Fast Structural Query with Application to Chinese Treebank Sentence Retrieval

Chia-Hsin Huang[1]
jashing@iis.sinica.edu.tw

Tyng-Ruey Chuang[2]
trc@iis.sinica.edu.tw

Hahn-Ming Lee[3]
hmlee@mail.ntust.edu.tw

Institute of Information Science[1,2]
Academia Sinica
Taipei 115, Taiwan

Department of Electronic Engineering[1],
Department of Computer Science and Information Engineering[3]
National Taiwan University of Science and Technology
Taipei 106, Taiwan

ABSTRACT

In natural language processing, a huge amount of structured data is constantly used for the extraction and presentation of grammatical structures in sentences. For example, the Chinese Treebank corpus developed at the Institute of Information Science, Academia Sinica, Taiwan, is a semantically annotated corpus that has been used to help parse and study Chinese sentences. In this setting, users usually use structured tree patterns, instead of keywords, to query the corpus.

In this paper, we present an online prototype system that provides exploratory search ability. The system implements two flexible and efficient structural query methods and employs a user-friendly web-based interface. Although the system adopts the XML format to present the corpora and search results, it does not use conventional XML query languages. As searching the Chinese Treebank corpora is structural in nature and often deals with structural similarities, conventional XML query languages, such as XPath and XQuery, are inflexible and inefficient. We propose and implement a query algorithm, called Parent-Child Relationship Filter (*PCRF*), which provides flexible and efficient structural search. *PCRF* is sufficiently flexible to provide several similarity-matching options, such as wildcard, unordered sibling, sub-trees, ancestor-descendant matching, and their combinations. In addition, *PCRF* supports stream-based matching to help users query their XML documents online. We also present three accelerating rules that achieve a 1.5- to 8-fold performance improvement in query time. Our experiment results show that our method archive a 10- to 1000-fold performance improvement compared to the usual text-based XPath query method.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems – *Pattern matching*; H.3.3 [**Information Systems**]: Information Storage and Retrieval – *Search process.*

DocEng'04, October 28–30, 2004, Milwaukee, Wisconsin, USA.

General Terms

Algorithms, Performance.

Keywords

Corpus, Structural Query, XML.

1. INTRODUCTION

In natural language processing (NLP), many corpora (which are comprised of structured data) are constantly used for the extraction and presentation of grammatical structures in sentences. As more and more corpora become available, it is important to develop a system with comprehensive structural representation and exploratory search ability.

The CKIP Chinese Treebank [11] corpus developed at the Institute of Information Science, Academia Sinica, Taiwan, and the Penn Chinese Treebank corpus developed at the Linguistic Data Consortium [22], are semantically annotated corpora that have been used to help parse and study Chinese sentences. Many research methods are based on these corpora, such as those on building example-based parsers [15], developing and enlarging Treebank [16], comparing and integrating grammar [7], machine translation processing [17], and grammar extraction [6]. In [15], example-based parsers are proposed using Treebank entries as training examples. In [16], Treebank corpora are used as seeds to develop or enlarge corpora. In [6][7], several types of matching are proposed to compare sentence trees. Meanwhile, [17], focuses on tree retrieval based on similarity, rather than on exact match. Despite the above research, we are facing the following system issues in NLP research:

Problem 1 – Lack of comprehensive representation schemes:
Most Treebank corpora do not use a general and comprehensive representation schemes. The sentences are often tagged with some syntactic annotations [22], such as part-of-speech (POS) annotations. A dedicated preprocessor, therefore, is needed to convert the POS annotation before the corpora can be used. Unfortunately, the POS annotation could be too complicated to use, especially when a user does not have previous experience. It creates a barrier to developing new NLP tools, or integrating corpora into other tools.

Problem 2 – Complicated Grammar (data schema):
A data schema is useful for formulating queries, browsing data structures, and enabling query optimizations [18]. Unfortunately, the grammar for sentences in the Chinese Treebank and other corpora is usually too complicated to produce data schema. Without the help of overall structural summaries of the corpora [12], corpus-based NLP applications may run less efficiently.

Problem 3 – High computational complexity in NLP:
The complexity of matching a sentence structure to a large chunk of a corpus is usually high. In order to make example-based parser [15] and corpus-based parsing [16] techniques truly useful, the matching process must be sufficiently efficient to be used for on-line time critical applications.

Problem 4 – The need for flexible approximate tree-pattern matching algorithms:
In NLP, an approximate tree-pattern matching algorithm (rather than an exact tree-pattern matching algorithm) is frequently employed to answer similarity queries [7][17]. Although developing an all-purpose and efficient tree-pattern matching algorithm is difficult, flexible matching algorithms that can provide various types of matching options are always in great demand.

Problem 5 – The need for good user interfaces:
As shown in [10], a user-friendly interface is essential in NLP to help users formulate complicated queries efficiently. Moreover, it would be more useful if the interface could provide functions like result-set operations that can be used to combine the results from previous queries.

Before we describe how to deal with the above problems, we briefly introduce XML processing technologies, which have become the standard for data exchange over the Web. XML documents are essentially self-describing tree-like data. Several query languages, such as XPath [24] and XQuery [23], have been developed to query XML documents. However, these languages are designed primarily for exact match and processing of (parts of) XML documents, and are not really suitable for filtering a collection of documents by their intrinsic structures (either exactly or approximately). As languages like XPath often assume a tree-traversal model of processing, XPath expressions can be quite inefficient if indexing or structural summary techniques are not employed.

Methods like DataGuides [18], T-Indexes [21], ToXin [8], Accelerating XPath [20], Light-weight DFA [12], XFilter [13], YFilter [25] and XTrie [2] have been proposed to improve the performance of path-based query processing. In brief, DataGuides provides structural summaries, and T-Index indexes the paths in semi-structured data to accelerate query processing. ToXin is an indexing scheme for XML data that fully exploits the overall path structure of the database in all query processing stages, while Accelerating XPath uses an R-tree instead of a B-tree data structure to index XML data. It shows that R-tree performs better than B-tree in edge-based indexing schemes. Methods like light-weight DFA, XFilter, YFilter and XTrie are designed to filter XML documents with XPath expressions. Both XFilter and YFilter treat a tree pattern as a set of finite state automata (FSA), each of which is used to match some paths in the tree pattern. Finally, light-weight DFA improves the above mentioned automata-based filtering methods by reducing the memory requirement and supporting the process of branching XPath expressions. XTrie decomposes tree patterns into a collection of substrings and indexes them by using a Trie data structure. Like path-based query, some of the above approaches, such as Light-weight DFA and YFilter, can improve the performance of structural query processing, but they hardly deal with the approximate tree-pattern matching problem.

Wang et al implemented a system called Approximate-Tree-By-Example (ATBE) that allows inexact matching of trees [10]. It provides a flexible query language, has a user-friendly interface, and uses editing distance to evaluate the difference between the pattern tree and the subject tree. The method is related to the unordered tree-pattern matching problem, in which the ancestor-descendant relationship is emphasized, but the order of the siblings is unimportant. In addition, Wang et al developed another approximate unordered labeled tree-pattern matching algorithm, called ATreeGrep [5]. By giving a threshold value *DIFF*, a query pattern *Q*, and a database *D* of trees, the algorithm can find all trees *T* in *D*, where *T* approximately contains *Q* within distance *DIFF*. The algorithm first builds a suffix array database *SD* for all the trees in *D*, and then compares the root-to-leaf paths of *Q* with the paths in *SD* to locate those substructures approximately matching *Q*. The TIGER Project [19] uses a similar approach. It presents the TIGER Treebank, several representation formats, a search tool, and a graphic user interface. Although the project presents a language to query the Treebank, the query method does not support approximate tree-pattern matching.

An Overview of Our Approach
Here we give an overview of our approach, and describe how we deal with the NLP-related problems mentioned above. We have developed a prototype system that provides exploratory search ability, implements two flexible and efficient structural query methods, and employs a user-friendly interface. Our contributions in this paper include the following:

1. We propose a novel approximate tree-pattern matching algorithm. The concept of our algorithm is to filter out ineligible trees as soon as possible, and not spend time traversing them. Two query methods based on this concept are implemented. The *PCRF* method, which needs to index the XML documents (*i.e.* the dataset) first, but is very efficient; and s*tream-based PCRF*, which is able to perform stream-based matching to help users query XML documents online.

2. Our matching algorithm has similarity-matching ability. It is sufficiently flexible to provide several similarity-matching options, such as wildcard, unordered sibling, sub-trees, ancestor-descendant matching, and their combinations. By giving a weight for each matching option, users can rank the search results by a user-defined similarity measure.

3. We propose three accelerating rules for our algorithm, which can dramatically reduced the matching time of some special query patterns. Our experiment results show that the performance can be improved by a factor of 1.5 to 8.3.

4. In addition, we implement a friendly web-based tool, named *TreeBuilder*, to help users design complicated queries efficiently. It provides two result-set operations, union and intersection, for users to combine the matched results from previous queries.

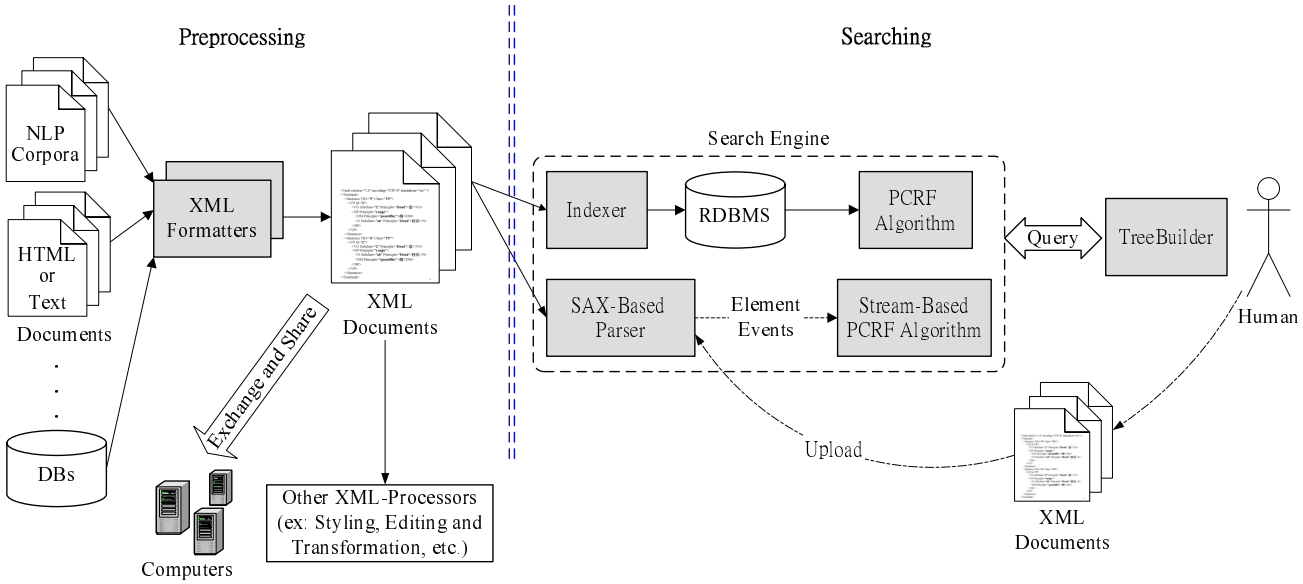


Figure 1. The system architecture.

5. Finally, our prototype system adopts the XML format to present the corpora and search results. As more and more third-party XML tools become available on the Internet, users will be able to develop new and interesting applications easily.

Note that, as mentioned in Problem 2, the schema (grammar) of Treebank corpora is too complicated to summarize their document structures [12]. We used the Parent-Child Relationship indexing scheme (edge indexing) in our methods because it is more suitable for filter processing. It is also very simple (it only keeps edge information) and efficient.

2. SYSTEM OVERVIEW

Figure 1 shows the architecture of our prototype system. First, the *XML Formatters* in the preprocessing part transform various types of documents, such as annotated NLP corpora (Figure 2), HTML documents, pure text documents, and structured data from databases, into XML formats (Figure 3). As numerous third-party XML tools for styling, editing, and transforming XML documents are now available on the Internet, users can develop various applications themselves. In the searching part, the *Indexer* component (a preprocessing module of the *PCRF* algorithm) decomposes the XML documents into a set of parent-child pair (edge) records, and stores them in a relational database management system (RDBMS). By means of the efficient indexing technique employed by the RDBMS, our *PCRF* method can fetch and filter edge records quickly. Moreover, if users want to query XML documents online, the *stream-based PCRF* method (which uses the SAX [3] parser to stream the XML documents) can also be used. Finally, a web-based tool, *TreeBuilder*, helps users design complicated queries efficiently. By using this tool, the query we used in our research can be generated easily (see Figure 9). It also provides two result-set operations, union and intersection, for users to compose the matched results. A snapshot of *TreeBuilder* is shown in Figure 4. The prototype is implemented as a Java applet so that it can be executed on any platforms that support Java.

```
VP(
 agent:NP(
  property:NP(
   quantifier:DM:多家 [many] |
   Head:Ncb:旅行社 [travel agencies] )|
  property:A:高階 [high-level] |
 Head:Nab:主管[managers] )|
Head:VE2:表示 [express] )
```

Figure 2. An example from the CKIP Chinese Treebank.

```
<Sentence TID="19308" Class="VP">
 <VP>
  <NP Role="agent">
   <NP Role="property">
    <DM Role="quantifier" Word="多家" />
    <N Subclass="cb" Role="Head" Word="旅行社" />
   </NP>
   <A Role="property" Word="高階" />
   <N Subclass="ab" Role="Head" Word="主管" />
  </NP>
  <VE Subclass="2" Role="Head" Word="表示" />
 </VP>
</Sentence>
```

Figure 3. The XML format that corresponds to the example in Figure 2.

3. QUERY ALGORITHM

In this section, we describe the Parent-Child Relationship Filter (*PCRF*) algorithm, four advanced query options, three acceleration rules, and present an analysis of the time complexity of the *PCRF* algorithm. Hereafter, we use *P*, *T* and *S* to denote, respectively, the *pattern tree* (the query), all the *subject trees* (the Treebank corpus), and a sub-tree from *T* (a sentence of Treebank corpus). We use $|P|$ and *height(P)* to refer to the number of nodes in *P* and the height of *P* (where the root has height 0).

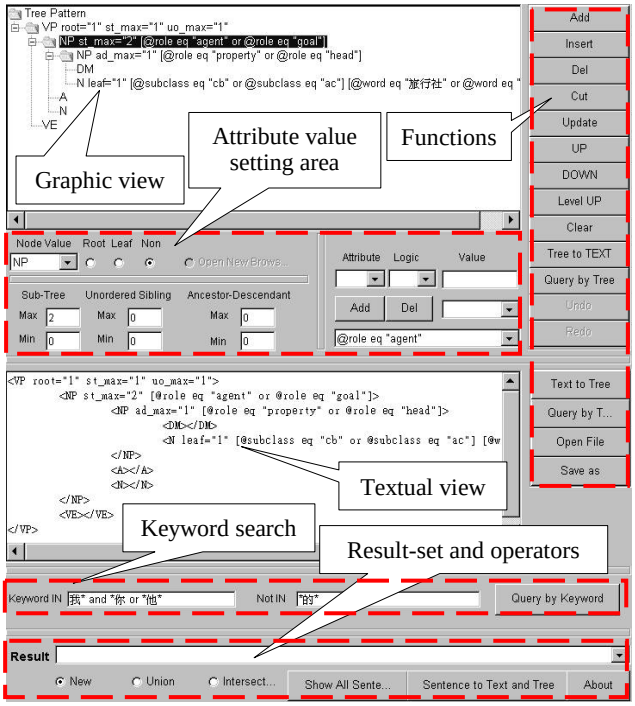


Figure 4. A snapshot of TreeBuilder

3.1 Methodology

The steps of backtracking and tree traversal are the bottlenecks of the tree-pattern matching problem. They are particularly evident when one wants to search for a tree pattern against a very large collection of trees, where only some trees will be acceptable. In this case, we want to filter out ineligible trees as soon as possible, and not spend time traversing them. Our query algorithm reduces the tree-pattern matching problem to a searching-and-combining problem. In fact, the computational complexity of our algorithm is $O(|T| \times d\log_d|T|)$ in the worst case, where d denotes the minimum degree of the B-tree structure (see Section 3.7), but the experiment results show that our *PCRF* method runs faster than the linear time XPath method [9].

3.2 Parent-Child Relationships Filter (PCRF) Algorithm

PCRF consists of five main steps: indexing, searching, fetching, checking sub-tree structure, and validating connection relationship. We describe them below:

1. Indexing

This is a preprocessing step. We store and index all Parent-Child Relationships (*PCR*) of *T* in a data structure that can be retrieved efficiently. A *PCR* is defined as a parent-child pair (edge). In our application, a *PCR* record consists of the tree identifier (*TID*), the parent node identifier (*PID*), the child node identifier (*CID*), the type of the parent node (*PNODE*), and the type of the child node (*CNODE*), where the node identifier is equal to the preorder numbering of a node in the tree. In our current implementation, we use a RDBMS to store and index *T's PCR* records (the attributes and text data are stored in another table). The primary key of the *PCR* table is {*TID*, *PNODE*, *CNODE*}. The overall time cost of storing and indexing those *PCR* records is reported in the experiment results (Section 4).

2. Searching

In the searching step, we convert *PCRs* of *P* into a structured query language (SQL) expression. Suppose that the sentence fragment in Figure 3 is a pattern *P*. Figure 5 shows the corresponding SQL expression, where *TAB* denotes the table name that contains the entire *T*'s *PCR* records. Each *select* command searches a *PCR* of *P*. Consequently, if *S* contains all *PCRs* of *P*, it will be reserved as a candidate; otherwise, it will be filtered out.

```
select distinct TID from TAB where PNODE='NP' and
CNODE='N' group by TID having count(*) >= 2
and TID in (
select TID from TAB where PNODE='NP' and CNODE='NP'
and TID in (
select TID from TAB where PNODE='NP' and CNODE='DM'
and TID in (
select TID from TAB where PNODE='NP' and CNODE='A'
and TID in (
select TID from TAB where PNODE='VP' and CNODE='VE'
and TID in (
select TID from TAB where PNODE='VP' and CNODE='NP')))))
```

Figure 5. The SQL expression for locating nodes that may match the pattern in Figure 3.

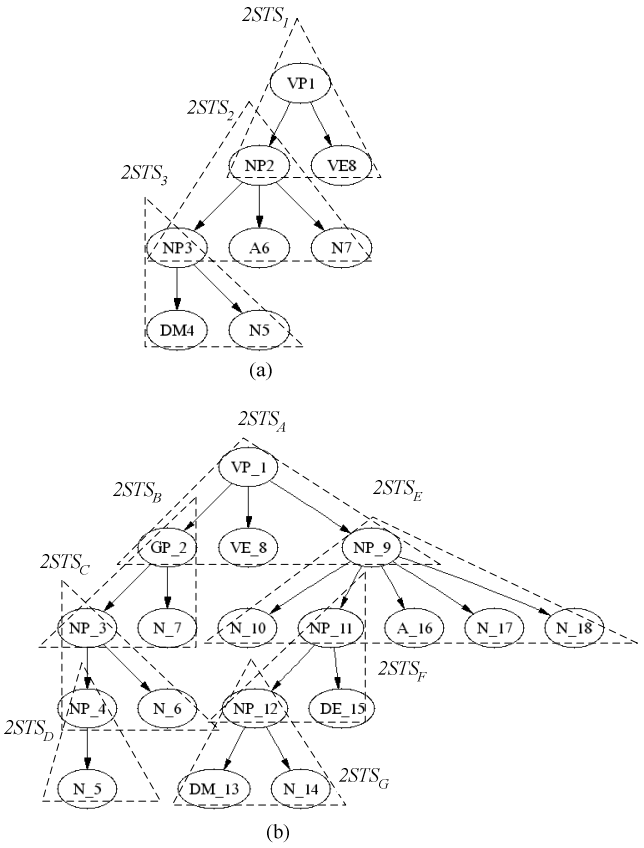
3. Fetching

After the searching operation, we have a subset $T' \subseteq T$ as a candidate. In practice, it is often $|T'| << |T|$. We then use those *TIDs* in *T'* to fetch all *PCR* records from the RDBMS and store them in a hash table. This fetching operation is very expensive in the *PCRF* algorithm, as it consumes more than 80% of the time of the matching process. Because $|T'|$ could still be very large, we propose three accelerating rules (see Section 3.4) to avoid fetching unnecessary *PCR* records into the hash table. The experiment results (Section 4) show a 1.5- to 8-fold performance improvement by applying these accelerating rules to our *PCRF* algorithm.

We now describe in detail the data structure used to store *PCR* records. We define a two-level sub-tree structure (*2STS*) as: a set of nodes that contain a parent node and all of its child nodes. That is, a *2STS* is a set of *PCR* records that have identical parent nodes (*i.e.*, rooted at the same *PID*). Figures 6 (a) and (b) show three and seven *2STSs* of *P* and *S'*, respectively, (*S'* being a sentence tree in *T'*). A node in *P* is denoted by concatenating its node type and identifier, for example, *VP1*. For a node in *S'*, its node type and identifier are separated by an underline symbol, for example, *VP_1*.

Naturally, we use a two-level hash table to store the *2STSs*. The abstract view of the hash table is shown in Figure 7. The *PNODE* (*i.e.*, the type of a parent node) of a *2STS* is stored in the first-level, while the entire *2STS* is stored in the second-level. The *2STSs* in the second-level can be distinguished by their *PIDs* (*i.e.*, the node identifier of a parent node). Note that, some *2STSs* of *S'* are useless during matching processing, so we filter them out. First, if the *PNODE* of a *2STS* does not exist in the first-level of

the hash table of *P*, it is filtered out. $2STS_B$ is an example. If the *PNODE* exists, but some of its child nodes' *CNODEs* (*i.e.*, the type of a child node) do not exist in the second-level hash tables of *P*, they are also filtered out. The *GP* node of $2STS_A$ is such a case. The next stage is to check the size of each *2STSs* in *S'*. If it is smaller than the minimal size of those *2STSs* in the corresponding parent node of *P*, it too is filtered out. $2STS_D$ and $2STS_F$ are two examples. Finally, if *S'* has an empty entry in the first-level of the hash table, it is filtered out as well.



Figures 6. (a) The two-level sub-tree structures of the query tree. (b) The two-level sub-tree structures of a sentence tree of the subject tree.

4. Checking Sub-tree Structure

A candidate tree *S'* in *T'* contains all *PCRs* of *P*, but it might have a wrong two-level sub-tree structure (*2STS*). In this step, we filter out those *S'* whose *2STSs* are not a superset of those in *P*. For each *2STS s* in *S'*, we check if it covers some *2STS p* of *P*. That is, whether *s* and *p* have the same *PNODE*, and whether the child nodes of *s* are a superset of the child nodes of *p*. If so, the *2STS s* will be kept. If *S'* has fewer *2STSs* than those in *P*, it is filtered out. In our example, $2STS_1$, $2STS_2$, and $2STS_3$ match $2STS_A$, $2STS_E$, and $2STS_G$, respectively. This matching information is stored in run-time memory for further use. Finally, if there is a *p* that can not match any *s*, then *S'* can not be matched by *Q*, so *S'* is also filtered out.

This checking step can be performed efficiently because all the *2STSs* have been grouped. That is, only *2STSs* of the same *PNODE* will be compared to each other (one in *P* and the other in *S'*). After checking all sub-tree structures, we construct $T'' \subseteq T'$, a collection of candidate trees with each having all the *2STSs* in *P*. In practice, it is often $|T''| << |T'| << |T|$.

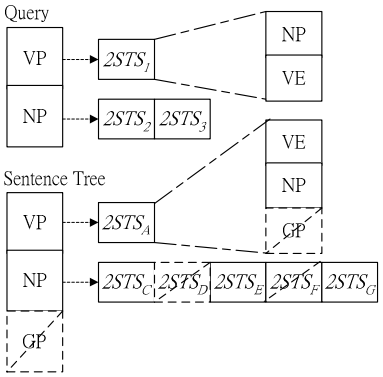
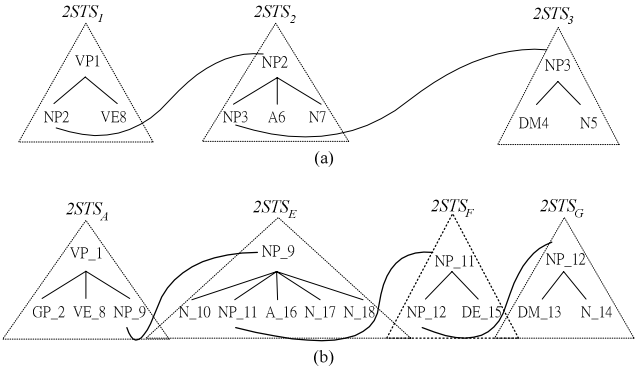


Figure 7. The abstract view of the two-level hash table.



Figures 8. (a) The connection relationships of the query tree. (b) The connection relationships of the sentence tree of the subject tree.

5. Validating Connection Relationship

In this step, we check the connection relationships among *2STSs* in *S''* (*S''* being a sentence tree in *T''*). If *S''* passes this step, then it is a result tree. We define a connection relationship between two *2STSs u* and *v* as: the parent node of *v* is a child node of *u*. In Figures 8 (a) and (b), the connection relationships in *P* and *S''* are denoted by the arcs. Obviously, only the parent nodes of *2STSs* that have been matched in the previous step need to be checked.

The checking procedure runs bottom-up, so that it can be performed efficiently. For example, the $2STS_2$ and $2STS_3$ of *P* only match $2STS_E$ and $2STS_G$ of *S''*, and the *NP3* node plays the role of connection relationship between $2STS_2$ and $2STS_3$. Thus, we check whether the *NP_12* of $2STS_G$ exists in the child node of $2STS_E$ or not. In this case, *NP_12* does not exist in $2STS_E$, so *P* can not match *S''*. Note that, $2STS_F$ is an ancestor-descendant connection relationship (described later), and *P* can match *S''* if

the ancestor-descendant matching option is enabled. Furthermore, $2STS_1$ and $2STS_2$ of *P* only match $2STS_A$ and $2STS_E$ of *S''* as they are connected by *NP2* and *NP_9*, respectively. Therefore, *P* can match only *S''* if the ancestor-descendant matching option is enabled.

3.3 Advanced Matching Options

Our *PCRF* method provides four advanced matching options: wildcard, unordered sibling matching (*uo*), sub-tree matching (*st*), and ancestor-descendant matching (*ad*). By giving a weight to each query option, users can rank the matching results by similarity. In the following, we describe them and also show an example.

The pseudo-syntax of a structural query pattern *P* is shown in Figure 9, and its graphic structure is drawn on the left side of Figure 10 (*Pattern*). A matched tree *S* is shown on the right side of Figure 10 (*Subject*). The dotted lines point out the roots of the *2STSs* where they match one another; the gray nodes are the matched nodes.

```
<VP root="1" st_min="0" st_max="1" uo_min="0" uo_max="1">
  <NP st_min="0" st_max="2" [@role eq "agent" or @role eq "goal"]>
    <wildcard ad_max="1">
      <DM></DM>
      <N></N>
    </wildcard>
    <A></A>
    <N></N>
  </NP>
  <VE leaf="1"></VE>
</VP>
```

(Default attribute value) min: 0 max: 2 Submit Reset

Figure 9. The pseudo-syntax of a structural query.

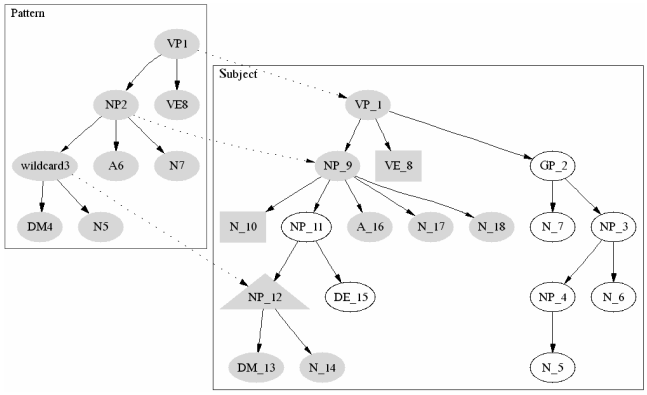


Figure 10. The tree pattern and a possible match.

We express structural patterns using XML-like syntax, where the element name denotes a node type and the attributes denote its matching options. For example, the first node *VP* in Figure 9 has "*uo_min*" and "*uo_max*" attributes, and its values are 0 and 1, respectively. This means that the node's children can be unordered, with misplacement more than or equal to "*uo_min*" positions, and less than or equal to "*uo_max*" positions. If a matching option does not occur in a node, then the default attribute values are applied. In this example, root node *VP* will have "*ad_min*" and "*ad_max*" attributes set to the default values 0 and 2, respectively. In addition, the *root* and *leaf* attributes mean the matched node must be a root or a leaf in the matched tree. Finally, the original attributes of a node, such as the *role* attribute of second node *NP* in the pattern *P*, must be surrounded by two square brackets with an additional prefix, *@*, to distinguish them from matching options. We now describe the four advance query options.

- A wildcard means an internal node with any type. To realize this option, the searching step (Section 3.2) must be modified slightly because the parent or child node of the *PCR* is unknown. Thus, the *select* command cannot specify the *PNODE* or *CNODE* constraints, and an additional bottom-up *join* operation is employed to find all possible parent nodes whose children are a superset of those of the wildcard node. In Figure 10, the *wildcard3* only matches *NP_12*.
- The unordered sibling matching (*uo*) option means sibling ordering within a *2STS* is unimportant. It can be implemented by ignoring the order of the siblings in the checking sub-tree structure step (Section 3.2). For example, although *VP1*'s children are ordered as *NP2* and *VE8*, and options *uo* and *st* have been set, *VP1* will match *VP_1*. However, if *VP1*'s "*uo_max*" attribute is set to be less than 1, *P* will not match *S*. Currently, we employ a greedy algorithm that uses the preorder numbering of *P* when determining the number of out-of-order siblings in the matched tree.
- The sub-tree matching (*st*) option allows *P* to be used to match only part of *S*. In other words, it is like partial matching in string matching. This option can be implemented by checking the number of children within the matched *2STSs* in the checking sub-tree structure step (Section 3.2). For example, *NP2* has three children and *NP_9* has five children; if the attribute value of "*st_max*" of *NP2* is less than 2, then *NP2* will not match *NP_9*.
- Finally, the ancestor-descendant matching (*ad*) option relaxes the parent-child connection relationship between two *2STSs* to an ancestor-descendant connection relationship. In Figure 8 (b), $2STS_F$ shows an ancestor-descendant connection relationship between $2STS_E$ and $2STS_G$. In Figure 10, the triangle node *NP_12* is such a node. If the attribute value of "*ad_max*" of *wildcard3* is less than 1, then *wildcard3* will not match *NP_12*. In order to realize this function, we maintain another hash table to store the child-to-parent relationships. That is, we use *CID* as a key to retrieve *PID* quickly. Thus, the ancestor-descendant connection relationship between two nodes can be determined efficiently.

3.4 Accelerating Rules

As mentioned previously, *PCRF* consists of three filtering steps. If *S* fails in any of these steps, *P* cannot match against it. It is possible to accelerate query processing by omitting some of these steps if *P* is just a parent-child pair, a *2STS*, or a path-like pattern (see queries 3, 4, and 5 in Table 1). We describe these three accelerating rules as follows:

- Case 1: If $|P| <= 2$, that is, P is either a single node or a parent-child pair, the search results can be found once the searching step is finished. In other words, we omit the fetching, checking sub-tree structure, and validating connection relationship steps. The matching process is accelerated dramatically because we do not fetch any *PCR* records from the RDBMS to the hash table and omit many steps.
- Case 2: If $|P| > 2$ and *height(P)* = 1, that is, P is a *2STS* (a parent node and its child nodes), the search results can be decided once the checking sub-tree structure step is finished (*i.e.*, we can omit the validating connection relationship step).
- Case 3: If P is a path-like pattern, then the checking sub-tree structure step can be omitted. That is, only the searching, fetching, and validating connection relationship steps need to be performed.

Recent research in XML query and text pattern matching, such as XRel [14], translates a core subset of XPath expressions into SQL expressions without using an additional filter library. However, the SQL expressions they construct might be very complex, and unable to provide flexible query ability. By applying the above three accelerating rules to our methods, we find that the complex SQL expression method has a better performance for the query patterns in Cases 2 and 3. It is a tradeoff between flexibility and efficiency. We choose flexibility rather than efficiency. In Section 4, the experiment results show that, after applying the accelerating rules to our method, a 1.5- to 8-fold performance improvement is observed. Note that, any query patterns that belong to Case 1 can be answered by using a simple SQL expression. This is a special case, and cannot be improved further.

3.5 Attributes Filtering

XPath predicates (particularly those about attribute values) are very useful because users often employ them to constrain query results. In Figure 9, the second element *NP* has a *role* attribute with a value *agent*. Thus, the corresponding node in a matched tree must also have the same attribute value.

As mentioned before, all attributes (including text data) are stored in another table. Each attribute of an element will be stored as a record in the table. A record consists of *TID*, *NID* (node identifier), attribute name, and its value. By specifying *TID*, *NID* and attribute name in a SQL expression, the attribute value of the element can be retrieved. In this setting, we have two ways to deal with the attribute evaluation problem. The first, which we currently use, is to add another filter in our *PCRF* algorithm. The second way, which is more efficient but less flexible, is to use a more complex SQL expression in the searching step. The main reason we do not choose the second way is because the SQL expression is difficult to type-correct. In contrast, by adding an additional library in the attribute filter, the typing problem can be solved easily. In our current implementations, we use the built-in operators of Perl (a programming language) to evaluate attribute values, including string, integer, floating, and even text data. To avoid fetching too many *PCR* records from the disk, we add the attribute filter immediately after the searching step.

3.6 Stream-based PCRF

Users may need a stream-based query method to query XML documents. DOM (Document Object Model) [4] is very inefficient because it puts an entire XML data tree into the main memory before it is traversed. We have developed a *stream-based PCRF* method that uses SAX (Simple API for XML) [3] event handling to improve space and time utilization.

We now describe how to modify the *PCRF* algorithm to support the streaming process. The key point is to slightly modify the searching and fetching steps of the *PCRF* algorithm. Note that, the indexing step is omitted. First, we maintain a stack to transform element events to *PCR* records. Next, each *PCR* record is inserted into a two-level hash table (described in the fetching step in Section 3.2). The constraints of constructing the hash table must still be met. When the end of a sentence event is triggered, further steps of the *PCRF* algorithm are executed to determine whether S can be matched by P. This stream-based matching procedure processes the sentences one by one. Therefore, if the sentence size is small, the matching process is efficient, and only occupies a small amount of memory. If, however, the sentence size is very large, the process might be inefficient and occupy a lot of memory. Furthermore, the question of whether to provide advanced matching options is a tradeoff between memory usage and flexible query. Again, we choose flexibility rather than memory usage. Our experiment results show a 10-fold performance improvement when our *stream-based PCRF* is compared to the usual text-based XPath query method.

3.7 Complexity Analysis

We now present the results of theoretical complexity analysis of the *PCRF* algorithm. We exclude the cost of the indexing operation associated with the RDBMS, and that of the advanced matching options from our method. In the preprocessing stage, the indexing step stores and indexes all *PCR* records of T (the subject tree) in a RDBMS. If the RDBMS uses B-tree data structure, the cost of inserting all of the *PCR* records is $O(|T| \times d\log_d|T|)$, where d denotes the minimum degree of the B-tree. Moreover, in the searching step, a SQL expression with at most $|P|$ number of *select* commands is used to query the *TAB* (the table that contains all *PCR* records). This step costs $O(|P| \times d\log_d|T|)$. The search will result in T', and often $T' << T$. Similarly, the cost of fetching the *PCR* records of size $|T'|$ from the *TAB* is $O(|T'| \times d\log_d|T|)$, and the cost of storing these *PCR* records in a hash table (with chaining) is about $O(|T'|)$. In addition, in the checking sub-tree structure step, the cost of comparing each *2STSs* of S' (S' being a sentence tree in T') with those in P is about $O(|P| \times |T'|)$. After checking the sub-tree structure, T'' is the collection of candidate trees, and often $T'' << T' << T$. Finally, each *2STSs* of S'' (S'' being a sentence tree in T'') must be checked for its connection relationship. The cost of checking the connection relationship is $O(|P| \times |T''|)$ at most.

In brief, the cost of preprocessing is $O(|T| \times d\log_d|T|)$, and the cost of a matching process is $O(|P| \times d\log_d|T|) + O(|T'| \times d\log_d|T|) + O(|T'|) + O(|P| \times |T'|) + O(|P| \times |T''|)$. Obviously, the worst case occurs when $T'' \approx T' \approx T$ (for example, query 3 in Table 1), where the total complexity is $O(|T| \times d\log_d|T|)$ at most. However, in a typical case, often $|T''| << |T'| << |T|$, so the worst case would not happen. In our experience, the searching and fetching steps take up more than 80% of the time of the matching process. That is, the checking sub-tree structure step and the validating connection relationship step are both very efficient.

4. PERPORMANCE ANALYSIS

We now present some performance results for *PCRF*, *stream-based PCRF*, and the usual XPath methods. Our setup is a Dual Intel PIII PC running at 550MHz, with a 768MB RAM, an EIDE hard disk, with an MS Windows 2000 server OS, and MSSQL 2000 installed. All programs are coded in ActivePerl-5.6.1.629. The XPath implementation we use for performance comparison is the XML-XPath-Simple module (v0.05). In addition, we compare all the methods with the linear time XPath method (LTXPath) [9]. The release tag of LTXPath is 2003-01-30.

Table 1. Six structural query expressions.

NO.	PCRF	XPath (Unordered)
1	<S> <NP> <N></N> </NP> <C></C> <D></D> <PP> <P></P> <NP> <VH></VH> <N></N> </NP> </PP> <VH></VH> <VP> <V> <VH></VH> <DE></DE> </V> <VA></VA> </VP> </S>	//Sentence[.//S[NP/N and C and D and PP[P and NP[VH and N]] and VH and VP[V[VH and DE] and VA]]]
2	<VP> <NP> <N></N> <DM></DM> </NP> </VP>	//Sentence[.//VP/NP[N and DM]]
3	<NP> <N></N> </NP>	//Sentence[.//NP/N]
4	<S> <NP></NP> <D></D> <VC></VC> <GP</GP> </S>	//Sentence[.//S[NP and D and VC and GP]]
5	<S> <VP> <NP> <NP> <N></N> </NP> </NP> </VP> </S>	//Sentence[.//S/VP/NP/NP/N]
6	<EMPTY> </EMPTY>	//Sentence[.//EMPTY]

In our experiments, we used the CKIP Chinese Treebank corpus [11] and a 115MB XML document generated by *XMLgen* [1] as the datasets. We first converted the CKIP Chinese Treebank dataset (about 20,000 sentences, 2.95MB in flat text file format) into XML format (about 9.5MB), after which we generated the *PCR* records of the XML document and stored them in a RDBMS (334 seconds). For the second dataset, it took about 4 hours and 10 minutes to finish these processes.

In order to evaluate the performance of our methods, six structural queries are used (listed in Table 1). The default max and min attribute values, for options *uo*, *st*, and *ad*, are set to 65,535[1] and 0, respectively. The running time (each an average of five runs) and memory usages (both physical and virtual memory) are stated in Table 2. Note that, none of the XPath queries contain a following-sibling axis. That is, sibling ordering of the matched trees is unimportant. Also, the SAX processor we used takes about 65.8 seconds to parse the entire Treebank dataset (without doing any additional processing), while the XPath-Simple module takes about 406 seconds to build the DOM tree alone (the physical and virtual memory usages are 523MB and 525MB, respectively).

Table 2. Performance results.

NO.	PCRF	Stream-based PCRF		Linear Time XPath [9]	XPath	
	Time(s)	Time(s)	#(Sen)	Time(s)	Time(s)	#(Sen)
1	0.7	97	1	20.3	1,641	1
2	6.4	120	910	14.1	1,308	767
3	53.6	129	17,603	24.1	1,332	17,603
4	5.4	109	38	14.4	1,662	38
5	4.3	139	193	11.2	1,298	114
6	0.016	101	0	14.5	1,270	0
Memory Requirement (MB)						
Phy.	42	7.9		170	550	
Virt.	46	4.9		172	1,050	

Our experiments show that our *PCRF* and *stream-based PCRF* are, respectively, at least 25 and 10 times faster than the XPath method. They also use less memory. In general, our *PCRF* method runs faster than the LTXPath method in general, except for query 3. However, by applying the accelerating rules, our *Acc-PCRF* method is the fastest, and is at least twice as fast as the LTXPath method. Furthermore, a 1.5 to 8.3-fold performance improvement compared to the *Acc-PCRF* method to the *PCRF* method is observed. Note that, in the *PCRF* method, the RDBMS consumes 34MB and 40MB physical and virtual memory, respectively.

For the first query, which is the most complex of the six queries, our methods (*PCRF* and *stream-based PCRF*) take 0.7 and 97 seconds, respectively, but the LTXPath and XPath methods take 20.3 and 1,641 seconds, respectively. In the sixth query, our methods take 0.016 and 101 seconds, respectively, to determine that there is no match, but the LTXPath and XPath methods take 14.5 and 1,270 seconds, respectively. This is because *PCRF* knows very early in the searching step that no tree can be matched. In contrast, the other methods need to traverse the whole document, no matter whether there is a match or not. In the processing of queries 2 and 3, our *PCRF* method takes 6.4 and 53.6 seconds, respectively; *stream-based PCRF* takes 120 and 129 seconds, respectively; LTXPath takes 14.1 and 24.1 seconds, respectively; and XPath takes about 1,308 and 1,332 seconds, respectively.

[1] 65,535 is sufficiently large for the Chinese Treebank dataset.

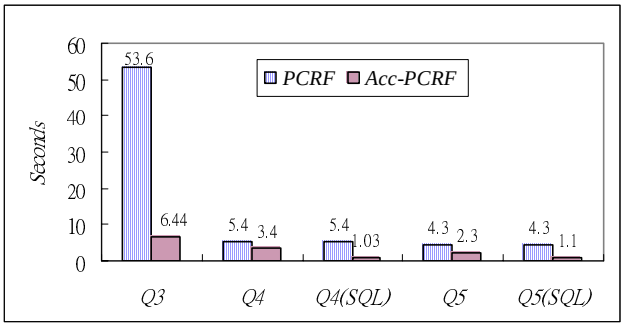


Figure 11. Performance of the PCRF and Acc-PCRF methods.

Notice that, in queries 2 and 5, both our methods return 910 and 194 sentences, respectively, but the LTXPath and XPath methods return 767 and 114 sentences, respectively. This is because we enable the ancestor-descendant matching option, so our methods return more matches. If we set both *min_ad* and *max_ad* to 0, all methods will match the same number of sentences.

To test the performance of the accelerating rules, we apply them to our *PCRF* method. The accelerated *PCRF* method is now named *Acc-PCRF*. Queries 3, 4 and 5 are accelerated to see if there is a performance difference. We show the experiment results in Figure 11. Obviously, the *Acc-PCRF* improves performance by 1.5 to 8.3 times. The matching process of *Q3* is highly accelerated. This is because *Q3* matches a large number of sentences (about 90% of sentences of the Treebank corpus contain *Q3*), and *Acc-PCRF* provides the results immediately the searching step is finished.

As mentioned in Section 3.4, we compare the performance difference between the complex SQL expression method and our *PCRF* method. The results are also shown in Figure 11, where the figures for *Q4(SQL)* and *Q5(SQL)* are from the complex SQL expression method. *Q4(SQL)* and *Q5(SQL)* clearly show that the complex SQL expression method is at least 4 times faster than the *PCRF* method. Note that, the Case 1 rule can be applied to query 3, and the results can be generated by a simple SQL expression. Therefore, *Q3* can not be further improved by using the complex SQL expression method.

We also apply *Acc-PCRF* and *stream-based PCRF* to the second dataset (115MB, generated by *XMLgen*). The query we use is:

<site><regions><namerica>
<item [*@id* = "*item20748*"]*><name></name></item>*
</namerica></regions></site>

This query is equivalent to the *Q1* in [1]. In this case, *Acc-PCRF* takes 1,516 ms to return the matched positions (*i.e., CIDs*), and the method in [1] takes 1,217 ms. Our *stream-based PCRF* takes 11.28 minutes to provide the text data of the *name* element[2], and its memory usages are about 3.1MB and 4.9MB for physical and virtual memory, respectively, but the LTXPath and XPath methods run out of memory and do not even finish[3]. Note that, the SAX processor we used takes about 369 seconds to parse this dataset, without doing any processes.

In another experiment, the accelerated B-tree XPATH method developed in [20] takes about 60 seconds to answer `Qa`, while the accelerated XPATH R-tree method [20] takes about 8 seconds. Moreover, our *Acc-PCRF* method takes only 0.828 seconds to answer `Qb` (returns the matched positions), while the *stream-based PCRF* takes about 12.25 minutes, and its memory usages are 398MB and 525MB, respectively[4]. The LTXPath takes about 3.8 minutes and its memory usages are 635MB and 1,003MB for physical and virtual memory, respectively. Because our methods do not support a descendent-axis, a wildcard node is used in the query to simulate this functionality. Note that queries `Qa` and `Qb` are equivalent, and return the same search results.

```
(Qa) //open_auction//description
(Qb) <open_auction><wildcard>
       <description></description>
     </wildcard></open_auctions>
```

5. CONCLUSIONS

We address five significant problems in NLP. To deal with these problems, we have developed a prototype system that provides exploratory search ability. The system implements two flexible and efficient structural query methods, and employs a user-friendly web-based interface. One of our methods, *PCRF*, needs to index the XML documents first, but has better search capability. The other one, *stream-based PCRF*, can perform stream-based matching over large XML documents online. Both our methods are sufficiently flexible to support several useful similarity-matching options. We also propose three accelerating rules to reduce query time.

Recent approaches to XML query and text pattern matching, such as XRel [14], also translate a core subset of XPath expressions into SQL expressions, but unlike our method, they do not use an additional filter library. To provide more flexible searching ability, the SQL expressions can be very complicated in XRel. It will be interesting to compare recent research results in XML query with those in tree-pattern matching problems, and thereby gain new research insights.

We conclude by outlining some possible lines of future research. First, powerful query languages could be designed to help users express complex approximate structural queries in XML documents. Second, more similarity-based query options and result-set operations could be devised, specifically for NLP, or for XML processing in general. Finally, an extendable XML schema could be defined to integrate multiple corpora and develop innovative NLP applications.

[2] The execution environments of [1] and ours are different. We show the time it takes for both execution environments to load the benchmark document. In [1], the platform takes 49.7 seconds; and ours takes 59.7 seconds.

[3] As the LTXPath processor does not support attributes test ability yet, so we transform all attributes of elements into sub-elements. After transformation, the document size is about 129MB. However, the LTXPath processor can answer the queries in Table 1 correctly without any transformation.

[4] Although, the execution environments of [20] and ours are also different, we still report the experimental results here.

ACKNOWLEDGMENTS

We would like to thank Dr. Keh-Jian Chen for his suggestions and insightful comments on this work. He also provided us with the CKIP Chinese Treebank corpus for our experiments. Finally, we would like to thank the reviewers who gave us many valuable suggestions.

This project is supported, in part, by the Institute of Applied Science and Engineering Research, Academia Sinica. Chia-Hsin Huang is a Ph.D. candidate in the Department of Electronic Engineering of National Taiwan University of Science and Technology, and is supported by a graduate student fellowship from Academia Sinica.

6. REFERENCES

[1] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. *The XML Benchmark Project*. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, 2001.

[2] C. Y. Chan, P. F., M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of the 18th International Conference on Data Engineering*, 2002, 235-244.

[3] D. Megginson. *SAX: A Simple API for XML*, See http://www.megginson.com/SAX/

[4] DOM, World Wide Web Consortium. *Document Object Model* (*DOM*), W3C Recommendation. See http://www.w3.org/DOM/

[5] D. Shasha, J. T. L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate Searching in Unordered Trees. In *Proc. of the 14th International Conference on Scientific and Statistical Database Management*, 2002, 89-98.

[6] F. Xia, C. H. Han, M. Palmer, and A. Joshi. Comparing Lexicalized Treebank Grammars Extracted from Chinese, Korean, and English Corpora. In *Proc. of the 2nd Chinese Language Processing Workshop*, 2000.

[7] F. Xia, M. Palmer. Comparing and Integrating Tree Adjoining Grammars. In *Proc. of the 5th International Workshop on Tree Adjoining Grammars and Related Formalism*, 2000, 25-27.

[8] F. Rizzolo and A. Mendelzon. Indexing XML data with ToXin. In *Proc. of WebDB 2001*, 2001.

[9] G. Gottlob, C. Koch, and R. Pichler, Efficient algorithms for processing XPATH Queries, In *Proc. of VLDB 2002*, 2002, 1-14.

[10] J. T. L. Wang and K. Zhang. A System for Approximate Tree Matching. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 4, 1994, 559-571.

[11] K. J. Chen, C. C. Luo, Z. M. Gao, M. C. Chang, F. Y. Chen, C. J. Chen, and C. R. Huang. The CKIP Chinese Treebank. In *Journees ATALA sur les Corpus annotes pour la syntaxe, Talana, Paris VII*, 1999.

[12] M. Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *Proc. of the 12th CIKM*, 2003, 342-349.

[13] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of VLDB 2000*, 2000, 53-64.

[14] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, Vol. 1, No. 1, 2001, 110-141.

[15] O. Streiter. Reliability in Example-based Parsing. In *Proc. of the 5th International Workshop on Tree Adjoining Grammars and Related Formalism*, 2000, 257-260.

[16] O. Streiter. Corpus-based Parsing and Treebank Development. In *Proc. of the 19th International Conference on Computer Processing of Oriental Languages*, 2001, 115-120.

[17] K. Oflazer Error-tolerant retrieval of trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 19, No: 12, 1997, 1376-1380.

[18] R. Goldman and J. Widom. DataGuides: Enable Query Formulation and Optimization in Semisturctured Databases. In *Proc. of 23rd International Conference on Very Large Data Bases*, 1997, 436-445.

[19] S. Brants, S. Dipper, S. Hansen, W. Lezius, and G. Smith. The TIGER Treebank. In *Proc. of the Workshop on Treebanks and Linguistic Theories*, Sozopol, 2002.

[20] T. Grust. Accelerating XPath Location Steps. In *Proc. of the 21st ACM SIGMOD Int'l Conference on Management of Data*, 2002, 109-120.

[21] T. Milo and D. Suciu. Index Structure for Path Expression. In *Proc. of 7th International Conference on Database Theory*, 1999, 277-295.

[22] The Linguistic Data Consortium (LDC), the web site: http://www.ldc.upenn.edu/

[23] XQuery, World Wide Web Consortium. *XML Query* (*XQuery*). W3C Recommendation. See http://www.w3.org/XML/Query

[24] XPath, World Wide Web Consortium. *XML Path Language* (*XPath*). W3C Recommendation. See http://www.w3.org/TR/XPath

[25] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. of ICDE*, 2002.