An Indexing Scheme for Typed Feature Structures

Takashi NINOMIYA,[†‡] **Takaki MAKINO,**[#¶] and **Jun'ichi TSUJII**[†‡]
[†]Department of Computer Science, University of Tokyo
[‡]CREST, Japan Science and Technology Corporation
[#]Department of Complexity Science and Engineering, University of Tokyo[*]
[¶]BSI, RIKEN
e-mail: {ninomi, mak, tsujii}@is.s.u-tokyo.ac.jp

Abstract

This paper describes an indexing substrate for typed feature structures (ISTFS), which is an efficient retrieval engine for typed feature structures. Given a set of typed feature structures, the ISTFS efficiently retrieves its subset whose elements are unifiable or in a subsumption relation with a query feature structure. The efficiency of the ISTFS is achieved by calculating a unifiability checking table prior to retrieval and finding the best index paths dynamically.

1 Introduction

This paper describes an indexing substrate for typed feature structures (ISTFS), which is an efficient retrieval engine for typed feature structures (TFSs) (Carpenter, 1992). Given a set of TFSs, the ISTFS can efficiently retrieve its subset whose elements are unifiable or in a subsumption relation with a query TFS.

The ultimate purpose of the substrate is aimed at the construction of large-scale intelligent NLP systems such as IR or QA systems based on unification-based grammar formalisms (Emele, 1994). Recent studies on QA systems (Harabagiu et al., 2001) have shown that systems using a wide-coverage noun taxonomy, quasi-logical form, and abductive inference outperform other bag-of-words techniques in accuracy. Our ISTFS is an indexing substrate that enables such knowledge-based systems to keep and retrieve TFSs, which can represent symbolic structures such as quasi-logical forms or a taxonomy and the output of parsing of unification-based grammars for a very large set of documents.

The algorithm for our ISTFS is concise and efficient. The basic idea used in our algorithm uses a necessary condition for unification.

(Necessary condition for unification) Let $\mathtt{Path}_F$ be the set of all feature paths defined in TFS F, and $FollowedType(\pi,F)$ be the type assigned to the node reached by following path π.[1] If two TFSs F and G are unifiable, then $FollowedType(\pi,F)$ and $FollowedType(\pi,G)$ are defined and unifiable for all $\pi \in (\mathtt{Path}_F \cup \mathtt{Path}_G)$.

The Quick Check algorithm described in (Torisawa and Tsujii, 1995; Malouf et al., 2000) also uses this condition for the efficient checking of unifiability between two TFSs. Given two TFSs and statically determined paths, the Quick Check algorithm can efficiently determine whether these two TFSs are non-unifiable or there is some uncertainty about their unifiability by checking the path values. It is worth noting that this algorithm is used in many modern unification grammar-based systems, e.g., the LKB system (Copestake, 1999) and the PAGE system (Kiefer et al., 1999).

Unlike the Quick Check algorithm, which checks unifiability between two TFSs, our ISTFS checks unifiability between one TFS and n TFSs. The ISTFS checks unifiability by using dynamically determined paths, not statically determined paths. In our case, using only statically determined paths might extremely degrades the system performance. Suppose that any statically determined paths are not defined in the query TFS. Because there is no path to be used for checking unifiability, it is required to unify a query with every element of the data set. It should also be noted that using all paths defined in a query TFS severely degrades the system performance because a TFS is a huge data structure comprised of hundreds of nodes and paths, i.e., most of the retrieval time will be consumed in filtering. The

[*] This research is partially funded by JSPS Research Fellowship for Young Scientists.

[1] More precisely, $FollowedType(\pi,F)$ returns the type assigned to the node reached by following π from the root node of $FSPATH(\pi,F)$, which is defined as follows.

$$FSPATH(\pi,F) = F \sqcup PV(\pi)$$
$$PV(\pi) = \left\{ \begin{array}{l} \text{the least feature structure where} \\ \text{path } \pi \text{ is defined} \end{array} \right.$$

That is, $FollowedType(\pi,F)$ might be defined even if π does not exist in F.

ISTFS dynamically finds the index paths in order of highest filtering rate. In the experiments, most 'non-unifiable' TFSs were filtered out by using only a few index paths found by our optimization algorithm.

2 Algorithm

Briefly, the algorithm for the ISTFS proceeds according to the following steps.

1. When a set of *data TFSs* is given, the ISTFS prepares a *path value table* and a *unifiability checking table* in advance.
2. When a *query TFS* is given, the ISTFS retrieves TFSs which are unifiable with the query from the set of data TFSs by performing the following steps.
   (a) The ISTFS finds the index paths by using the unifiability checking table. The index paths are the most restrictive paths in the query in the sense that the set of the data TFSs can be limited to the smallest one.
   (b) The ISTFS filters out TFSs that are non-unifiable by referring to the values of the index paths in the path value table.
   (c) The ISTFS finds exactly unifiable TFSs by unifying the query and the remains of filtering one-by-one, in succession.

This algorithm can also find the TFSs that are in the subsumption relation, i.e., more-specific or more-general, by preparing subsumption checking tables in the same way it prepared a unifiability checking table.

2.1 Preparing Path Value Table and Unifiability Checking Table

Let $\mathscr{D}(=\{F_1,F_2,\ldots,F_n\})$ be the set of data TFSs. When $\mathscr{D}$ is given, the ISTFS prepares two tables, a path value table $D_{\pi,\sigma}$ and a unifiability checking table $U_{\pi,\sigma}$, for all $\pi \in \mathtt{Path}_{\mathscr{D}}$ and $\sigma \in \mathtt{Type}$. [2] A TFS might have a cycle in its graph structure. In that case, a set of paths becomes infinite. Fortunately, our algorithm works correctly even if the set of paths is a subset of all existing paths. Therefore, paths which might cause an infinite set can be removed from the path set. We define the path value table and the unifiability checking table as follows:

$$D_{\pi,\sigma} \equiv \{F | F \in \mathscr{D} \wedge \mathit{FollowedType}(\pi,F) = \sigma\}$$

$$U_{\pi,\sigma} \equiv \sum_{\substack{\tau \\ (\tau \in \mathtt{Type} \wedge \sigma \sqcup \tau \text{ is defined})}} |D_{\pi,\tau}|$$

[2] $\mathtt{Type}$ is a finite set of types.

Assuming that σ is the type of the node reached by following π in a query TFS, we can limit $\mathscr{D}$ to a smaller set by filtering out 'non-unifiable' TFSs. We have the smaller set:

$$U'_{\pi,\sigma} \equiv \bigcup_{\substack{\tau \\ (\tau \in \mathtt{Type} \wedge \sigma \sqcup \tau \text{ is defined})}} D_{\pi,\tau}$$

$U_{\pi,\sigma}$ corresponds to the size of $U'_{\pi,\sigma}$. Note that the ISTFS does not prepare a table of $U'_{\pi,\sigma}$ statically, but just prepares a table of $U_{\pi,\sigma}$ whose elements are integers. This is because the system's memory would easily be exhausted if we actually made a table of $U'_{\pi,\sigma}$. Instead, the ISTFS finds the best paths by referring to $U_{\pi,\sigma}$ and calculates only $U'_{\pi,\sigma}$ where π is the best index path.

Suppose the type hierarchy and $\mathscr{D}$ depicted in Figure 1 are given. The tables in Figure 2 show $D_{\pi,\sigma}$ and $U_{\pi,\sigma}$ calculated from Figure 1.

2.2 Retrieval

In what follows, we suppose that $\mathscr{D}$ was given, and we have already calculated $D_{\pi,\sigma}$ and $U_{\pi,\sigma}$.

Finding Index Paths

The best index path is the most restrictive path in the query in the sense that $\mathscr{D}$ can be limited to the smallest set by referring to the type of the node reached by following the index path in the query.

Suppose a query TFS X and a constant k, which is the maximum number of index paths, are given. The best index path in $\mathtt{Path}_X$ is path π such that $U_{\pi,\sigma}$ is minimum where σ is the type of the node reached by following π from the root node of X. We can also find the second best index path by finding the path π s.t. $U_{\pi,\sigma}$ is the second smallest. In the same way, we can find the i-th best index path s.t. $i \leq k$.

Filtering

Suppose k best index paths have already been calculated. Given an index path π, let σ be the type of the node reached by following π in the query. An element of $\mathscr{D}$ that is unifiable with the query must have a node that can be reached by following π and whose type is unifiable with σ. Such TFSs $(= U'_{\pi,\sigma})$ can be collected by taking the union of $D_{\pi,\tau}$, where τ is unifiable with σ. For each index path, $U'_{\pi,\sigma}$ can be calculated, and the $\mathscr{D}$ can be limited to the smaller one by taking their intersection. After filtering, the ISTFS can find exactly unifiable TFSs by unifying the query with the remains of filtering one by one.

Suppose the type hierarchy and $\mathscr{D}$ in Figure 1 are

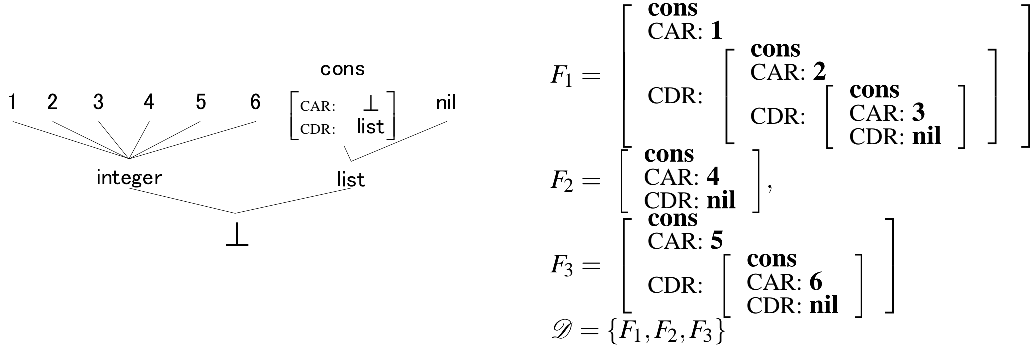

Figure 1: An example of a type hierarchy and TFSs

$D_{\pi,\sigma}$

π	σ: ⊥	integer	1	2	3	4	5	6	list	cons	nil
ε	·	·	·	·	·	·	·	·	·	$\{F_1,F_2,F_3\}$	·
CAR:	·	·	$\{F_1\}$	·	·	$\{F_2\}$	$\{F_3\}$	·	·	·	·
CDR:	·	·	·	·	·	·	·	·	·	$\{F_1,F_3\}$	$\{F_2\}$
CDR:CAR:	·	·	·	$\{F_1\}$	·	·	·	$\{F_3\}$	·	·	·
CDR:CDR:	·	·	·	·	·	·	·	·	·	$\{F_1\}$	$\{F_3\}$
CDR:CDR:CAR:	·	·	·	·	$\{F_1\}$	·	·	·	·	·	·
CDR:CDR:CDR:	·	·	·	·	·	·	·	·	·	·	$\{F_1\}$

· is an empty set.

$U_{\pi,\sigma}$

π	σ: ⊥	integer	1	2	3	4	5	6	list	cons	nil
ε	3	0	0	0	0	0	0	0	3	*3	0
CAR:	3	*3	1	0	0	1	1	0	0	0	0
CDR:	3	0	0	0	0	0	0	0	3	*2	1
CDR:CAR:	2	2	0	1	0	0	0	*1	0	0	0
CDR:CDR:	2	0	0	0	0	0	0	0	*2	1	1
CDR:CDR:CAR:	1	1	0	0	1	0	0	0	0	0	0
CDR:CDR:CDR:	1	0	0	0	0	0	0	0	1	0	1

Figure 2: An example of $D_{\pi,\sigma}$ and $U_{\pi,\sigma}$

	QuerySetA	*QuerySetB*
# of the data TFSs	249,994	249,994
Avg. # of unifiables	68,331.58	1,310.70
Avg. # of more specifics	66,301.37	0.00
Avg. # of more generals	0.00	0.00

Table 1: The average number of data TFSs and answers for *QuerySetA* and *QuerySetB*

given, and the following query X is given:

$$X = \begin{bmatrix} \textbf{cons} \\ \text{CAR: } \textbf{integer} \\ \text{CDR: } \begin{bmatrix} \textbf{cons} \\ \text{CAR: } \textbf{6} \\ \text{CDR: } \textbf{list} \end{bmatrix} \end{bmatrix}$$

In Figure 2, $U_{\pi,\sigma}$ where the π and σ pair exists in the query is indicated with an asterisk. The best index paths are determined in ascending order of $U_{\pi,\sigma}$ indicated with an asterisk in the figure. In this example, the best index path is CDR:CAR: and its corresponding type in the query is **6**. Therefore the unifiable TFS can be found by referring to $D_{\text{CDR:CAR:},\mathbf{6}}$, and this is $\{F_3\}$.

3 Performance Evaluation

We measured the performance of the ISTFS on a IBM xSeries 330 with a 1.26-GHz PentiumIII processor and a 4-GB memory. The data set consisting of 249,994 TFSs was generated by parsing the

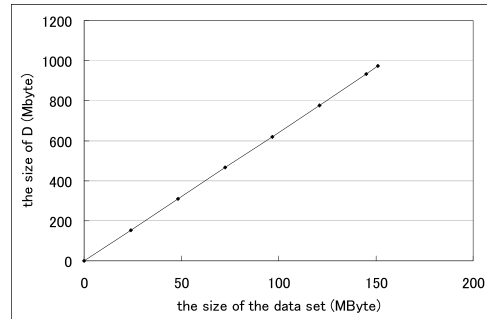

Figure 3: The size of $D_{\pi,\sigma}$ for the size of the data set

800 bracketed sentences in the Wall Street Journal corpus (the first 800 sentences in Wall Street Journal 00) in the Penn Treebank (Marcus et al., 1993) with the XHPSG grammar (Tateisi et al., 1998). The size of the data set was 151 MB. We also generated two sets of query TFSs by parsing five randomly selected sentences in the Wall Street Journal corpus (*QuerySetA* and *QuerySetB*). Each set had 100 query TFSs. Each element of *QuerySetA* was the daughter part of the grammar rules. Each element of *QuerySetB* was the right daughter part of the grammar rules whose left daughter part is instantiated. Table 1 shows the number of data TFSs and the av-

erage number of unifiable, more-specific and more-general TFSs for *QuerySetA* and *QuerySetB*. The total time for generating the index tables (i.e., a set of paths, the path value table ($D_{\pi,\sigma}$), the unifiability checking table ($U_{\pi,\sigma}$), and the two subsumption checking tables) was 102.59 seconds. The size of the path value table was 972 MByte, and the size of the unifiability checking table and the two subsumption checking tables was 13 MByte. The size of the unifiability and subsumption checking tables is negligible in comparison with that of the path value table. Figure 3 shows the growth of the size of the path value table for the size of the data set. As seen in the figure, it grows proportionally.

Figures 4, 5 and 6 show the results of retrieval time for finding unifiable TFSs, more-specific TFSs and more-general TFSs respectively. In the figures, the X-axis shows the number of index paths that are used for limiting the data set. The ideal time means the unification time when the filtering rate is 100%, i.e., our algorithm cannot achieve higher efficiency than this optimum. The overall time is the sum of the filtering time and the unification time. As illustrated in the figures, using one to ten index paths achieves the best performance. The ISTFS achieved 2.84 times speed-ups in finding unifiables for *QuerySetA*, and 37.90 times speed-ups in finding unifiables for *QuerySetB*.

Figure 7 plots the filtering rate. In finding unifiable TFSs in *QuerySetA*, more than 95% of non-unifiable TFSs are filtered out by using only three index paths. In the case of *QuerySetB*, more than 98% of non-unifiable TFSs are filtered out by using only one index path.

4 Discussion

Our approach is said to be a variation of *path indexing*. Path indexing has been extensively studied in the field of automated reasoning, declarative programming and deductive databases for term indexing (Sekar et al., 2001), and was also studied in the field of XML databases (Yoshikawa et al., 2001). In path indexing, all existing paths in the database are first enumerated, and then an index for each path is prepared. Other existing algorithms differed from ours in i) data structures and ii) query optimization. In terms of data structures, our algorithm deals with typed feature structures while their algorithms deal with PROLOG terms, i.e., variables and instantiated terms. Since a type matches not only the same type or variables but unifiable types, our problem is much more complicated. Yet, in our system, hierarchical relations like a taxonomy can easily be represented by types. In terms of query optimization, our algorithm dynamically selects index paths to minimize the searching cost. Basically, their algorithms take an intersection of candidates for all paths in a query, or just limiting the length of paths (McCune, 2001). Because such a set of paths often contains many paths ineffective for limiting answers, our approach should be more efficient than theirs.

5 Conclusion and Future Work

We developed an efficient retrieval engine for TFSs, ISTFS. The efficiency of ISTFS is achieved by calculating a unifiability checking table prior to retrieval and finding the best index paths dynamically.

In future work, we are going to 1) minimize the size of the index tables, 2) develop a feature structure DBMS on a second storage, and 3) incorporate structure-sharing information into the index tables.

References

B. Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, U.K.

A. Copestake. 1999. The (new) LKB system. Technical report, CSLI, Stanford University.

M. C. Emele. 1994. TFS – the typed feature structure representation formalism. In *Proc. of the International Workshop on Sharable Natural Language Resources (SNLR-1994)*.

S. Harabagiu, D. Moldovan, M. Paşca, R. Mihalcea, M. Surdeanu, R. Bunescu, R. Gîrju, V. Rus, and Morărescu. 2001. Falcon: Boosting knowledge for answer engines. In *Proc. of TREC 9*.

B. Kiefer, H.-U. Krieger, J. Carroll, and R. Malouf. 1999. A bag of useful techniques for efficient and robust parsing. In *Proc. of ACL-1999*, pages 473–480, June.

R. Malouf, J. Carroll, and A. Copestake. 2000. Efficient feature structure operations without compilation. *Journal of Natural Language Engineering*, 6(1):29–46.

M. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330.

W. McCune. 2001. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Automated Reasoning*, 18(2):147–167.

R. Sekar, I. V. Ramakrishnan, and A. Voronkov. 2001. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier Science Publishers.

Y. Tateisi, K. Torisawa, Y. Miyao, and J. Tsujii. 1998. Translating the XTAG English grammar to HPSG. In *Proc. of TAG+4*, pages 172–175.

K. Torisawa and J. Tsujii. 1995. Compiling HPSG-style grammar to object-oriented language. In *Proc. of NLPRS-1995*, pages 568–573.

M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. 2001. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141.

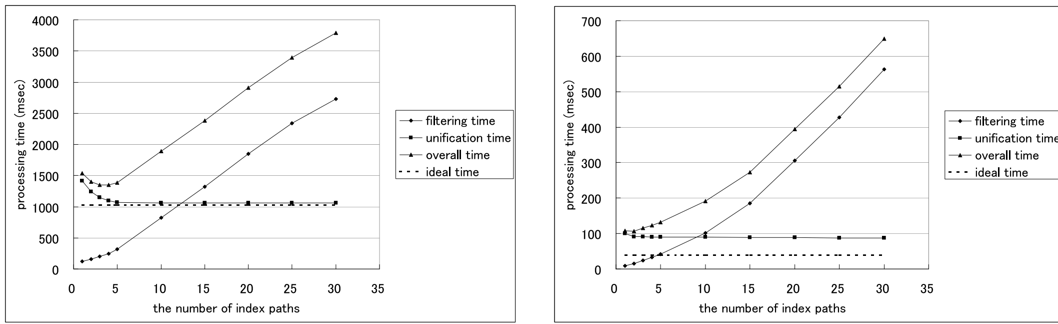

Figure 4: Average retrieval time for finding unifiable TFSs: *QuerySetA* (left), *QuerySetB* (right)

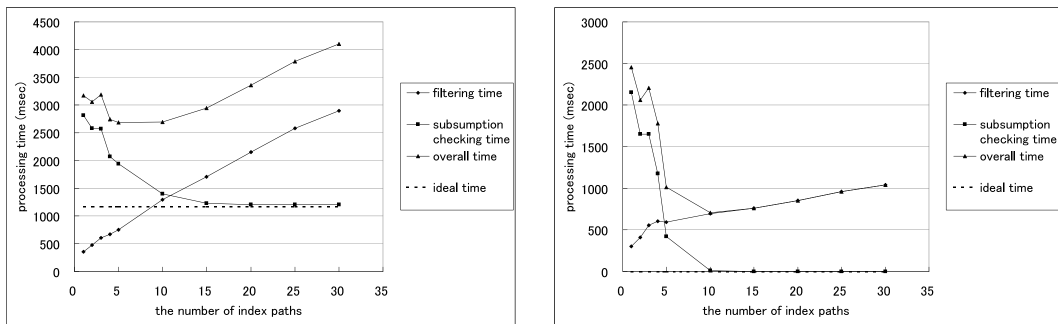

Figure 5: Average retrieval time for finding more-specific TFSs: *QuerySetA* (left), *QuerySetB* (right)

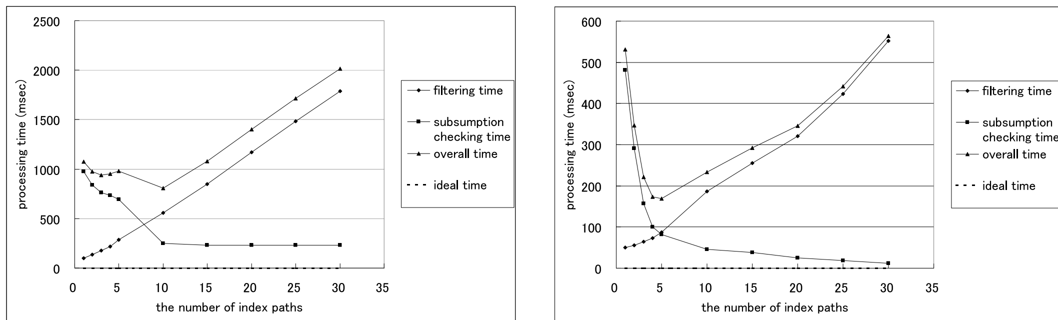

Figure 6: Average retrieval time for finding more-general TFSs: *QuerySetA* (left), *QuerySetB* (right)

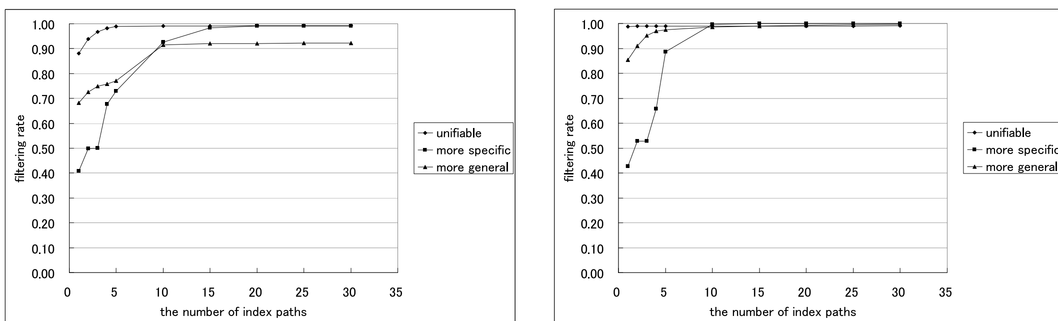

Figure 7: Filtering rate: *QuerySetA* (left) and *QuerySetB* (right)