Query Biased Snippet Generation in XML Search

Yu Huang
Arizona State University
yu.huang.1@asu.edu

Ziyang Liu
Arizona State University
ziyang.liu@asu.edu

Yi Chen
Arizona State University
yi@asu.edu

ABSTRACT

Snippets are used by almost every text search engine to complement ranking scheme in order to effectively handle user searches, which are inherently ambiguous and whose relevance semantics are difficult to assess. Despite the fact that XML is a standard representation format of web data, research on generating result snippets for XML search remains untouched.

In this paper we present a system, eXtract, which addresses this important yet open problem. We identify that a good XML result snippet should be a self-contained meaningful information unit of a small size that effectively summarizes this query result and differentiates it from others, according to which users can quickly assess the relevance of the query result. We have designed and implemented a novel algorithm to satisfy these requirements and verified its efficiency and effectiveness through experiments.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search Process

General Terms

Algorithms, Design

Keywords

XML, snippets, keyword search

1. INTRODUCTION

The semantics of searches issued by web or scientific users, especially when specified using keywords, are inherently ambiguous. Various ranking schemes have been proposed to assess the relevance of query results so that users can focus on the ones that are deemed to be highly relevant. However, due to the ambiguity of search semantics, it is impossible to design a ranking scheme that always perfectly gauges query result relevance with respect to users' intentions. To compensate the inaccuracy of ranking functions, result snippets are used by almost every text search engine. The principle of result snippets is orthogonal to that of ranking functions: let users quickly judge the relevance of query results by providing a brief quotable passage of each query result, so that users can choose and explore relevant ones among many results.

Despite the fact that XML is a standard representation format of web data, the problem of generating result snippets for XML search remains untouched. Compared with result snippets for text document search, XML presents better opportunities for generating meaningful and helpful search result snippets. In document search, due to the lack of structure, a common strategy is to use document fragments that contain keywords along with the surrounding words as snippets in order to approximate a semantic summary of the document. On the other hand, XML data is semi-structured with mark-ups providing meaningful annotations to data content, and therefore has a better hope to generate semantically meaningful snippets.

In this paper we identify the specific goals that a semantically meaningful result snippet should meet. First, a result snippet should be *self-contained* so that the users can understand it. Second, different result snippets should be *distinguishable* from each other, so that the users can differentiate the results from their snippets with little effort. Third, a snippet should be *representative* to the query result, thus the users can grasp the essence of the result from its snippet. At last, a result snippet should be *small* so that the users can quickly browse several snippets. However, achieving these goals is highly challenging.

To be self-contained, a result snippet should represent a semantic unit. Suppose Figure 1 shows the fragments of a result of query `Texas apparel`. If we choose the fragment between keyword matches in the corresponding XML document as the snippet of this query result, just as what a text search engine does, the users will not be able to see that both matches are nested in the tag `retailer`, and thus not able to easily understand that this query result is about an apparel retailer in Texas (rather than a book discussing the popular apparel styles in Texas).

To illustrate other goals, let us look at a different query `Texas apparel retailer` as the running example. As snippet generation, whose inputs are the user query and a query result, is independent of query result generation, we omit the description of source data and the query result generation techniques in this paper, and only show the fragments

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

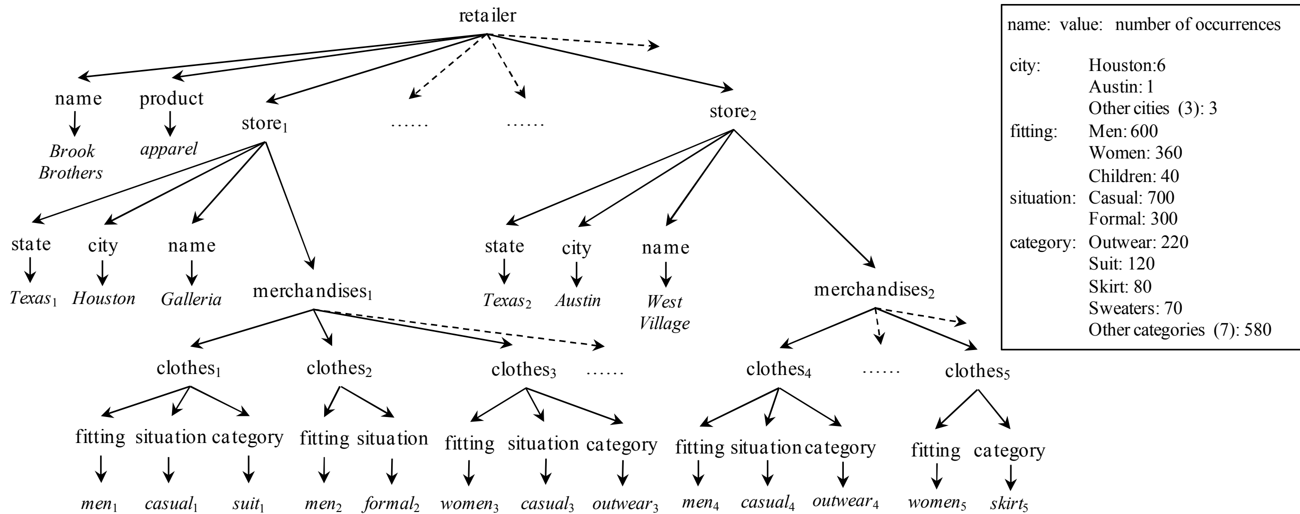


Figure 1: Part of a Query Result of Query `Texas apparel retailer` and Statistics about Value Occurrences.

of a sample query result in Figure 1. Some statistics of the full query result are presented at the right portion in the figure, where for each distinct name-value pair, we record the number of its occurrences in the query result. For instance, "city: Houston: 6" indicates that there are 6 occurrences of Houston as a value of node city in the query result. Values with low occurrences are omitted.

The second goal of snippets is to allow users to easily distinguish different query results from each other. To achieve this in text document search, result snippets often include the document titles. Analogously, we propose to select the unique identifier (aka. key) of a query result into its snippet to identify this query result and highlight the fundamental differences among results. However, it is not clear how to identify the key of a query result. Intuitively, a node in a query result is a key if the values of the nodes with the same name are all distinct in all the query results. A plausible solution would use such nodes as the key of a query result. According to this, the `name` of a `store` could be considered as a key of the results of query `Texas apparel retailer`. Nevertheless, using the `name`s of the `store`s as the key of a query result is unlikely to be reasonable, as the user searches for `retailer`, which can have hundreds of stores.

The third goal is to design snippets that provide a representative summary of the query result by including the most prominent features in the result. Intuitively, a prominent feature is often reflected by a large number of occurrences of such a feature in the result. Continuing our example, suppose `Brook Brothers` has 1000 clothes of different styles, among which 600 are for men and 40 for children. Therefore including clothes of style `men` instead of `children` in the snippet shows a prominent feature of this query result: this retailer targets clothes for men. However, the relationship between the prominence of a feature and the number of occurrences is not always reliable. In our example, the number of occurrences of `Houston`: 6, is much less than that of `children`: 40. However, considering that the majority of `Brook Brothers` stores are in `Houston` in the query result, `Houston` should be considered as a prominent feature. The challenge is how to capture the prominent features from various data values in an XML query result tree.

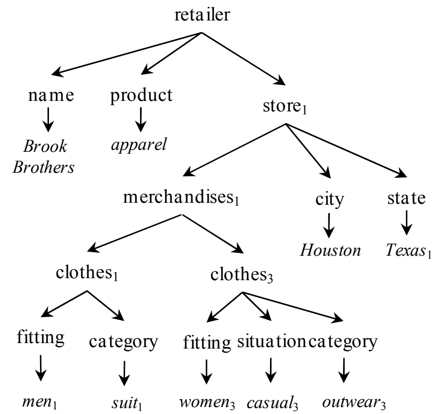


Figure 2: A Snippet of the Query Result in Figure 1

The three goals discussed above address the requirements on the semantics of a result snippet. Clearly the larger a snippet is, then the more information it has, the better it can meet these goals. A trivial solution to generate snippets that meet these goals could be using the query result itself as a snippet. Nevertheless, this is obviously undesirable. The last goal specifies a conflicting requirement: a snippet should be small so that a user can quickly and efficiently browse and understand snippets of several query results. Therefore the challenge is how to provide as much information as possible in a snippet to meet the first three goals within an upper bound of the snippet size.

In this paper we present a system eXtract which addresses the important yet open problem of generating effective snippets for XML search results. We identify that a good XML result snippet should be a *self-contained* information unit of *a bounded size* that effectively *summarizes* the query result and *differentiates* itself from others. To achieve this, we first analyze the semantics of the query result. We discover entities in a query result, whose names can effectively describe the snippet and thus make it self-contained. We then identify the key and dominant features of a query result, based on which a snippet information list is generated, which con-

tains the most significant information in the query result that should be selected into the snippet. Finally, we need to select as many items in the information list as possible given an upper bound of the snippet size. However, we show that this problem is NP-complete. Finally a novel algorithm is proposed that efficiently generates semantic snippets with a given size bound for XML search.

As an illustration, the snippet for the query result in Figure 1 is shown in Figure 2. It captures the heart of the query result in a small tree: the query result is about retailer `Brook Brothers`, which has many stores in `Houston` and features `casual` clothing for `men` and `women`, especially `suits` and `outwears`.

The contributions of our work include:

- This is the first work that studies the problem of generating query result snippets for XML search.
- We identify four goals that a good query result snippet should meet in order to help users quickly get the essence of a query result and assess its relevance.
- To address the goals, we identify the significant information in a query result to be selected into the snippet.
- We prove that the decision problem of whether we can construct a snippet of a given size limit that contains all the significant information is NP-complete.
- We design an efficient algorithm to generate snippets that capture the identified significant information given the snippet size limit.
- A system for generating snippet for XML search has been implemented and tested for its efficiency and effectiveness through experimental studies.

The rest of paper is organized as follows: Section 2 presents how to meet the first three goals in generating meaningful snippets. Section 3 discusses the additional challenge for meeting the fourth goal. Experimental studies are presented in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2. IDENTIFYING SIGNIFICANT INFORMATION IN QUERY RESULTS

We have discussed the four goals in generating result snippets for XML search and the challenges to achieve them. In this section we discuss how to tackle the challenges to meet the first three goals, such that the most significant information in a query result to be selected in its snippet is identified. We start with preliminaries on some notations.

2.1 Preliminaries

We model XML data D as a rooted, labeled, unordered tree. Every internal node in the tree is labeled with a name, and every leaf node is labeled with a data value. In Figure 1, we use subscripts to distinguish nodes with the same label.

A *query result* $R(Q, D)$ of a keyword query Q on XML data D is an XML tree, whose nodes and edges are extracted from D.

A *snippet* $S(R, Q)$ of a query result $R(Q, D)$ is an XML tree, whose nodes and edges are extracted from R.

We define a *snippet information list*, denoted as $IList(R, Q)$, which contains the most significant information in a query result that a snippet should include to meet the first three goals. We initialize this list with the keywords, as shown in the first step in Figure 3, since their matches should be included in the snippet. In the next several subsections we discuss what other items should be added to the list in order to make the snippet self-contained, distinguishable, and representative.

2.2 Self-contained Snippets

Goal 1: A query result snippet should be self-contained so that the user can understand it.

In text search, due to the inherent ambiguity (multi-meaning) of words, providing the *context* of the keyword matches helps users to judge the relevance of query results. Users prefer result snippets that are one or more "windows" on the document containing the complete phrases/sentences where the keyword matches appear because they are self-contained and can be easily read [12].

Analogously, an XML result snippet should also be self-contained. As discussed in Section 1, windows on an XML document where the keyword matches appear are in general not self-contained. XML data may not contain complete phrases/sentences, but use a tree structure (i.e. nested mark-ups) to provide the context information. It is not obvious what is the counterpart in XML data for complete phrase/sentences in text document, which functions as basic semantic units.

To identify basic semantic units, we analyze that an XML database contains information about real-world entities with associated attributes as well as their relationships. An entity represents a basic semantic information unit. We adopt the approach in [10] that leverages the XML data structure to classify XML nodes into three categories: entities, attributes, and connection nodes. Other classification approaches can also be used.

Definition 2.1: A node represents an *entity* if it corresponds to a *-node in DTD. If a node is not a *-node and only has one child which is a data value, then this node, together with its value child, represents an *attribute* [1]. A node is a *connection node* if it represents neither an entity nor an attribute. ■

Definition 2.2: An attribute is associated with an entity E, if E is the nearest ancestor entity of A. ■

For example, in the query result in Figure 1, `retailer`, `store` and `clothes` are considered as entities (assuming each is a *-node in the DTD of the XML data). `fitting`, `situation` and `category`, together with their values, are considered as attributes associated the `clothes` entity. `merchandises` is a connection node.

A self-contained XML result snippet should contain the names of the entities involved in the query result as context information, even though names of such entities may not necessarily appear between two keyword matches in the XML document.

Example 2.1: In our sample query result in Figure 1, entity names `retailer`, `store` and `clothes` should therefore be added to the snippet information list $IList$, as shown in step 2 in Figure 3. ■

[1]In the rest of the paper, attribute refers to the concept in an Entity-Relationship model, rather than the one defined in XML specification.

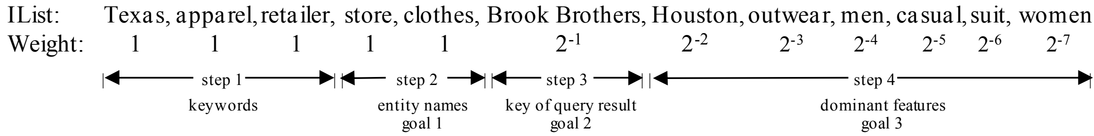


Figure 3: *IList* of the Query Result in Figure 1

2.3 Distinguishable Snippets

Goal 2: A snippet should make the corresponding query result distinguishable from (the snippets of) other query results such that the users can differentiate them with little effort.

As discussed in Section 1, we propose to select the key of a query result into its snippet, reminiscent to the document title in result snippets in text document search, such that a query result can be identified and differentiated from other results.

However, it is not obvious how to identify the key of a query result. If the values of the nodes with the same name are all distinct in all the query results, then such node name can be considered as a key. As we have discussed in Section 2.2, a query result may contain several entities along with their attributes. Therefore a key should be associated with an entity. Then the question is which entities' keys should be considered as the key of the query result. To answer this question, we observe that different entities play different roles in a query result. Intuitively, each query has a search goal. The search goal can be used to classify the entities in a query result into two categories.

1. *return entities* are what the users are looking for when issuing the query.
2. *supporting entities* are used to describe the return entities in a query result.

For example, the goal of query `Texas apparel retailer` is likely to be searching for the `retailer` of `apparel` in `Texas` state. Therefore `retailer` should be considered as the return entity of this query. On the other hand, `store` and `clothes` in the query result are likely to be supporting entities, which are used to describe the return entities.

Since return entities are the core of a query result, their keys can function as the key of the query result and can be used to differentiate this query result from others. Indeed there are often too many differences between two query results. The differences among their return entities best reflect the fundamental differences among query results with respect to the users' search goal. The keys of the return entities identify the return entities, and thus can identify the corresponding query result and highlight the semantic differences among query results concisely. In our example query `Texas apparel retailer`, it is reasonable and concise to use the key of the return entity: `retailer`'s `name` rather than the keys of supporting entities, say `store`'s `name`, or other attributes, to differentiate query results.

Now the remaining question is how to identify these two types of entities in a query result. There could be many heuristics, and we propose the one in Definition 2.3.

Definition 2.3: An entity in a query result is a *return entity* if its name matches a query keyword or its attribute name matches a keyword. If there is no such entity, that is, no keyword matches node names, then we use the highest entity (i.e. entities that do not have ancestor entities) in the query result as the default return entity. ∎

For example, for query `casual outwear`, there is no keyword matching a node name. Assuming a result of this query is a smallest XML tree that contains both keywords, then entity `clothes` is considered as the return entity.

Example 2.2: In our running example, for query `Texas apparel retailer`, entity `retailer` matches a keyword, and therefore is considered as a return entity, corresponding to the user's search goal. The key attribute of `retailer`: `name` is considered the key of this query result, and is added to the snippet information list. The current *IList* comprises the first three steps in Figure 3. ∎

The key of XML nodes can be directly obtained from the schema of XML data, if available, specified as ID or Key node. Otherwise, we find the most selective attribute of the return entity and use it as the key. Specifically, for all query results, we find the attribute of the return entity that has the fewest duplicate values.

2.4 Representative Snippets

Goal 3: A snippet should be representative of the query result, so that the users can grasp the essence of the result from its snippet.

Similar to text document search, a snippet should provide a summary of the query result. Since there can be a lot of information in a query result, a good summary should be concentrated on the most prominent ones referred as dominant features.

We define a feature as a triplet (entity name e, attribute name a, attribute value v). For example (`store`, `city`, `Houston`) is a feature. The pair (entity name e, attribute name a) is referred as the type of a feature, and attribute value v is referred as the value of a feature. Note that the entity name is taken into account because different entities may share the same attribute names. For example, both `retailer` and `store` have attribute `name`. For presentation purpose, we refer a feature by its value when there is no ambiguity.

As discussed in Section 1, a dominant feature of a query result is often reflected by a large number of occurrences of the feature in the result. For example, the fact that there are more clothes for `men` than `children` in the query result indicates that `Brook Brothers` is specialized for `men` instead of `children` clothes.

However, the relationship between the dominance of a feature and the number of occurrences is not reliable due to two reasons. First, different features have different domain sizes. The domain size of a feature type (e, a) is defined as the number of distinct values (e, a, v) of this type, denoted as $D(e, a)$. The smaller size a domain has, the more chances for a value to have more occurrences in the result.

For example, the number of occurrences of `outwear`: 220, is less than that of `women`: 360. However, considering the domain sizes of their corresponding feature types in the query result: $D($`clothes`, `category`$) = 11$, $D($`clothes`, `fitting`$) = 3$, `outwear` could be more dominant than `women` in their respective domains.

Second, due to the tree structure of XML data, different features have different total number of occurrences in the query result, denoted as $N(e, a)$. The more occurrences of a feature type, the more chances that a value of this feature type to occur. For example, a value `Houston` only occurs 6 times, while `children` occurs 40 times in the query result. However, considering the number of occurrences of their corresponding feature types: $N($`store`, `city`$) = 10$, $N($`clothes`, `fitting`$) = 1000$, `Houston` is likely to be more dominant than `children`.

As observed from these examples, comparing the number of occurrences of values of different feature types may not make sense in determining dominant features. To quantify the above intuition, we propose to use normalized frequency, called *dominance score*, to measure the significance of a feature in a query result.

Definition 2.4: We define *dominance score* of a feature $f = (e, a, v)$ as follows:

$$DS(f, r) = \frac{N(e, a, v)}{\frac{N(e,a)}{D(e,a)}} \quad (1)$$

where R is a query result, $N(x)$ denotes the number of occurrences of x in R, $D(e, a)$ denotes the domain size of (e, a) in R. ■

A feature is dominant if its dominance score is larger than 1, in other words, its number of occurrences is more than the average number of occurrences of the feature values of the same type: $N(e, a)/D(e, a)$. There is one exception: if the domain size is 1, $D(e, a) = 1$, then there is only one feature value of this type, which is trivially considered to be dominant even though its dominance score is 1.

Definition 2.5: A feature f is *dominant* in a query result R if one of the following holds: (a) $DS(f, R) > 1$ if $D(e, a) > 1$; or (b) $DS(f, R) = 1$ if $D(e, a) = 1$. ■

We include dominant features into the snippet information list in the decreasing order of their dominance scores.

Example 2.3: Continuing our example, we compute the dominance scores for features in the query results. In the following the corresponding feature types are omitted for conciseness.
$DS($`Houston`$) = 6 / (10 / 5) = 3.0$
$DS($`men`$) = 600 / (1000 / 3) = 1.8$
$DS($`women`$) = 360 /(1000 / 3) = 1.08$
Similarly, we get $DS($`casual`$) = 1.4$, $DS($`outwear`$) = 2.2$, and $DS($`suit`$) = 1.2$.

We add the dominant features into the snippet information list, which now contains the items in all four steps in Figure 3. ■

Since the dominant features of a query result provide a good summary of the result, together with the key of the result they can help user distinguish different query results.

2.5 Algorithm for Snippet Information List Construction

As have been discussed, the snippet information list $IList$ contains the following four components in order. Each item in the list is referred as an *informative item*.

1. Keywords;
2. The names of the entities in the query result, contributing to self-contained snippets;
3. The key of the query result, reflected by the keys of the return entities, contributing to distinguishable snippets;
4. An ordered list of dominant features contributing to representative snippets.

The snippet information list can be generated during a pre-order traversal of the query result, as shown in Algorithm 1. We use e, a and v to denote the last entity name, attribute name and attribute value that have been encountered during the traversal, respectively. First, we add the keywords into $IList$ (line 3). For each node n visited in the traversal, if n is an entity, we set e as $n.name$ (line 6), and add it to $IList$ if it is not already in it (line 7-8). We consider e as a return entity if e matches a keyword (line 9-10). If n is an attribute name, we set a as $n.name$ (line 12), and increase the number of occurrences of feature type (e, a) (line 13). If a matches a keyword, entity e is considered as a return entity (line 14-15). If n is an attribute value, we set v as $n.value$ (line 17), increase the number of occurrences of feature (e, a, v) (line 18) and increase the domain size of feature type (e, a) if feature (e, a, v) has not appeared before (line 19-20). Then we add the key attribute values of the return entities as well as all dominant features into $IList$ (line 21-25). At last, we sort the dominant features in $IList$ according to their dominance scores (line 26).

Now we analyze the time complexity of Algorithm 1. A hash index was built off-line on the XML data, which takes an input of a node ID and returns the information about this node, such as node type (entity, attribute, or connection node) and key values (if exists). Hash indexes are also built to access $N(e, a)$, $D(e, a)$ and $N(e, a, v)$ in $O(1)$. Therefore the cost of traversing the query result (line 4-20) is bounded by the size of the query result, $O(|QR|)$. Since the number of dominant features is bounded by $|QR|$, computing their dominance scores (line 22-23) also takes $O(|QR|)$ time. Sorting $IList$ (Line 26) takes $O(|L|\log|L|)$, where $|L|$ is the size of $IList$. Therefore, the complexity of Algorithm 1 is $O(|QR| + |L|\log|L|)$.

In the following section, we discuss how to extract data nodes from the query result to capture the items in the list as much as possible.

3. GENERATING SMALL AND MEANINGFUL RESULT SNIPPETS

We have discussed how to identify the snippet information list for a query result, which contributes to a self-contained, representative and distinguishable snippet. Besides the requirements on the semantics, we also need to meet a conflicting goal on snippet size.

Goal 4: A query result snippet should be small so that the user can quickly browse several snippets.

Algorithm 1 Construction of Snippet Information List

constructIList (QR, Q)
1: $IList = \emptyset$
2: $returnEntity = \emptyset$
3: $IList = IList \cup Q$
4: **for** each node n in a pre-order traversal of QR **do**
5: **if** n is an entity **then**
6: $e = n.name$
7: **if** $e \notin IList$ **then**
8: $IList = IList \cup \{e\}$
9: **if** e matches a keyword **then**
10: $returnEntity = returnEntity \cup \{e\}$
11: **else if** n is an attribute name **then**
12: $a = n.name$
13: $N(e,a)++$
14: **if** a matches a keyword **then**
15: $returnEntity = returnEntity \cup \{e\}$
16: **else if** n is a value **then**
17: $v = n.value$
18: $N(e,a,v)++$
19: **if** triplet (e,a,v) has not appeared before **then**
20: $D(e,a)++$
21: $IList = IList \cup$ the set of key attribute values of $returnEntity$
22: **for** each feature $f = (e,a,v)$ **do**
23: calculate $DS(f, QR)$ using Eq. (1)
24: **for** each dominant feature f **do**
25: $IList = IList \cup \{f\}$
26: sort all features in $IList$ by dominance score
27: return $IList$

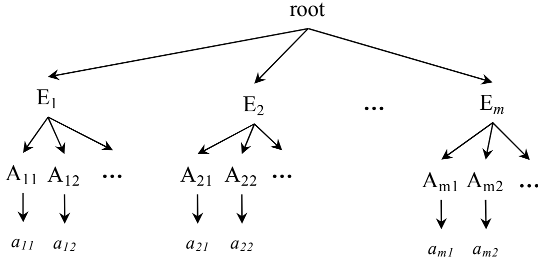


Figure 4: Reduction from Set Cover

3.1 Problem Definition

The challenge is given an upper bound on the size, how to include as many items in the snippet information list as possible into the snippet to make it maximally informative.

Recall that an informative item in the list, such as a keyword and a dominant feature value, can have multiple occurrences in the query result. Although the instances of the same informative item are not distinguishable in terms of semantics, different instances have different impacts on the size of the snippet. To include an instance of an informative item in a tree-structured snippet, we need to add a path to the snippet from its nearest ancestor in the query result that is already in the snippet to this node. Therefore we should carefully select instances such that they are close to each other in order to capture as many informative items as possible given the size limit of the snippet. For example, considering the instance of `Houston`, to capture informative item `outwear`, choosing instance $\texttt{outwear}_3$ results in a smaller snippet tree compared with $\texttt{outwear}_4$.

However, the problem of maximizing the number of informative items selected in a snippet given the snippet size upper bound is hard. We prove that its decision problem is NP-complete as follows.

Definition 3.1: For an XML tree T, let $label(u)$ be the label of node $u \in T$, and $label(T) = \bigcup(label(u) \mid u \in T)$. The tree size $|T|$ is the number of edges in T. We use a boolean $cont(T,v)$ to denote whether tree T contains a label v.

Given an XML tree T, an integer c, and a set P of labels v, $v \in label(T)$, the *instance selection problem* is to find T's subtree T', such that $|T'| \leqslant c$, and $\forall v \in P$, $cont(T,v) = true$. ■

The problem can be illustrated as: given a tree T, a set of labels P and a size bound c, whether it is possible to find a subtree T' of T, such that T' contain every label in P and has a size no more than c.

Theorem 3.1: *Instance selection problem* is NP-Complete.

PROOF. It is easy to see that this problem is in NP. Given a T's subtree T', we can check in polynomial time whether $|T'| \leqslant c$, and $\forall v \in P$, $cont(T,v) = true$.

Now we prove that it is NP-Complete by reducing the set cover problem to it, Set Cover $\leq_P$ Instance Selection. Recall the set cover is the following problem: given a universe $U = \{a_1, a_2, \ldots, a_n\}$, a collection C of m subsets $s_i \subseteq U (1 \leqslant i \leqslant m)$ and an integer k, can we select a collection C' of at most k subset in C, whose union is U, i.e., $\bigcup(s \mid s \in C') = U$?

For any instance of set cover, we construct a tree as shown in Figure 4. For each $s_i \in C$, we construct a node E_i, whose parent is `root`. Let the j-th element in s_i be a_{ij}. For each a_{ij}, we create a node A_{ij} with value a_{ij} as a child of E_i. Except leaf nodes, no node label is the same as an element in U. For every $a_i \in U$, we have a corresponding label a_i and let P denote the set of such labels. Let $c = k + 2|U|$. This transformation takes polynomial time. Next we show that this transformation is a reduction: the set cover problem can be answered if and only if this constructed instance selection problem can be answered.

Given an answer to set cover, we obtain T's subtree T' as follows: $\forall s_i \in C'$, we select E_i and a subset of its children, such that every element $a_i \in U$ has exactly one leaf node in T with value a_i selected. Such leaves along with their ancestors up to the *root* compose T'. Now we have $\forall a \in P$, $cont(T', a) = true$, and $|T'| \leqslant k + 2|U| = c$. T' is an answer to the instance selection problem.

Given an answer to the instance selection problem T', $|T'| \leqslant c$ and $\forall a \in P$, $cont(T', a) = true$. Let R denote the set of nodes E_i in T'. $2|U| + |R| \leqslant |T'| \leqslant c$, therefore $|R| \leqslant c - 2|U| = k$. Let C' be the collection of set s_i, such that $E_i \in R$. The union of s_i is U and $|C'| = |R| \leqslant k$, therefore C' is an answer to the set cover problem. ■

3.2 Algorithm for Instance Selection

As has been shown, the decision problem of instance selection is NP-complete. However, snippet generation must be efficient as web users are often impatient. We propose a greedy algorithm that efficiently selects instances of informative items in generating a meaningful and informative snippet for each query result given an upper bound on size.

There are several challenges in instance selection. First, XML nodes interact with each other. Selecting each individual node in isolation can result in a large snippet. Second, the cost associated with a node, measured by the number of edges to be added to the snippet if this node is selected, changes dynamically during the selection procedure. Third, due to dynamic costs of node selection, we are not able to

determine the number of informative items that can be covered till the very end.

Next we discuss how to address these challenges, by effectively determining the data unit for selection, measuring the benefit and cost of each selection, and designing an efficient instance selection algorithm.

Since informative items in the information list have different priorities, we assign weights to these items. Though we know that the items will be selected in the order of their appearance in the information list until the snippet size limit is reached, we are not able to determine the number of items to be selected beforehand. Note that an item in the list should not be considered before all its precedents are chosen to be in the snippet, otherwise the dominance of different features of the query result can not be faithfully reflected in the snippet. Based on this, we assign the weights to the items in the list to reflect the higher importance of the items that appear earlier in the list. Specifically, the weight of an item is half of the weight of its previous one. For the first several items that are keywords or names of the entities involved in the query result, each is assigned a weight of 1. It is easy to see that such a weighting scheme satisfies the requirement: an item is more important to be selected than all the items after it in the list combined together, as an item must be included in the snippet before any of its successors is included.

Example 3.2: The weight of each item in the snippet information list for the example is annotated below the items in Figure 3. Note that all keywords and entity names have the highest weight of 1. ■

Entity Path Based Selection. For an instance of an informative item to be selected into the snippet, we need to include the path from the closest ancestor of this node that is in the current snippet to the node. We thus make the selections based on paths instead of nodes, which makes the selection procedure more efficient as fewer data units need to be considered for selection. One solution would consider each root-to-leaf path in the query result tree as a data unit for selection in determining which one to be included in the snippet. However, each path often only contains an instance of a single informative item, as most of the informative items are feature values, which are leaf nodes. Therefore each selection is still based on individual informative item covering without considering possible interaction among them.

We consider entity-based paths as data units for selection. We use *leaf entity* to refer to the entities that do not have descendant entities. An *entity path* consists of a path from the closest ancestor of a leaf entity in the query result that is currently included in the snippet, to the leaf entity, along with all the attributes of the entities on the path. If a node instance of an informative item itself or its associated entity is on an entity path, then we say this informative item is *covered* by the path.

Example 3.3: To concisely illustrate our algorithm, here we only present how to choose from the paths in the query result fragment shown in Figure 1, although there are many paths in the query result, based on which the $IList$ in Figure 3 is computed. There are five leaf entities in the figure, all of which are `clothes` entity. Therefore there are five entity paths, each of which is from `retailer` to a `clothes` node. We use p_1 - p_5 to denote the path from `retailer` to `clothes`$_1$ - `clothes`$_5$ respectively. ■

Algorithm 2 Instance Selection of Snippet Information List

$selectInstance$ $(QR, IList, sizeLimit)$

1: $init(QR)$
2: $snippet = \emptyset$
3: $sizeLimitExceeded$ = false
4: $currSize = 0$
5: **repeat**
6: v = the next item in $IList$ to be included in $snippet$
7: **if** v is already covered **then**
8: add the covered instance of v to $snippet$ if it is not in $snippet$ yet
9: $currSize$ += number of edges added to $snippet$
10: **else**
11: find the path p with the highest $(benefit/cost)$ that covers v
12: add the shortest prefix p' of p to $snippet$, such that p' covers v
13: $currSize$ += number of edges added to $snippet$
14: **if** $p' = p$ **then**
15: $p.benefit = 0$
16: **for** each path p'' that share a prefix with p' **do**
17: $p''.cost$ -= length of common prefix of p' and p''
18: **for** each item v' in $IList$ covered by p' **do**
19: put a mark on v' to denote that v' has been covered
20: **for** each path p'' that covers v' **do**
21: $p''.benefit$ -= weight of v' in $IList$
22: **if** $currSize > sizeLimit$ **then**
23: remove all nodes added to $snippet$ in this iteration
24: $sizeLimitExceeded$ = true
25: **until** all items in $IList$ are selected in $snippet$ or $sizeLimitExceeded$ = true
26: return $snippet$

$init$ (QR)

1: $QRroot.ancCover$ = the informative items covered by the root of QR if it is an entity
2: **for** each entity n in the depth-first traversal of QR **do**
3: n' = the nearest ancestor entity of n in QR, and if it does not exist, $QRroot$
4: $n.ancCover = n'.ancCover \cup$ the informative items covered by n
5: **if** n is a leaf entity **then**
6: p = the path from $QRroot$ to n
7: $p.benefit$ = the sum of weights of the items in $n.ancCover$
8: $p.cost$ = number of edges on the path from $QRroot$ to n

Benefit-Cost of an Entity Path. To decide which path to select, we choose the one that has the maximal benefit-cost ratio. The cost of selecting a path p $p.cost$, is the number of edges to be added into the snippet tree when selecting p. Initially, $p.cost$ is the number of nodes on p.

The benefit of selecting path p, $p.benefit$, is the summation of the weights of all the informative items covered by this path. $p.benefit$ is initialized during a depth first traversal of the query result, as presented in procedure *init* in Algorithm 2. For each node n in the query result, we use $n.ancCover$ to denote the set of the informative items covered if n is selected. Note that if n is selected, then all its ancestors in path p will be included in the snippet, therefore $n.ancCover = n'.ancCover \cup V$, where n' is the parent of n, and V is the set of informative items that are covered by n's label and its attributes (if exists). We set $p.cover = n.ancCover$, where n is the leaf entity of p.

Example 3.4: Take the path p_1 from `retailer` to $\texttt{clothes}_1$ for example. p_1 covers informative items `Texas, apparel, retailer, store, clothes, Brook Brothers, Houston, men, casual,` and `suit`, thus $p_1.benefit$ is the summation of their weights, $1 + 1 + 1 + 1 + 1 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-6} \approx 5.86$. Similarly, *benefit* of the paths p_2, p_3, p_4 and p_5 are initialized to be 5.81, 5.91, 5.72, 5.51 respectively. The cost of all paths is 3 initially. ■

Path Selections and Benefit-Cost Updates. The algorithm for selecting informative items is presented in procedure *selectInstance* in Algorithm 2. For a query, each of its query results QR, an upper bound of the snippet size *sizeLimit* and its information list *IList*, we generate a snippet. Initially, the snippet is empty. We process the items in *IList* one by one in order, and select an entity path in the query result that can cover this item with maximal benefit-cost into a snippet, till all the items are covered in the snippet or the upper bound of snippet size is reached.

At each step, let v be the current item being considered. If an instance of v is already included in the snippet, nothing needs to be done. If its associated entity is included in the snippet, it can be easily added into the snippet by adding the path from the associated entity to itself to the snippet (line 6-8). For example, if we want to include an instance of item `outwear` in the snippet, and entity $\texttt{clothes}_3$ is already in the snippet, we simply add the path from $\texttt{clothes}_3$ to $\texttt{outwear}_3$ without choosing another entity path that covers `outwear`, and therefore has a minimal cost.

Otherwise, we need to choose a new entity path to cover the current informative item v. For all entity paths covering v, we choose the one p that has the best cost-benefit $p.benefit/p.cost$ to the snippet (line 11). Notice that for a chosen entity path, we only need to add its shortest prefix p' to cover the current informative item. If p' is a proper prefix of p, then p will not be removed from the entity path lists, but has its cost adjusted, as to be discussed soon; otherwise p can be disregarded.

Example 3.5: The running example is continued here. We start with the first informative item in the list `Texas`. Since all five entity paths cover it, we choose the one with the highest *benefit/cost*, which is p_3. In fact, we only need to add a prefix of p_3, from entity `retailer` to entity $\texttt{store}_1$, together with their associated attributes into the snippet to cover `Texas`. ■

After an entity path p' is added to the snippet, we need to update the information list, the cost and benefit of affected paths, according to the following.

First, for each item in *IList* that is covered by p', we put a mark on it, denoting that it has been covered (line 19). If one of these items is encountered in future, it has a node instance that can be added to the snippet with the low cost (i.e., the number of nodes from this instance to its associated entity, or zero if it is already in the snippet), without the need of choosing another entity path.

Second, we update the costs of the affected entity paths. For each entity path p'' that has a common prefix with p', including p itself, its cost is decreased by the length of the common prefix.

To efficiently calculate the length, we assign each XML node a *Dewey* label [18] as a unique ID. A Dewey ID is composed of integers concatenated by dots. The Dewey ID of a node contains in order the Dewey ID of its parent, a dot, and one more integer denoting the position of this node among its siblings. The Dewey ID of the root is 0. For example, the nodes on the path from `retailer` to $\texttt{Texas}_1$ in Figure 1 should have Dewey IDs 0, 0.2, 0.2.0, 0.2.0.0, respectively.

The length of the common prefix of two entity paths can now be easily calculated as the length of the common prefix of the Dewey IDs of the leaf entities of p' and p'' (line 17). To efficiently identify these affected paths, we sort all entity paths by the Dewey ID of their leaf entities in a list. For the first node n in path p', we find the first and the last path in the entity path list whose leaf entity is a descendant of n, using a binary search. Each of the paths in the entity path list between them has a common prefix with p'.

At last, we need to update the benefits of the affected entity paths. For each entity path p'' that covers an item which is already covered by p', its benefit $p''.benefit$ is decreased by the weight of the corresponding item, for all the commonly covered items of p'' and p' (line 20-21).

After an instance of the current informative item v in *IList* is included into the snippet, we need to check whether the snippet exceeds the size limit. If so, we must remove the nodes that were added into the snippet in this iteration (line 23), and set the flag that no more nodes need to be added into the snippet as its size limit is reached (line 24).

Example 3.6: Continuing the running example, after selecting the highest benefit-cost entity path p_3 to cover item `Texas`, we include its prefix from `retailer` to $\texttt{store}_1$ to the snippet, and perform the following updates. First, we annotate in the *IList* that the informative items `Apparel, retailer, store, Brook Brothers` and `Houston` are covered. Second, we update the costs of affected path. Since paths p_1, p_2 and p_3 have a common prefix with the path included in the snippet, their costs are decreased by the length of this common prefix. Now the costs of the updated entity paths p_1, p_2, and p_3 are all 2.

We also update the benefits of the affected paths. Since `Texas, apparel, retailer, store, Brook Brothers` and `Houston` are all considered to be covered by the first selected path, the benefits of the paths that cover these informative items need to be subtracted accordingly. Specifically, since p_1, p_2 and p_3 originally cover the above six items, each of their benefits is subtracted by the sum of these items' weights: 4.75. p_4 and p_5 cover the first five items in the above list, and each has its benefit subtracted by 4.5.

Now, the next uncovered item in *IList* is `clothes`. We choose the path that covers it and has the highest benefit-cost, hence p_3, which is from $\texttt{merchandises}_1$ to $\texttt{clothes}_3$. Now p_3 is removed from the entity path list, as the entire path is included in the snippet. Now besides `clothes`, item `outwear`, `casual` and `women` in *IList* are also marked as covered. The cost of p_1 and p_2 is now reduced to 1. For each path that covers `clothes`, i.e., p_1, p_2, p_4 and p_5, its benefit is reduced by the weight of `clothes`: 1. For the paths that cover `outwear`: p_4, we subtract its benefit by 2^{-3}. We also subtract the weight of `casual` from the benefits of p_1 and p_4, and subtract the weight of `women` from the benefit of p_5.

We cover the items in *IList* one by one in this manner. Suppose the size limit of the snippet allows us to include all the items in the *IList* in Figure 3 into the snippet, the final snippet is presented in Figure 2. ■

To efficiently select the entity path to cover the current

Film	
QF_1	films, Hitchcock, Paramount
QF_2	films, Hitchcock
QF_3	EasyVirtue, 1927
QF_4	Lifeboat, 1943
QF_5	Dram, films
QF_6	1922, GB, Famous
QF_7	Hitchcock, Paramount
QF_8	30m, films
Retailer	
QR_1	Store, formal
QR_2	Store
QR_3	retailer, Texas, men
QR_4	Store, Texas
QR_5	retailer, California, sportswear
QR_6	Store, Houston
QR_7	Store, Texas, men
QR_8	Retailer, apparel, Store, Philadelphia, formal

Figure 5: Data and Query Sets

item in the snippet information list and to perform updates after a selection, we build a bitmap index for a query result during a traversal. The path dimension has all the paths sorted by the Dewey ID of their leaf entities. The value dimension has all the distinct informative items in the order of their appearance in the query result. Each entry $B(p, v)$ in the index records whether an item v is covered by p, and if so, which node on p covers it. For each path p, we also record its benefit $p.benefit$ and cost $p.cost$.

Now we analyze the complexity of the algorithm. Let QR be the query result, P the set of all entity paths in QR, d the document depth, and $|L|$ the size of $IList$. To include one item v in the information list $IList$ into the snippet, the algorithm searches for the path with the best benefit-cost that covers v (line 11 in $selectInstance$) by traversing all the entries $B(p, v)$, which entails a cost $O(|P|)$. After selecting the entity path p', we update the cost of all the paths that share a prefix with p' (line 16-17). Finding such a path using binary search on the path list has a cost $O(log|P|)$. Each of such paths has the cost updated according to Dewey label prefix computation, which is bounded by d. The complexity of performing cost updates is $O(|P|d)$, as in the worst case all paths need to be updated. We also update the benefits of the affected paths. For each item v' covered by p', we reduce the benefits of the paths that cover v' by traversing all entries $B(p'', v')$ (line 20-21). The complexity of performing benefit updates is $O(|L||P|)$. The total number of iterations is bounded by $|L|$, and the cost of adding nodes into the snippet is $O(|P|d)$. Therefore, the total time complexity for instance selection is $\max\{|L||P|d,\ |L|^2|P|\}$.

4. EXPERIMENTS

We have developed an effective and efficient snippet generation system for XML search: eXtract, which is available online at http://extract.asu.edu.[2] We have evaluated eXtract on three metrics: *quality* of the snippets, *processing time* of snippet generation and *scalability* over the increase of query result size, as well as the upper bound of snippet size in terms of the number of edges in a tree.

The experiments were performed on a 3.6 GHz Pentium 4 machine, running Windows XP, with 2GB memory and 160 GB hard disk.

[2]The web site also shows the snippets generated by Google Desktop as a reference.

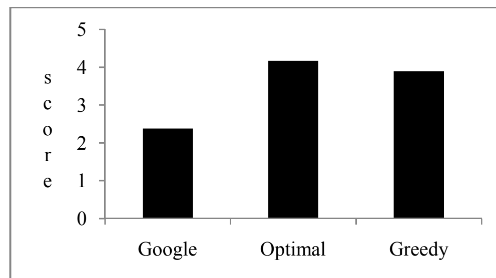


Figure 6: Average Scores of Google Desktop, Optimal Algorithm and Greedy Algorithm over All Queries

We have tested two data sets. Film is an XML data set about the movie and director information.[3] retailer is a synthetic data set that has the same schema and similar domains for node values as the one in Figure 1, while the value of a node is randomly selected from its corresponding domain. For each data set we have tested eight queries, as shown in Figure 5. For each query, a snippet size limit is randomly selected ranging from 6 to 11. The query results are generated using one of the existing keyword search approaches [20].

Since there is no existing snippet generation system designed for XML search in the literature, we compared our system with a popular text document search engine, Google Desktop. To focus the comparison on snippet generation instead of query result generation, we store each keyword query result generated by [20] as an XML file. Then we issue the test query using Google Desktop on the corresponding XML file to obtain its result snippet.

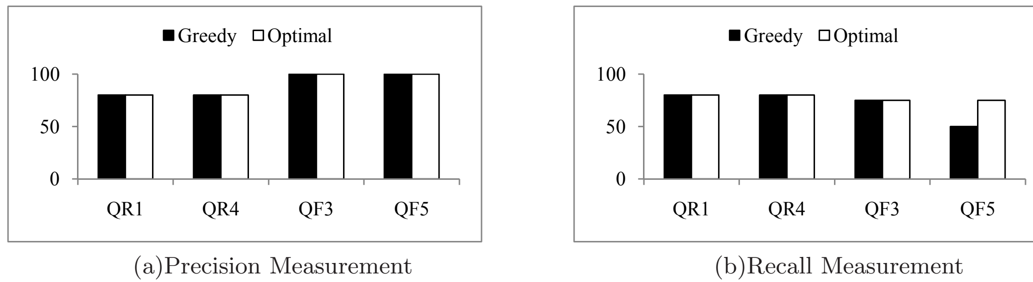
To evaluate our approach, we also implemented an algorithm which generates result snippets that maximally cover the items in the information list within a upper bound of the snippet size, referred as *Optimal* algorithm. The Optimal algorithm enumerates all possible combinations of instances of each item in the list with some pruning of the search space in order to find the optimal solution. In contrast, our approach uses Algorithm 2 for instance selection and is referred as *Greedy* algorithm. Both Optimal algorithm and Greedy algorithm use Algorithm 1 to generate the snippet information list for a query result.

4.1 Snippet Quality

As there is no benchmark for evaluating the snippet quality for XML keyword search, we performed a user study. The quality test involves two parts: scoring snippets generated using three different approaches by users, and measuring the precision and recall of snippets based on the ground truth set by users. Ten graduate students majoring in computer science who were not involved in our project were invited to participate in the user study in assessing the snippet quality.

Assessment of Snippets by Scoring Them. For each query result of the sixteen queries in Figure 5, we use three approaches, Greedy algorithm, Optimal algorithm and Google Desktop, to generate snippets. Since a query result may be large, the users are given the statistical information of each query result (like the one shown in Figure 1) together with

[3]http://infolab.stanford.edu/pub/movies



(a)Precision Measurement (b)Recall Measurement

Figure 7: Precision and Recall Measurements

its snippet. Each user is asked to give a score for each snippet generated by each approach, respectively, on a scale of 5. The snippets are arranged in a random order for each query result without the information about its generation system. The evaluation result is shown in Figure 6. The score for each algorithm shown in the figure is the average score of all the queries provided by all the users. As we can see, the score of our approach is close to that of the Optimal approach, and is much better than that of Google Desktop. For most queries, our approach either generates the same snippet as the Optimal approach, or misses one or two items in the snippet information list compared with the Optimal algorithm, thus their scores are close. Google Desktop is a search engine designed for text documents. It does not generate the snippets as tree structures, but simply concatenates the values in the XML document and outputs a fragment of it. Since Google Desktop has a low score for snippet generation on XML documents, we do not further compare with it in the experiments.

Assessment of Snippets by Comparing with Ground Truth. To make a deep analysis of our approach, we have conducted user surveys to define ground truth for the snippet of a given query result. We found that it is extremely difficult for users, who may not have experience or background in XML, to decide which subtree in the query result should be the desired snippet. Therefore, we asked the users to focus on the content, instead of the tree structure, in a query result. For each query result, each user is asked to provide a set of top k most important items in the query result, which they think should be included in the snippet. Since this part of the user study requires a lot of effort from the users, we randomly choose four queries in Figure 5 to perform this study. After collecting the set of items provided by each user, in order to get the ordered snippet information list, we combine the top-k items from all the users together to form a universal set. Then we rank the items according to the numbers of their occurrences in the universal set, i.e., the number of users who think that the item should be selected in the snippet. The list obtained in this way is considered as the ground truth of the snippet information list. Since the Optimal algorithm guarantees to find the optimal snippet with respect to a given snippet information list, we invoked the instance selection part of the Optimal algorithm on each ground truth information list, whose result is considered as the ground truth snippet for the corresponding query result.

Based on the ground truth of the snippets, we assess the relevance of the snippets generated by Greedy algorithm and Optimal algorithm, both of which invokes Algorithm 1 to generate the same snippet information list, where the number of dominant features is set to be k.

Figure 7 shows the quality assessment of the snippets produced by each approach. Precision measures the percentage of the informative items output by an algorithm that are in the ground truth snippet. Recall measures the percentage of the informative items in the ground truth snippet that are output by an algorithm.

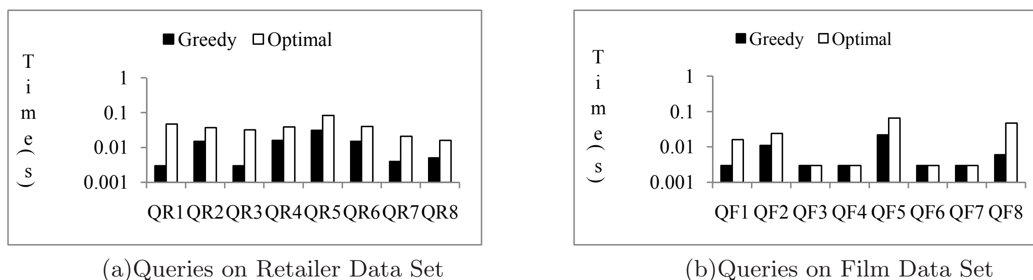
The first observation is that both approaches have a good precision and recall, which confirms the intuition of our approach. This is because the snippet information list that our algorithm generates is similar as the one given by the user study, especially the items that appear earlier in our list. Therefore the snippets generated according to our information list, by Optimal algorithm or Greedy algorithm, has a high quality.

On the other hand, the precisions and recalls of both algorithms are not perfect, mainly because the information list that we generate are often not the same as the ground truth. The main reason of these differences is that the user may care about interesting features of an entity, instead of/besides dominant features. Take QR_1 (`Store, formal`) for example. This query looks for the information of the stores selling formal clothes. To summarize a query result, our approach selects the most dominant features, such as the "fitting:men" of the clothes, which is an attribute of most of the clothes sold by this store. However, users sometimes prefer having attributes of their own interest to be included in the snippet, even though their values may not be a dominant feature. For example, users may choose "brand:Adidas" as an interesting feature to be included in the snippet, which might only appear in a few of the clothes.

Finally, we compare the Optimal algorithm with the Greedy algorithm. For test queries QR_1, QR_4, QF_3, the precisions and recalls of these two algorithms are the same. In fact, the snippets generated by both algorithms for these queries are the same. Recall that the Optimal algorithm maximizes the number of items in the information list to be selected in the snippet. When the snippet size is small, the Greedy algorithm can cover the same set of informative items. Indeed, the upper bounds of the snippet size for the results of the above three queries, are 7, 8 and 6, respectively. On the other hand, for QF_5, which has an upper bound of 11 of the snippet size, the Optimal approach selects more items in the information list into snippets and therefore achieves a better recall. In practice, when the snippet size is small, the Greedy algorithm can achieve similar quality as the Optimal algorithm.

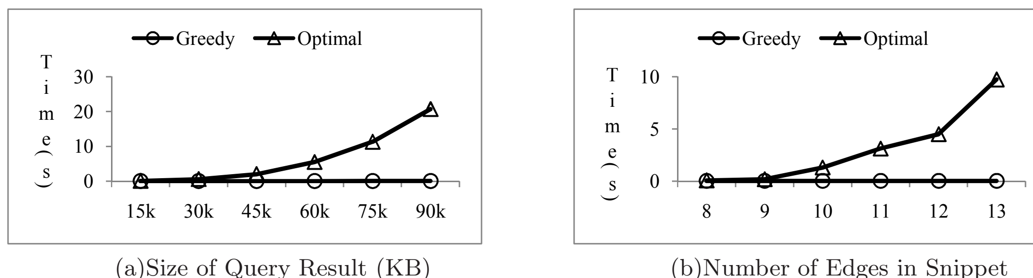
4.2 Processing Time

To evaluate the efficiency of our approach for generating result snippets, we test the processing times of the queries



(a)Queries on Retailer Data Set (b)Queries on Film Data Set

Figure 8: Processing Time on Retailer and Film Data Sets



(a)Size of Query Result (KB) (b)Number of Edges in Snippet

Figure 9: Scalability Test on Size of Query Result and Number of Edges

listed in Figure 5. The processing times, comprising the time of generating *IList* and selecting instances, of the Greedy algorithm and the Optimal algorithm are shown in Figure 8. The sizes of the query results vary from 1KB to 13KB, and the snippet size limits vary from 6 to 11 edges. As we can see, the Greedy algorithm is much faster than the Optimal algorithm.

Both algorithms need to traverse the query result and construct an ordered snippet information list, in the same way. The difference lies in the cost of selecting instances of informative items in the list. The Optimal algorithm searches for the optimal solution by enumerating possible combinations of instances to each item in the snippet information list, which leads to a cost exponential to the size of the information list. When the query result size or the snippet size limit is small (e.g. QF_3 whose query result size is 2KB and snippet size limit is 6, and QF_7 whose result size is 1KB and snippet size limit is 6), the processing times of both algorithms are small. However, when the query result size is relatively large, indicating a potentially large number of instances of each informative item, the difference between the processing times of these two approaches becomes significant (e.g. QR_5 whose result size is 4K and snippet size limit is 8, and QF_8 whose result size is 2K and snippet size is 11).

4.3 Scalability

We test the scalability of our system on the Film data set over two parameters: *query result size* and *snippet size*.

Query Result Size. The scalability test with respect to query result size is shown in Figure 9 (a). A query result of QF_1 is replicated between 1 and 6 times to make the size of query result increasingly larger each time. The upper bound for the snippet size is fixed to be 9. We have tested the performances of the Optimal algorithm and the Greed algorithm on these query results. As we can see, the processing time of the Optimal algorithm grows very rapidly as the complexity of the Optimal algorithm is high-order polynomial to the size of query result, the order of which is the size of the snippet information list. On the other hand, the processing time of the Greedy algorithm grows slowly. For query result of 90KB, it only requires 0.4 second for snippet generation.

Snippet Size. In this test we evaluate the performance of the Greedy algorithm and the Optimal algorithm with respect to the increase of snippet size upper bound, while keeping the query result size to be 4KB. Recall that when the snippet size increases, more items in the snippet information list can be included in the snippet, thus more nodes in the query result need to be processed in order to cover those items. The result in Figure 9 (b) shows that the processing time of the Greedy algorithm increases much slower than that of the Optimal algorithm, where the later has a time complexity exponential to the number of informative items to be output.

In summary, experimental evaluation shows that the snippets generated by our algorithm for XML search has high quality, as reflected by a high score in user evaluations, and high precision and recall with respect to the user defined ground truth. The snippet generation is efficient for various queries, and scales well when the query result size and snippet size increase. Compared with the Optimal algorithm, our algorithm based on a greedy approach has a close quality in practice, and is much more efficient.

5. RELATED WORK

To the best of our knowledge, the research on result snippet generation is within the scope of keyword search on text documents. Early search engines generated query-independent snippets, consisting of the first several bytes of the result document. Such an approach is efficient but often ineffective. Selecting sentences for inclusion in the summary based on the degree to which they match the keywords has become

the state-of-the-art of query-biased result snippet generation for text documents [14, 16, 19, 17]. Commonly used metrics for sentence selection include whether the sentence is a heading or the first line of the document, the number of keywords and distinct keywords that appear in the sentence, etc. Snippet generation considering implicit evidence observed from users' behaviors has also been studied [19]. Approaches for improving efficiency have been investigated [17]. However, snippet generation techniques designed for text documents are unable to leverage the hierarchical structure of XML data, and therefore do not perform well, as observed in the user studies.

There are many works on generating meaningful query results for XML keyword search by inferring the semantics from various perspectives. Some works focus on identifying relevant keyword matches [20, 6, 4, 8, 2, 9, 15], some investigate in how to display query results [7, 10]. As we have discussed, our work addresses the problem of result snippet generation, which takes as input the query results that can be produced by any of the existing XML keyword search engines, and generates meaningful yet small snippets.

There are also many proposals on designing effective ranking schemes on XML search, including [4, 2, 7, 1]. Factors including the distance between keyword matches, term frequency, document frequency, links in the XML documents are explored. Ranking for keyword search on relational databases and graphs have also been investigated [5, 11]. [3] discusses how to order the attributes in a tuple to reflect its influence on the rank of this tuple. Result snippet generation is orthogonal to ranking functions. Due to the ambiguity of keyword search, no ranking scheme can be absolutely perfect and fit all users. The design principles of snippets are therefore independent to those of the ranking scheme, such that the users can make their own relevance judgement based on the snippets without being biased by the relevance assessment made by ranking schemes. Ranking and snippets complement each other to help users find relevant results.

XML data summarization has also been investigated [13], which uses a small amount of space to store the XML data while still achieving accurate results as much as possible for query processing. Techniques are thus developed for data compression, e.g., storing one instance of each distinct tag name with its number of occurrences, recording the frequency that a node name is nested in another node name, and the distribution of values using histograms, wavelets, etc. The goal of data summarization is for query processing efficiency and the data summary is often not user readable, while the goal of snippet generation is to provide meaningful snippets such that users can easily assess the relevance of the corresponding query results.

6. CONCLUSIONS

To the best of our knowledge, this is the first work that addresses the problem of generating result snippets for XML search. We identify four features of a good XML result snippet: *self-contained*, *distinguishable*, *representative* and *small*. To meet the first three requirements in generating semantically meaningful snippets, we identify the most significant information in the query result that should be selected into a snippet in a snippet information list. To satisfy the fourth requirements, we need to generate the snippet that is maximally informative with respect to this list given an upper bound of the snippet size. However, its decision problem is proven to be NP-complete. Finally, we have designed and implemented a novel algorithm to efficiently generate informative yet small snippets. We verified the effectiveness and efficiency of our approach through experiments.

7. ACKNOWLEDGMENTS

This research was supported in part by NSF grant IIS-0740129 and IIS-0612273.

8. REPEATABILITY ASSESSMENT RESULT

All results except user study (Figure 6 and 7) in this paper have been verified by the SIGMOD repeatability committee.

9. REFERENCES

[1] M. Barg and R. K. Wong. Structural Proximity Searching for Large Collections of Semi-structured data. In *Proceedings of CIKM*, pages 175–182, 2001.

[2] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

[3] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan. Ordering the Attributes of Query Results. In *SIGMOD*, 2006.

[4] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[5] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *SIGMOD*, 2007.

[6] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4), 2006.

[7] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *ICDE*, 2003.

[8] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *CIKM*, 2007.

[9] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[10] Z. Liu and Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. In *SIGMOD*, 2007.

[11] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: Top-k Keyword Query in Relational Databases. In *SIGMOD*, 2007.

[12] C. D. Manning, P. Raghavan, and H. SchÍźtze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[13] N. Polyzotis and M. Garofalakis. XCluster Synopses for Structured XML Content. In *ICDE*, 2006.

[14] H. G. Silber and K. F. McCoy. Efficiently Computed Lexical Chains As an Intermediate Representation for Automatic Text Summarization. In *Comput. Linguist.*, volume 28(4), 2002.

[15] C. Sun, C.-Y. Chan, and A. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, 2007.

[16] A. Tombros and M. Sanderson. Advantages of Query Biased Summaries in Information Retrieval. In *SIGIR*, 1998.

[17] A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast Generation of Result Snippets in Web Search. In *SIGIR*, 2007.

[18] V. Vesper. Let's Do Dewey. http://www.mtsu.edu/ vvesper/dewey.html.

[19] R. W. White, I. Ruthven, and J. M. Jose. Finding Relevant Documents using Top Ranking Sentences : An Evaluation of Two Alternative Schemes. In *SIGIR*, 2002.

[20] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, 2005.