbitrary node in $x_l, \ldots, x_r$, x'_a is an arbitrary node in $x'_r, \ldots, xl''$, and $dis(x_a, x'_a) > \omega$, then according to Lemma 3.2: if $dis(xl, xa') \leq \omega$, the position of x'_a's ancestor in I is left of x_a; if $dis(x_r, x'_a) \leq \omega$, the position of x'_a's ancestor in I is right of x_a. However, $dis(x_l, x'_a)$ and $dis(x_r, x'_a)$ are both less equal to ω, the position of x'_a's ancestor node in I conflicts, so $dis(x_a, x'_a) \leq \omega$. For the second thesis, apparently, the distance between x_r and any node left of x'_r and the distance between x_l and any node left of x''_l all exceed ω.

In the same way, we can prove that, when I' is an upper list Lemma 3.4 still holds. Otherwise, when $x'_r, \ldots, x''_l$ is not a core, we can divide it into several cores, and for each core Lemma 3.4 holds. The similar cores as $x'_r, \ldots, x''_l$ are called "related cores" of $x_l, \ldots, x_r$. After all the related cores of $x_l, \ldots, x_r$ being found, we choose one core from each list except I, along with the core origin $x_l, \ldots, x_r$ a core set is obtained, and obviously some optimal clusters containing all the nodes of $x_l, \ldots, x_r$ can be found from it. The pseudocode of algorithm CC is given in Fig. 5, and for simplicity it only considers the case that $x'_r, \ldots, x''_l$ is a core.

Line 1-11 of Algorithm 2 costs $O(n)$; line 6-9 in function *findRelatedCores* costs $O(\log n)$, and line 11 costs $O(n)$, so the total cost of *findRelatedCores* is $O(\log n) + O(n)$. For the function *findOptimalClusters* at line 14 of Algorithm 2, its purpose is to find all optimal clusters which contain all nodes in a cluster origin O. Detailed code is omitted here. It is similar to searching cliques in a graph, but the difference is that we consider cores instead of nodes. The core set S contains a number of cores and each of which comes from a distinct list; assume there are t cores in S, then they have 2^t possible combinations at most; for each of the combination a function similar to *findRelatedCores* needs to be invoked for at most t^2 times; because t is always a small number (less than h), we can see that the complexity of function *findOptimalClusters* is same as *findRelatedCores*.

Also we propose two extreme cases here: (1) when ω is really large, then m is a small number, and the complexity will be $O(n)$; (2) when ω is very small, m tends to n, however line 11 in *findRelatedCores* will cost $O(1)$, so the complexity will be $O(n \cdot \log n)$. Moreover, in the worst case the complexity is $O(n^2)$.

3.3 LCC: Loosened Core-Driven Clustering Algorithm for XML Keyword Search

In most cases users don't have accurate requests for the returned results; we can loosen the restrictions of results for the purpose of improving efficiency. At the beginning of algorithm LCC, H is also traversed to get all the cores origin. Afterwards for each list I in H, two additional lists are built: a head node list *HNL* and a tail node list *TNL*. *HNL* and *TNL* orderly store the first nodes and the last nodes of cores origin in I respectively. Thereafter, for each core origin O, we find some cores origins close enough, and then instead of looking for optimal clusters out of them, we easily add all nodes of them into O to form a result. The pseudocode of LCC is in Fig. 6. Line 1 of Algorithm 3 is to find all the cores

ALGORITHM 2 (CORE-DRIVEN CLUSTERING)
Input: a hierarchical structure H
Output : a set of optimal clusters C

```
1.  for (every list l in H)
2.      O ← Ø;
3.      x_f ← first node of l;
4.      for (every node x_i in l)
5.          if (dis(x_f , x_i) <= ω)
6.              add x_i into O;
7.          else
8.              save Co;
9.              O ← Ø;
10.             x_f ← x_i;
11.     save O;
12. for (each core origin O)
13.     S ← findRelatedCores(O);
14.     C_i ← findOptimalClusters(S);
15.     add optimal clusters in C_i into C;

findRelatedCores(O)
1.  S ← Ø;
2.  for (each layer l except the one O belongs to)
3.      R ← Ø;
4.      x_l ← the most left node in O;
5.      x_r ← the most right node in O;
6.      p_l ← findAnceOrDescPosition(x_l, l);
7.      p_r ← findAnceOrDescPosition(x_r, l);
8.      use binary search to find x_l″;
9.      use binary search to find x_r′;
10.     if (x_r′ is x_l″ or left of x_l″)
11.         add nodes from x_r′ to x_l″ into R ;
12.         add R into S;
```

Fig. 5. Algorithm CC

ALGORITHM 3 (LOOSEN CORE-DRIVEN CLUSTERING)
Input: a hierarchical structure H
Output : a set of clusters C

```
1.  find all cores origin;
2.  for (each core origin O)
3.      S ← findRelatedCoresOrigin(O);
4.      for (each core R in S)
5.          O ← O ∪ R;
6.      add O into C;

findRelatedCoresOrigin(O)
1.  S ← Ø;
2.  for (each layer l except the one O belongs to)
3.      x_l ← the most left node in O;
4.      x_r ← the most right node in O;
5.      p_l ← findADPositionInHNL(x_l, l);
6.      p_r ← findADPositionInTNL(x_r, l);
7.      find the most left x_l″ satisfy to dis(x_l, x_l″) > ω
        in the nodes of HNL right of p_l;
8.      find the most right x_r′ satisfy to dis(x_r, x_r′) > ω
        in the nodes of HNL left of p_r;
9.      if (x_r′ is x_l″ or left of x_l″)
10.         add all cores origin between x_r′ and x_l″ into S;
```

Fig. 6. Algorithm LCC

origin, the processing is the same which in Algorithm 2. For line 4-8 in function *findRelatedCoresOrigin*, the time complexity of each step is $O(\log n)$; and for line 10, the related cores origin only need to be recorded with identifiers rather than be traversed. Assume the number of cores origin is m, then apparently the time complexity of algorithm LCC is $O(n)+O(m\cdot\log n)$; the worst case happens when m tends to n, then it comes to $O(n\cdot logn)$; otherwise, when m is a small number, it is $O(n)$.

4 Ranking of Results

The whole process of ranking mechanism is: firstly sort all the clusters generated by clustering algorithms using the scoring function, and then transform them into MBTs orderly, finally return top-k or all ordered MBTs to users. The first criterion of the scoring function is the number of keywords, obviously containing more keywords indicates a better cluster, and the best clusters are those which contain all the keywords. On the other hand we consider the occurrence frequencies of keywords having no influence on ranking. For those clusters contain the same number of keywords, the criterion of comparison is the average distances of clusters.

Definition 4.1 (Average Distance). The average distance of a cluster C_i is the average value of all the distances between any two nodes in C_i. Assume C_i

contains m nodes, and function $dis_{mean}(C_i)$ is used to get the average distance of C_i, then, $dis_{mean}(C_i) = \frac{\sum_{x_i,x_j \in C_i; x_i \neq x_j} dis(x_i,x_j)}{2C_m^2}, m > 1$

Assume h is the tree height, then the range of $dis_{mean}(C_i)$ is $[1/h, 2h]$. Apparently for a cluster C_i, a smaller $dis_{mean}(C_i)$ indicates that all nodes of C_i are more compact in the tree. Otherwise, if C_i only contains one node, the average distance cannot be defined on it, however we can see that the MBT of it also contains a single node and means almost nothing to users, so this kind of clusters are worst for users. We define function $keywordNum(C_i)$ to get the number of keywords, and $score(C_i)$ as the scoring function of clusters, moreover our final scoring function is as follows:

$$score(C_i) = \begin{cases} h \cdot keywordNum(C_i) + \frac{1}{dis_{mean}(C_i)} & m > 1 \\ 0 & m = 1 \end{cases}$$

Example 4.1: For the four optimal clusters obtained in Example 2.1, their scores are: $score(C_1) = 13.50$, $score(C_2) = 12.50$, $score(C_3) = 18.75$ and $score(C_4) = 19.25$ respectively. So, the order of clusters is: C_4, C_3, C_1, C_2.

5 Experiments

The environment of our experiments is a PC with a 2.8GHZ CPU and 2G RAM; and the software includes: Windows XP, JDK 1.6, and the parser "Xerces". The data sets used to compare three clustering algorithms are DBLP (size 127M, 6332225 nodes) and Treebank (size 82M, 3829511) [12]. We build a vocabulary for each data set, which stores some terms existing in the document; the occurrence frequency of each term is between 5,000 and 15,000. We randomly choose several ones from vocabularies as keywords to search in the documents. The process is repeated for forty times and afterwards the average values are evaluated as the final results.

From Fig. 7 it's easy to find that: (1) given certain keywords and distance threshold, the time costs of three algorithms is ranked as "GC, CC, LCC" (from big to small) except when the distance threshold is very small; (2) with the increasing of the distance threshold, time cost of GC always increases, while time costs of CC and LCC both first increase and then decrease. In Fig. 8 (a), when the distance threshold equals to 0.0, the returned results are those nodes whose labels contain multiple keywords, because each of them is considered as some different nodes; when the distance threshold equals to 6.0, almost all the keyword nodes gather into a few large clusters; also with the threshold increasing, the amount of clusters except single-node ones firstly increases and then reduces to 1. Similar situations happen in other subfigures in Fig 8. We have evaluated the average distances of returned clusters: the results of GC and CC are the same, and LCC's only very litter larger than them (at 4 decimal places). So, the results of GC and CC are the same (all the optimal clusters), and the results of LCC are less and bigger but not quite worse than them. we can see that in any case DBLP has more average keyword nodes, however, Fig. 7 indicates that

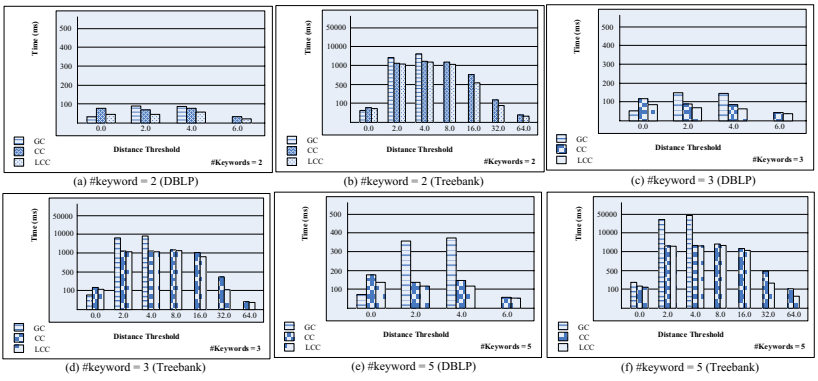

Fig. 7. Efficiencies of Clustering Algorithms

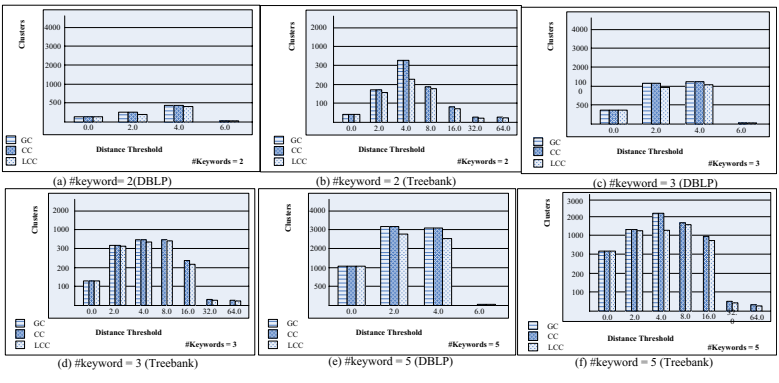

Fig. 8. Result Quantities of Clustering Algorithms

any of the three algorithms costs more time in Treebank than in DBLP with the same conditions, which means the topology of the XML document tree definitely affect the efficiency strongly. The efficiencies become lower when the height is larger and the topology is more complicated.

6 Conclusion

In this paper, to obtain all fragments of the XML document meaningful to users, we propose a novel approach called XKLUSTER, which include a novel semantic distance model, three clustering algorithms (GC, CC, LCC); a ranking mechanism.

Acknowledgments. This research is supported in part by the National Natural Science Foundation of China under grant 60773076, the Key Fundamental Research of Shanghai under Grant 08JC1402500, Xiao-34-1.

References

1. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on XML graphs. In: Proceedings of the 19th International Conference on Data Engineering, pp. 367–378. IEEE Computer Society Press, Bangalore (2003)
2. Xu, Y., Papakonstantinou, Y.: Efficient Keyword Search for Smallest LCAs in XML Databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 537–538. ACM, Baltimore (2005)
3. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, pp. 72–83. Morgan Kaufmann, Toronto (2004)
4. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword Proximity Search in XML Trees. IEEE Transactions on Knowledge and Data Engineering 18(4), 525–539 (2006)
5. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A Semantic Search Engine for XML. In: Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany, pp. 45–46 (2003)
6. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, pp. 16–27 (2003)
7. Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 329–340. ACM, Beijing (2007)
8. Kong, L., Gilleron, R., Lemay, A.: Retrieving meaningful relaxed tightest fragments for XML keyword search. In: 12th International Conference on Extending Database Technology, pp. 815–826. ACM, Saint Petersburg (2009)
9. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML Keyword Search with Relevance Oriented Ranking. In: Proceedings of the 25th International Conference on Data Engineering, pp. 517–528. IEEE, Shanghai (2009)
10. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer.: A System for Keyword-Based Search over Relational Databases. In: Proceedings of the 18th International Conference on Data Engineering, pp. 5–16. IEEE Computer Society, San Jose (2002)
11. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 305–316. ACM, Beijing (2007)
12. XML Data Repository, `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`