Retrieving Annotated Corpora for Corpus Annotation

Yoshida Kyôsuke*, Hashimoto Taiichi*, Tokunaga Takenobu*, Tanaka Hozumi*

* Department of Computer Science, Tokyo Institute of Technology
Tokyo Meguro Ôokayama 2-12-1, Japan
{rincho, taiichi, take, tanaka}@cl.cs.titech.ac.jp

Abstract

This paper introduces a tool *Bonsai* which supports human in annotating corpora with morphosyntactic information, and in retrieving syntactic structures stored in the database. Integrating annotation and retrieval enables users to annotate a new instance while looking back at the already annotated sentences which share the similar morphosyntactic structure. We focus on the retrieval part of the system, and describe a method to decompose a large input query into smaller ones in order to gain retrieval efficiency. The proposed method is evaluated with the Penn Treebank corpus, showing significant improvements.

1. Introduction

Statistical approach has been a main stream of natural language processing research for the last decade, and it contributed to improved performance of natural language processing systems, particularly in morphological and syntactic analysis. Large scale language resources, such as annotated corpora, played a key role in achieving these improvements. The size of corpora is important factor for reliable estimation of various parameters of statistical models. However, enlarging the corpus size while keeping its quality is not an easy task. The richer information we put in corpora, the more difficult it becomes to keep the annotation consistency, since annotating rich information generally requires human intervention. To obtain both large size and consistency of corpora, we need a tool supporting humans in building annotated corpora (Cunningham et al., 2003; Plaehn and Brants, 2000).

We are developing *Bonsai* , a tool which supports humans in annotating corpora with morphosyntactic information, and helps to retrieve syntactic structures stored in the database. Integrating annotation and retrieval enables users to annotate a new instance while looking back at the already annotated sentences sharing similar morphosyntactic structure. With help of such confirmation, it becomes easier to keep consistency of corpora.

Note that we aim to realize structure-based retrieval instead of string-based one like a UNIX command "grep". String-based retrieval would be powerful enough for annotation at morphological level, but we do not believe it is enough for syntactic level annotation. Structure-based retrieval would be a great help when annotators think of multiple candidate structures for a sentence to annotate.

In this paper, we focus on the retrieval part of our system, in particular, its efficiency in retrieving target syntactic structures. Section 2. describes the database structure for storing syntactically annotated corpora, and section 3. proposes an efficient retrieval method. Our system uses Structured Query Language (SQL), and achieves fast retrieval by decomposing large queries into small ones based on node frequency. Section 4. describes experiments conducted to evaluate the proposed method, and Section 5. concludes the paper and looks at the future work.

2. Database Structure

Since we adopt context free grammar to annotate syntactic information, each sentence is represented as a tree with nonterminal symbols as its intermediate nodes, and terminal symbols (words) as its leaf nodes as shown in Figure 1.

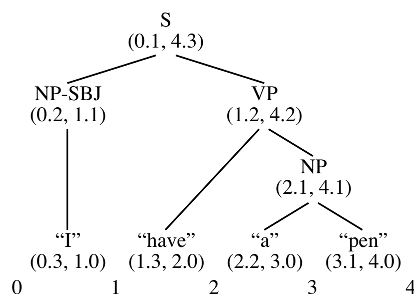

Figure 1: Example of an annotated sentence

Yoshikawa proposed a method to store and retrieve XML documents efficiently by using the relational database (RDB) (Yoshikawa et al., 2001). We use Yoshikawa's method to store tree structures in the database. Since an XML document is represented as a tree, their method is also applicable for storing syntactic trees in the database.

Yoshikawa's method uses a location index to represent hierarchical relations between nodes. The location index of a node is represented in terms of a pair of decimal numbers as shown in Figure 1. The integer part of a location index designates an index to a place between leaf nodes starting from 0. A pair of the decimal parts denotes a range of leaf nodes which is dominated by the node. For example, an index "$(0.x, 1.y)$" of a node means that this node dominates the leftmost leaf, i.e. a leaf between boundary indexes 0 and 1. The decimal part of a location index represent hierarchical relation between nodes. The decimal part is assigned by traversing a tree. When descending, the decimal place of the left number increments until a leaf node. The right number of the leaf node is assigned to 0, and the decimal place of the right number increments with ascending the tree.

Using this indexing method, hierarchical relation of two nodes can be calculated by comparing their location

indexes. For example, given nodes $N_1 = (l_1, r_1)$ and $N_2 = (l_2, r_2)$, the relations $l_1 < l_2$ and $r_2 < r_1$ suggest that N_1 properly dominates N_2, and the relation $r_1 < l_2$ suggests that N_1 locates at the left of N_2 and so on.

Following Yoshikawa's method, we define the Base table which stores the core structure of sentences as shown in Table 1.

Field Name	Description
sentenceID	Sentence identifier
nodeID	Node identifier
labelID	Index to the node label
pathID	Index to the path from root node
parentID	nodeID of the parent node
nextSibID	nodeID of the immediately right sibling
l_pos	Left location index
r_pos	Right location index

Table 1: Definition of Base table

3. Retrieving Trees

3.1. Overview of Retrieval

An input query is represented as a tree like the one given in Figure 2. Here, "$*$" denotes a wildcard which matches any structure. The sentence shown in Figure 1 will be one of the retrieved outputs for the query in Figure 2. In this case, the node labeled "have" matches the wildcard in the query.

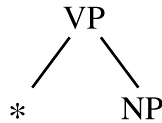

Figure 2: Example of an input query

The system provides two retrieval modes; the exact matching mode and the partial matching mode. In exact matching, a retrieved tree should include exactly the same subtree as the input query. On the other hand, partial matching allows any trees containing the query as a subtree to be retrieved. For example, both trees given in Figure 3 will be retrieved by the query in Figure 2 with the partial matching mode, but only (a) will be retrieved with the exact matching mode. Note that the subtree rooted in "SBAR-PRP" is an extra portion matching the input query in partial matching. Users can select matching mode when issuing a query.

The query in Figure 2 is actually translated into the SQL query shown in Figure 4, before the search is performed. In Figure 4, n1, n2 and n3 correspond to nodes "VP", wildcard and "NP" respectively. The value of labelID is an index to the Label table (shown in Figure 5). The values 11 and 3 correspond to "VP" and "NP" respectively. The difference between exact and partial matching is illustrated in the second and third conditions of the "where" clause. The exact matching translation requires (1) n2 is the leftmost immediate child of n1, (2) n3 is the immediately next (right) sibling of n2, and (3) there is no immediately next (right) sibling of

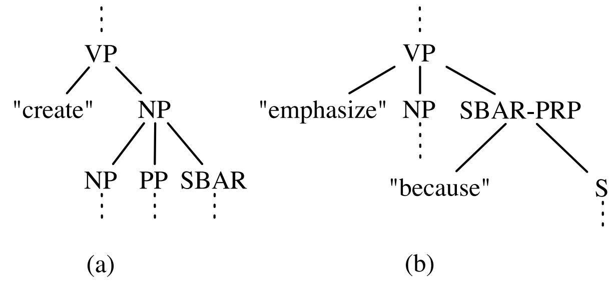

Figure 3: Exact matching and partial matching

Exact matching

```
select n1.sentenceID
from   tbl n1, tbl n2, tbl n3
where n2.parentID=n1.nodeID
  and n2.l_pos−n1.l_pos=0.1
  and n2.nextSibID=n3.nodeID
  and n3.nextSibID=NULL
  and n1.labelID=11
  and n3.labelID=3
```

Partial matching

```
select n1.sentenceID
from   tbl n1, tbl n2, tbl n3
where n2.parentID = n1.nodeID
  and n3.parentID=n1.nodeID
  and n2.r_pos<n3.l_pos
  and n1.labelID=11
  and n3.labelID=3
```

Figure 4: SQL translation

n3. These conditions define exactly the same tree as that in Figure 2. On the contrary, the partial matching translation requires only (1) n1 is the parent of both n2 and n3, and (2) n2 locates at the left of n3. The wildcard is realized by not specifying the labelID of n2.

As shown in Figure 4, nodes to be retrieved are declared in the "from" clause, and conditions on the nodes and the relations between them are put in the "where" clause in the translated SQL query. Therefore a query involving a large number of nodes generates a longer SQL query. As we will show in 3.2., we found that the retrieval speed decreased greatly as the number of nodes in a query increases. It is not practical to translate all queries into a single SQL query. We propose a method to decompose an input query into several subtrees, and translate each subtree into a SQL query.

3.2. Query Size and Retrieval Performance

To find a adequate size of subtrees, we conducted preliminary experiments with the Penn Treebank corpus (Marcus et al., 1993) which consists of 48,884 syntactically annotated sentences. All these sentences were stored in the database as described in Section 2.. Randomly extracted 8,455 subtrees from the whole Treebank were used as input queries. These queries were submitted to the system in partial matching mode. Since partial matching generally retrieves more results than exact matching, and requires more time. We investigated the influence on the retrieval time of the number of nodes in an input query, and of the maximum depth of a query.

Table 2 shows the relation between the number of nodes in a query and the corresponding retrieval time. We can see that the average retrieval time significantly increases when the number of nodes in the input query exceeds 17. Considering the maximum retrieval time as well, the next gap is found between 12 and 13. The third gap is between 5 and

No. of nodes	Retrieval time [sec] Ave.	Max.	Min.	Ave. outputs	No. of queries
2	2.04	10.6	0.001	24,330	65
3	1.59	11.3	0.001	9,855	99
4	0.84	11.5	0.002	2,189	168
5	0.38	10.1	0.002	337	281
6	0.19	6.42	0.002	65.1	437
7	0.12	1.64	0.003	17.0	611
8	0.09	1.64	0.003	7.16	762
9	0.07	1.35	0.004	3.32	860
10	0.06	1.31	0.008	1.91	902
11	0.05	1.31	0.009	1.36	901
12	0.07	1.32	0.010	1.13	869
13	0.11	3.84	0.012	1.04	801
14	0.18	9.36	0.018	1.01	678
15	0.28	45.5	0.041	1.00	498
16	0.56	2.25	0.065	1.00	300
17	1.94	6.12	0.27	1.00	140
18	7.51	18.1	1.09	1.00	47
19	31.1	47.5	8.02	1.00	10
20	161	186	135	1.00	3
21 ~	1,108	3,395	298	1.00	23

Table 2: Number of nodes in a query and retrieval time

6. This last gap can be explained by a huge number of retrieved results rather than the complex structure of an input query.

Table 3 shows the relation between the maximum depth of a query and the retrieval time. The maximum depth of a query and the number of nodes in the query correlate to a certain extent, thus queries with depth 7 require longer retrieval time. Comparing the average number of outputs and number of queries in Table 2 and 3, we can see that queries with depth 7 in Table 3 roughly correspond to the queries including more than 13 nodes in Table 2.

This observation leads us to a criterion for decomposing the input query into a set of subtrees whose number of nodes ranges between 6 and 12.

Max. depth	Retrieval time Ave.	Max.	Min.	Ave. outputs	No. of queries
1	2.08	10.63	0.001	19,092	134
2	0.34	11.33	0.001	609	694
3	0.14	45.45	0.002	24.6	2,544
4	0.16	1.64	0.007	38.6	458
5	0.13	1.35	0.007	8.3	760
6	0.16	18.1	0.007	2.0	1,680
7	2.60	3,394	0.020	1.0	2,185

Table 3: Maximum depth of a query and retrieval time

3.3. Decomposing a Query

Based on the preliminary experimental results, we propose a method to decompose a large query into small subtrees consisting of 6 to 12 nodes, and translate them into SQL queries. In this method, another important factor affecting the performance is the order of subqueries which will be performed sequentially. It is obvious that less the number of retrieved outputs in the earlier queries is preferable. To achieve this, we obtain statistics of the corpus in advance and use that information for decomposing large queries.

(1) Label table

label	labelID	freq
S	1	106,091
NP-SBJ	2	94,319
VP	11	179,161
"Indian"	12501	36
"movie"	1820	74
"star"	2971	58
...	...	...

(2) P-C table

parent	children	freq
S	NP-SBJ, VP	41,604
NP-SBJ	"Indian", NP	13
NP	"movie", "star"	25
...	...	...

Figure 5: Example of Label table and P-C table

Figure 5 shows examples of a Label table and a P-C table. A Label table stores relations between labelIDs and node labels together with the frequency of labels appearing in the corpus. A P-C table stores context free rules present in the corpus. The "parent" field and the "children" field correspond to the left hand side symbol and the right hand side symbols of a context free rule. For example, the first record of the P-C table in Figure 5 corresponds to a rule "S → NP-SBJ, VP". The "freq" field denotes the frequency of the rule in the corpus.

Using the statistics, we define a frequency of a node $frq(n)$ in a query as follows. $Frq(n)$ of a leaf node n is the frequency of n in the Label table. $Frq(n)$ of an intermediate node is the frequency of the rule in the P-C table, the "parent" field of which is n.

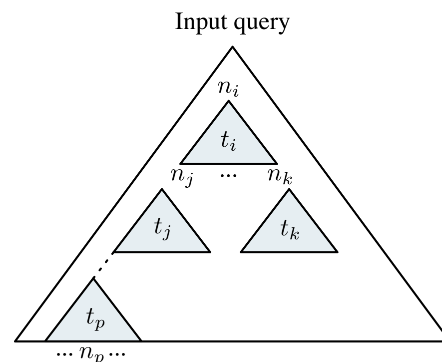

Figure 6: Decomposing an input query

The decomposition algorithm first finds node n with the least $frq(n)$ value, and then identifies a minimum subtree that includes node n. When n is an intermediate node, a minimum subtree rooted in n is adopted. Figure 6 shows an input query example. Suppose n_j is a node with the least $frq(n_j)$ value in this query, subtree t_j becomes the initial subtree. If n_p is the least frequent node, then t_p would be the initial subtree. Note that a minimum subtree cor-

responds to a context free rule and is illustrated as a gray triangle in Figure 6.

We start with this initial subtree, and extend it so as to have 6 to 12 nodes. The extension is performed by adding the minimum subtree which shares a node with the initial subtree. Suppose t_j in Figure 6 is the initial subtree, possible extension will be done by adding t_i or one of subtrees whose root node is a leaf of t_j. Among these, a minimum subtree which includes a node with the least frq value is selected. However, if adding a subtree makes the total number of nodes exceed 12, that subtree is not selected even though it has a node with the least frq value. The extension continues while the total number of nodes in the subtree does not exceeds 12. The resulting subtree is then translated into a SQL query as described in 3.1..

By adding less frequent nodes as the extension, we expect less retrieval outputs. The second retrieval is performed against the result of the first retrieval. The second subquery is generated by extending the first query subtree. As in the first query, we first find an initial minimum subtree which includes a node with the least frq value. In addition, the minimum subtree should share a node with the first query subtree. Note that, this shared node is already instantiated to a certain node of a sentence in the corpus, since we perform the second retrieval against the result of the first retrieval. This initial minimum subtree is extended in the same manner as the first subquery. Following subqueries are generated until they cover the whole input query.

4. Experiments

We conducted experiments to evaluate the proposed method. Queries are categorized with respect to the number of nodes, ranging from 13 to 25. Queries with less than 13 nodes were excluded, because our system does not decompose such queries. We used 100 queries (subtrees) in each category which were randomly extracted from the Penn Treebank corpus.

First, we compare the proposed method with retrieving by a single SQL query without decomposition. Results are shown in Table 4. For exact matching, the proposed method keeps constant retrieval time despite the increase in the number of nodes. In contrast, the retrieval time of the single SQL method increases significantly for the queries with more than 16 nodes.

For partial matching, the proposed method shows stable retrieval time again, but the single SQL method degrades the retrieval speed when the number of nodes increases. We aborted in measuring the retrieval time for queries with more than 19 nodes, because of it took so long time.

To verify the effect of ordering subqueries, another baseline was introduced, that is, decomposing queries without considering the order of subqueries. According to this baseline, an input query is decomposed into subtrees from the root node in top-down manner. This method is labeled as “Top-down” in Table 4. In the top-down method, a subtree is expanded from the root node by adding a minimum subtree in depth-first and left-to-right manner. The expansion continues while the number of nodes does not exceed 12. The constraint on the number of nodes is the same as the proposed method. The next subtree is searched for from the left-bottom node of the initial subtree in the same manner.

Comparing to the single SQL method, the top-down method shows stable retrieval time in both exact matching and partial matching. However, on average, the proposed method is 3 times faster in exact matching and 5 times faster in partial matching than the top-down method. According to the experimental results, we can conclude that decomposing and ordering of queries significantly improve the retrieval speed.

No. of node	Retrieval time [sec]					
	Proposed		Single SQL		Top-down	
	exact	partial	exact	partial	exact	partial
13	0.060	0.023	0.151	0.642	0.305	0.098
14	0.058	0.039	0.174	2.855	0.398	0.237
15	0.050	0.034	0.386	9.744	0.088	0.121
16	0.035	0.026	1.972	17.70	0.134	0.101
17	0.036	0.027	12.67	189.9	0.159	0.204
18	0.029	0.027	9.800	317.4	0.106	0.088
19	0.034	0.028	41.45	986.4	0.125	0.143
20	0.040	0.109	75.80	N/A	0.074	0.636
21	0.073	0.218	298.3	N/A	0.147	0.988
22	0.058	0.111	787.3	N/A	0.090	0.449
23	0.044	0.048	723.6	N/A	0.094	0.390
24	0.040	0.040	1369	N/A	0.101	0.249
25	0.040	0.041	3395	N/A	0.086	0.280

Table 4: Evaluation results

5. Conclusions and Future Work

In the experiments, we found that the number of nodes in a query affects the retrieval time. We estimate the optimal number of nodes as between 6 and 12 for the Penn Treebank corpus when decomposing a large query. However, this parameter might be different for the other corpus. We need to investigate a method to estimate the optimal parameters automatically given a corpus.

6. References

H. Cunningham, V. Tablan, K. Bontcheva, and M. Dimitrov. 2003. Language engineering tools for collaborative corpus annotation. In *Proceedings of Corpus Linguistics 2003*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2).

Oliver Plaehn and Thorsten Brants. 2000. Annotate – an efficient interactive annotation tool. In *Proceedings of the Sixth Conference on Applied Natural Language Processing ANLP-2000*, Seattle, WA.

Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. 2001. Xrel: A path-based approach to storage and retrieval of xml documents using relational database. *ACM Transactions on Internet Technology*, 1(1).