Error-Tolerant Retrieval of Trees

Kemal Oflazer

Abstract—We present an efficient algorithm for retrieving from a database of trees, all trees that differ from a given query tree by a small number additional or missing leaves, or leaf label changes. It has natural language processing applications in searching for matches in example-based translation systems, and retrieval from lexical databases containing entries of complex feature structures. For large randomly generated synthetic tree databases (some having tens of thousands of trees), and on databases constructed from Wall Street Journal treebank, it can retrieve for trees with a small error, in a matter of tenths of a second to about a second.

Index Terms—Example-based machine translation, approximate tree comparison, retrieval from lexical databases, tree databases.

1 Introduction

Recent approaches in machine translation known as example-based machine translation rely on searching a database of previous translations of sentences or fragments of sentences, and composing a translation from translations of any matching examples [4], [6]. The example database may consist of paired text fragments, or trees [6]. Most often, exact matches for new sentences or fragments will not be in the database, and one has to consider examples that are "similar" to the sentence or fragment in question. This involves associatively searching the database for trees that are "close" to the query tree. The paper first presents an approximate tree retrieval problem in an abstract setting and presents an algorithm for it. The algorithm relies on linearizing the trees and then compressing them into a trie structure. The problem then reduces to sequence correction problem akin to the string correction problem. The trie is used with an error-tolerant finite state recognition algorithm [5] to find trees that are "close" to a query tree in terms of additional or missing leaves or leaf label changes.

2 Approximate Tree Retrieval

In this paper, we address the computational problem of retrieving from a database, trees that are "close" to a given query tree where closeness is based on how similar to each other, the sets of the individual leaves of the trees are. The trees that we consider have labeled terminal and nonterminal nodes. We consider two trees "close" if we can

1) add/delete a small number of leaves to/from one of the trees (and where necessary, insert or delete internal nodes), and/or
2) change the label of a small number of leaves in one of the trees to get the second tree.

Thus, we do not allow for any relabeling of any internal node, but an internal node may be inserted if the insertion of leaf requires it, deleted if the leaf that is being removed is its last leaf descendant. A pair of such "close" trees in the context of representing the constituent structure of natural language sentences, is depicted in Fig. 1.

2.1 Linearization of Trees

Before proceeding any further we would like to define the concepts that we will be using in the following sections: We identify each leaf node in a tree with an ordered *vertex list* $(v_0, v_1, v_2, \cdots, v_d)$ where each v_i is the label of a vertex from the root v_0 to the leaf v_d at depth d, and for $0 \le i < d$, v_i is the parent of v_{i+1}. A tree with n leaves is represented by a *vertex list sequence* $VLS = V_1, V_2, \cdots, V_n$ where each $V_j = v_0^j, v_1^j, v_2^j, v_3^j, \ldots, v_{d_j}^j$ corresponds to a vertex list for a leaf at level d_j. This sequence is constructed by taking the left-to-right order of the leaves.[1]

For instance, the first tree in Fig. 1 would be represented by the vertex list sequence:

```
((S,NP,Det,a), (S,NP,NP,Adj,black),
 (S,NP,NP,N,cat),
 (S,VP,V,chased), (S,VP,NP,Det,the),
 (S,VP,NP,NP,Adj,little), (S,VP,NP,NP,N, mouse))
```

2.2 Distance Between Two Trees

We define the distance between two trees according to the *structural differences* or *differences in leaf labels.* We consider an extra or a missing leaf as a structural difference. If, however, both trees have leaves whose vertex lists match in all but the last (leaf vertex) label, we consider this as a difference in leaf labels. For instance, in Fig. 2, there is an extra leaf in tree (b) in comparison with the tree in (a), while tree (c) has a leaf label difference. We associate the following costs associated with these differences:

- If both trees have a leaf whose vertex list matches in all but the last (leaf vertex) label, we assign a label difference error of C.
- If a certain leaf is missing in one of the trees but exists in the other one, then we assign a cost S for this structural difference.

The intuition is that two trees which have exactly the same structure except for a leaf label difference are more "close" than two trees which differ by an additional leaf (and any additional internal nodes that come with the leaf). Thus C should be chosen to be less than S.

2.3 Converting a Set of Trees Into a Trie

A *tree database D* consists of a set of trees $T_1, T_2, \cdots, T_k$, each T_i being represented by a vertex list sequence for a tree. We convert this set (of VLSs) into a trie data structure. This trie will compress any possible redundancies in the prefixes of the vertex list sequences to achieve a certain compaction which helps during searching.[2] For instance, the three trees in Fig. 2 can be represented as a trie as shown in Fig. 3. The edge labels along the path to a leaf when concatenated in order gives the vertex list sequence for a tree, e.g., `((a,b,a,x), (a,b,c), (a,b,k), (a,e))` represents the tree (a) in Fig. 2.

2.4 Error-Tolerant Matching in the Trie

Our goal is not the retrieval of trees that match a query tree exactly but rather the retrieval of trees that match approximately (with the

• *The author is with the Department of Computer Engineering and Information Science, Bilkent University, Bilkent, Ankara, Turkey. E-mail: ko@cs.bilkent.edu.tr.*

Manuscript received 10 Oct. 1996; revised 6 Oct. 1997. Recommended for acceptance by T. Ishida.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 105851.

1. In the case where hierarchical relationships among the nodes are to be represented and left-to-right order of the leaves is immaterial (such as in lexical databases), then the vertex lists can be ordered using a lexicographic ordering, i.e., V_i is *lexicographically less than* V_{i+1}, based on the total ordering of the vertex labels.

2. Note that it is possible to obtain more space reduction by sharing prefixes of vertex label sequences by introducing intermediate nodes to the trie (e.g., in Fig. 3, the prefix (a, b) at the second level can potentially be shared, but these do not improve the execution time.

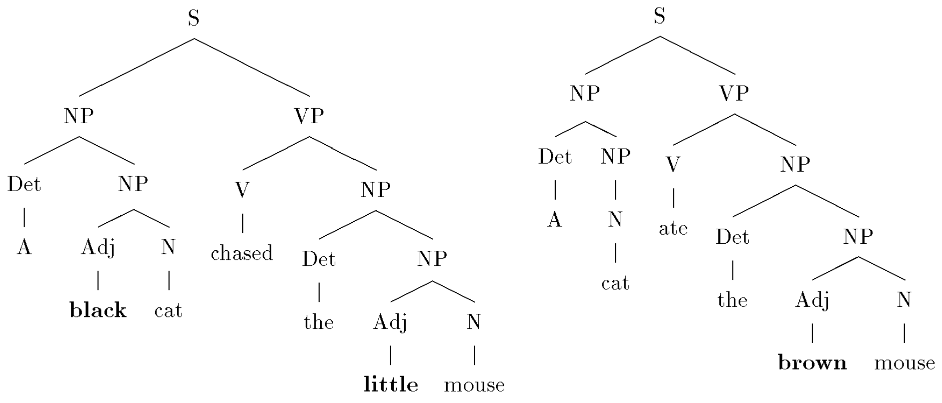

Fig. 1. Trees that are "close" to each other.

differences between the trees described earlier.) Given a vertex list sequence for a query tree, exact match over the trie can be performed using the standard techniques by following the edge labeled with next vertex list until a leaf in the trie is reached, and the query vertex label sequence is exhausted. For approximate tree retrieval, we use the error-tolerant finite-state recognition algorithm [5]. An adaptation of this algorithm to this problem will be briefly summarized here.

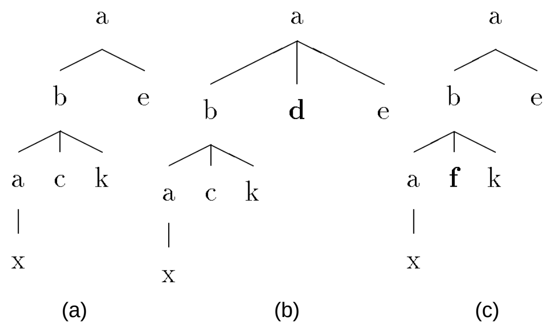

Fig. 2. Structural and leaf label differences between trees.

Let $Z = Z_1, Z_2, \ldots, Z_p$, denote a generic vertex list sequence of p vertex lists. $Z[j]$ denotes the initial subsequence of Z up to and including the jth vertex list. We will use X (of length m) to denote the query vertex list sequence, and Y (of length j) to denote the sequence that is a (possibly partial) candidate vertex list sequence (from the database of trees). Given two vertex list sequences X and Y, the distance, $dist(X[m], Y[n])$, computed according to the recurrence below, gives the minimum cost of leaf insertions, deletions or leaf label changes necessary to change one tree to the other.[3]

$$
\begin{aligned}
dist(X[i], Y[j]) &= dist(X[i-1], Y[j-1]) \\
&\qquad \text{(if } X_i = Y_j \text{ i.e., last vertex lists are same)} \\
&= \min(dist(X[i-1], Y[j-1]) + C, \\
&\qquad dist(X[i-1], Y[j]) + S, \\
&\qquad dist(X[i], Y[j-1]) + S) \\
&\qquad \text{(if } X_i \text{ and } Y_j \text{ differ only at the leaf label)} \\
&= \min(dist(X[i-1], Y[j]), \qquad \text{(Otherwise)} \\
&\qquad dist(X[i], Y[j-1])) + S
\end{aligned}
$$

Boundary Conditions

$$dist(X[0], Y[j]) = j \cdot S$$

$$dist(X[i], Y[0]) = i \cdot S$$

For the kinds of applications we have in mind, inserting a leaf to a tree may involve insertion of other internal nodes that are demanded, for instance, from the linguistic representation employed. Similarly deletion of a leaf may involve deletion of internal nodes. Thus whenever we mention insertion or deletion of a leaf, we are implicitly assuming that other internal nodes may be involved as necessary. For instance, in Fig. 4 the additional leaves in the second tree involve the addition of internal nodes.

For a tree database D and a distance threshold $t > 0$, we consider a query tree represented by a vertex list sequence $X[m]$ (not in the database) to match the database with an error of t, if the set $\{Y[n] \mid Y[n] \in D \text{ and } dist(X[m], Y[n]) \le t\}$ is not empty.

3. Note that a given cost may be realized by different combinations of these differences.

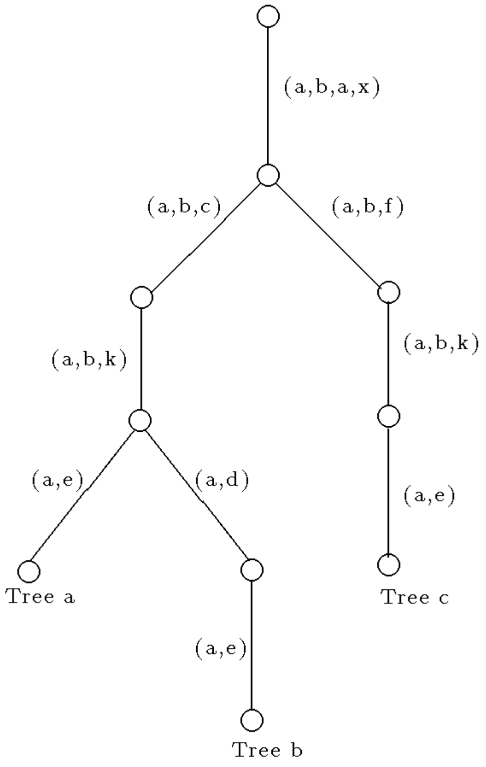

Fig. 3. Trie representation of the three trees in Fig. 2.

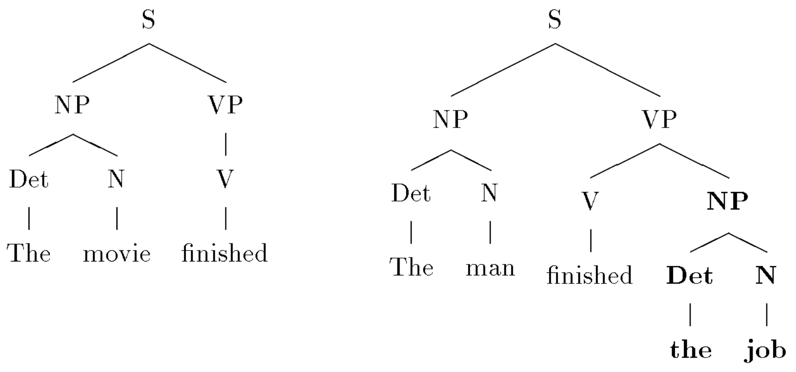

Fig. 4. Leaf insertions involving the insertions of internal nodes.

2.5 An Algorithm for Approximate Tree Retrieval

Standard searching with a trie corresponds to traversing a path starting from the start node (of the trie), to one of the leaf nodes (of the trie), so that the concatenation of the labels on the arcs along this path matches the input vertex list sequence. For error-tolerant searching, one needs to find *all* paths from the start node to one of the final nodes, so that when the labels (vertex lists) on the trie edges along a path are concatenated, the resulting vertex list sequence is within a given distance threshold *t*, of the query vertex list sequence.

This search has to be fast if approximate retrieval is to be of any practical use. This means that paths in the trie that can lead to no solutions have to be pruned early so that the search can be limited to a very small percentage of the search space. We need to make sure that any candidate vertex list sequence that is generated during the search does not deviate from certain initial subsequences of the query sequence by more than the allowed threshold. To detect such cases, we use the notion of a *cutoff distance*. The cutoff distance measures the minimum distance between an initial subsequence of the query sequence, and the (possibly partial) candidate sequence. Let *Y* be a candidate sequence whose length is *j*, and let *X* be the query sequence of length *m*. Let $l = \max(1, j - \lfloor t/S \rfloor)$ and $u = \min(m, j + \lceil t/S \rceil)$ where *S* is the cost of an insertion or deletion. The cutoff distance $cutdist(X[m], Y[j])$ is defined as

$$cutdist\big(X[m], Y[j]\big) = \min_{l \le i \le u} dist\big(X[i], Y[j]\big).$$

Note that except at the boundaries, the initial subsequences of the query sequence *X* considered are of length $j - \lfloor t/S \rfloor$ to length $j + \lceil t/S \rceil$. Any initial subsequence of *X* shorter than *l* needs more than $\lfloor t/S \rfloor$ leaf insertions, and any initial substring of *X* longer than *u* requires more than $\lceil t/S \rceil$ leaf deletions, to at least equal *Y* in length, violating the error threshold constraint.

Given a query vertex list sequence *X*, a partial candidate sequence *Y* is generated by successively concatenating labels along the arcs as transitions are made, starting with the start node. Whenever we extend *Y* going down the trie, we check if the cutoff distance of *X* and the partial *Y* is within the bound specified by the threshold *t*. If the cutoff distance goes beyond the threshold, the last edge is backed off (recursively if necessary) to the source node (in parallel with the shortening of *Y*) and some other edge is tried. If, during the construction of *Y*, a terminal node (which may or may not be a leaf of the trie) is reached without violating the cutoff distance constraint, *and* $dist(X[m], Y[j]) \le t$ at that point, then the vertex list sequence *Y* (of length *j*) corresponds to a tree in the database that matches the input query sequence approximately.[4]

4. Note that we have to do this check, since we may come to other irrelevant terminal nodes during the search.

Denoting the nodes of the trie by subscripted *q*s (q_0 being the initial node (e.g., top node in Fig. 3)) and the labels of the edges by *V*, and denoting by $\delta(q_i, V)$ the node in the trie that one can reach from node q_i with edge label *V* (denoting a vertex list), we present, in Fig. 5, the algorithm for generating all *Y*s that match the query tree with an error threshold *t* by a depth-first probing of the trie. The crucial point in this algorithm is that the cutoff distance computation can be performed very efficiently by maintaining an *m* by *j* matrix *H* (*j* varying suitably as *Y* changes), with element $H(i, j) = dist(X[i], Y[j])$ [1]. We can note that the computation of any element $H(i, j)$ recursively depends on only $H(i-1, j-1)$, $H(i, j-1)$, $H(i-1, j)$ from the earlier definition of the distance. During the depth first search of the trie, entries in column *j* of the matrix *H* have to be (re)computed, only when the candidate sequence *Y* is of length *j*. During backtracking, the entries for the last column are discarded, but entries in prior columns are still valid. Thus, all entries required by $H(i, j)$, except $H(i-1, j)$, are already available in column $j-1$. The computation of $cutdist(X[m], Y[j])$ involves a loop in which the minimum is computed. This loop (indexing over increasing *j*) computes $H(i-1, j)$ before it is needed for the computation of $H(i, j)$.

Let us assume that the tree database consists of *K* trees each with about *L* leaves, compressed into a trie so that the trie has *K* leaves, and the root and internal trie nodes have a branching factor of $\sqrt[L]{K}$ successors each. Let us further assume that we search for a tree already in the database, but with a threshold *t* which allows for $\lceil t/S \rceil$ insertions or deletions. In the worst case we will go down $L+1$ nodes of the trie (the edges between those nodes corresponding to the vertex lists for the query tree). In all but the last of these nodes, we will visit at most $\left(\sqrt[L]{K}\right)^{\lceil t/S \rceil}$ additional nodes for

```
/*push empty candidate, and start
 node to start search */
$push((\epsilon, q_0))$
while stack not empty
begin
  $pop((Y', q_a))$ /* pop partial sequence $Y'$
           and the node */
  for all $q_b$ and $V$ such that $\delta(q_a, V) = q_b$
    begin /* extend the candidate sequence */
      $Y = concat(Y', V)$
        /* $j$ is the current length of $Y$ */
    /* check if Y has deviated too much,
      if not push */
      if $cutdist(X[m], Y[j]) \le t$ then $push((Y, q_b))$
    /* also see if we are at a final state */
      if $dist(X[m], Y[j]) \le t$ and
        $q_b$ is a terminal node then output $Y$
    end
end
```

Fig. 5. Algorithm for error-tolerant recognition of vertex list sequences.

searching the neighborhood of a node, and spend $O(L \cdot \log L)$ time at each such visited node for the cutoff distance computation, the $\log L$ factor coming from the length of a vertex list sequence. Thus the total time for the search will be $O(L^2 \cdot \log L \cdot (\sqrt[L]{K})^{\lceil t/S \rceil})$. As expected this is exponential in the threshold t, as one has to consider all possibilities delineated by the threshold, but since the branching factor is usually small, the impact of this factor is not substantial. The time for preprocessing the tree database into into a trie is almost linear in the size of the database.

3 Experimental Results

We have experimented with three synthetically generated database of trees with the properties given in Table 1, and a database of syntactic trees constructed from Wall Street Journal Treebank, available from Linguistic Data Consortium CD. In the table for the synthetic data, the third column (label ALP) gives the average ratio of the vertices at each level which are randomly selected as leaf vertices in a tree. The fourth column gives the maximum number of children that a nonleaf node may have. The last column gives the maximum depth of the trees in that database.

TABLE 1
PROPERTIES OF THE SYNTHETIC DATABASES OF TREES

Database	Number of Trees	ALP	Max Children	Max Depth
1	1,000	1/3	8	5
2	10,000	1/2	16	5
3	50,000	1/2	8	4

From these synthetic databases, we randomly extracted 100 trees and then perturbed them with random leaf deletions, insertions and label changes so that they were of some distance from a tree in the original tree. We used thresholds $t = 2$ and $t = 4$, allowing an error of $C = 1$ for each leaf label change and an error of $S = 2$ for each insertion or deletion. We then ran our algorithm on these data sets and obtained performance information. All runs were performed on a Sun UltraSparc 140 with 128M real memory. The results are presented in Table 2.

TABLE 2
PERFORMANCE RESULTS FOR THE APPROXIMATE TREE MATCHING ALGORITHM

Database	Threshold	Avg. Leaves/ Query Tree	Avg. Search Time (Msec)	Avg. Trees Found/ Query
1	2	12.00	29	1.96
	4	12.42	39	16.65
2	2	24.65	473	3.32
	4	25.62	624	31.59
3	2	13.41	1145	14.21
	4	13.21	1810	67.43

For a second set of experiments we constructed a database of trees from the Wall Street Journal Treebank available on the Linguistic Data Consortium Penn Treebank CD. The trees in this database are however not in a format directly usable by the algorithm above. Particularly, various noun phrases and verb phrases are represented linearly without encoding any lower level structural relationships. For instance, there are quite a number of noun phrases like `(NP the auto maker)` represented in a flat manner. We converted the trees in the treebank so that such cases were also made hierarchical. Furthermore, for a given sentence tree in the treebank, that tree, and all of its subtrees (with the exception of leaves, and subtrees which contain a leaf and a root node denoting the category of the leaf lexical item) have also been entered as separate trees, each rooted by a node covering a phrase or a sentential form. This is likely to be useful in an example-based machine translation application, where, if the match for a whole sentence tree fails, one may check for matching subtrees to cover the given tree in a maximal way.

The resulting treebank database consisted of a total of 18,104 trees with an average of 14.9 nodes per tree, and average of 8.6 leaf nodes and an average depth of 4.25. We generated a query database of 100 trees which were selected from the original treebank, and perturbed so that they either had some leaf or structural differences. We limited the number of structural differences to a maximum of two leaf additions or deletions.

The results indicate that when the database was searched with threshold $t = 1$, which allowed for single leaf label difference, the average search time was 463 milliseconds. For threshold limits of $t = 2$ and $t = 4$, which allowed insertion or deletion of leaves or a combination of leaf label differences with insertions and deletions, the average search times were 642 and 1,054 milliseconds, respectively.

Another set of experiments were conducted to gauge the impact of the use of cutoff edit distance in controlling the search. The search code was modified so that the search was done without the cutoff edit distance controlling the search extent, for the three query sets described above. The resulting average search times for the same set of query trees were 2,415 milliseconds for $t = 1$; 2,090 milliseconds for $t = 2$; and 2,156 milliseconds for $t = 4$. It should be noted that the set of query trees used for these was the same used above, hence, the time results are slightly different from each other as the threshold in this case has no impact on search control except for final evaluation of any solution found.

In order to identify the time difference between exact and approximate searching, 100 trees randomly selected from the database were used as query trees with thresholds of $t = 0$ for exact match and for $t = 1$, 2, and 4. This can give us an impression of the overhead of searching the database for approximate matches. The average time for exact search (which is equivalent to standard trie searching) was 140 milliseconds. The average search times for $t =$ 1, 2, and 4 were 348, 580, and 1,074 milliseconds. So, the overhead of approximate searching is quite tolerable, but for large t (> 4), one is likely to search most of the database.

It can be seen that the approximate search algorithm is quite fast for the set of tree databases that we have experimented with, in that they can be used in applications without the approximate search component being the computational bottleneck.

From a qualitative respect the database was queried with a number trees to see what kinds of trees would be retrieved. We limited our queries to trees that we knew has close neighbors in the treebank, since the treebank was not meant to be used for an EBMT application per se. In response to our query tree (already in the database)

```
(VP (V file) (NP (DET the) (N reports)))
```

with $t = 1$, the system produced, as expected, (with errors in ()'s),

```
(VP (V file) (NP (DET the) (N reports))) (0)
(VP (V file) (NP (DET their) (N reports))) (1)
```

The query

```
(VP (V file) (NP (N transactions)))
```

with $t = 4$ produced

```
(VP (V represent) (NP (ADJ actual)
                      (N transactions))) (3)
(VP (V file) (NP (DET the) (N reports))) (3)
(VP (V file) (NP (DET their) (N reports))) (3)
(VP (V file) (NP (N reports))) (1)
(VP (V completed) (NP (DET a) (N transac-
tion))) (4)
(VP (V completed) (NP (DET his) (N testi-
mony))) (4)
```

while the query

```
(VP (V execute) (NP (N transactions)))
```

produced, with $t = 3$

```
(VP (V represent) (NP (ADJ actual) (N
transactions))) (3)
```

4 Related Work

There have been a number of prior studies that are relevant to the problem considered in this paper. Selkow [7] describes an algorithm that computes the minimum distance between two trees in terms of insertions and deletions at the leaves and label change at the root node. Tai [8] presents a very general model of edit operations on trees involving arbitrary node label changes, node deletions and insertions. These algorithms have however not been presented in a tree database retrieval context. Wu and Manber [11] describe *agrep*, an algorithm for fast approximate sequence searching, It relies on a very efficient pattern matching scheme whose steps can be implemented with arithmetic and logical operations but is very suitable when the pattern is very small and the sequence to be searched is very large. Myers and Miller [3] presents an $O(NM)$ algorithm for approximate matching to regular expressions with arbitrary costs, with M being the length of the sequence and N being the size of the regular expression. It again is suited for applications where the pattern or the regular expression is small and the sequence is large. Neither of the above treat the sequence or the pattern as representing a database, and do not exploit any structure in either.

The problem presented in this paper is in many respects similar to the problems presented in Wang et al. [10], Utsuro et al. [9], and Maruyama and Watanabe [2]. Wang et al. [10] describe tree database system for approximate tree matching, called the Approximate Tree-by-Example which is designed to support construction and querying of database of trees. They provide a special query language to describe constraints on the set of trees requested. Although this system allows for a more general set of differences between two trees, the examples they suggest for syntactic tree matching can trivially be handled by our approach, without the overhead of the more general cases. They have provided no performance evaluation of their system.

Utsuro et al. [9] describe an approach to retrieving matching surface case structures (represented as feature-value structures) from a database of such structures. The similarity of case frames are judged on two grounds: the syntactic features of the two structures (case markers in this case) should match maximally, and the semantic similarity of the lexical items filling the case roles, as defined by a suitable semantic ontology or thesaurus, should be close. They present a favorable performance evaluation of their system, but the number of examples used is very small. Our approach can also be used in this context. Since feature structures can be represented as trees with leaves corresponding to morphosyntactic features, the structural similarity can be trivially dealt with. The semantic similarity can be handled by casting it as a leaf label difference, but instead of considering leaf label differences strictly, one can consider the semantic distance obtained from the thesaurus.

Maruyama and Watanabe [2] describe a system which tries to cover a linguistic structure represented as a tree, by finding matching covers (of various subtrees) from a database of example trees, and selecting a set of matching subtrees with the lowest cost of covering. The cover algorithm is based on the dynamic programming paradigm. They have applied their system to a dependency structure representation of Japanese sentences.

5 Conclusions

This paper has presented an approach to retrieving from a database, trees that match a given query tree approximately. The proposed approach can be used as tree searching engine in example-based machine translation applications, or retrieval from lexical database represented as feature-value structures. The algorithm efficiently searches in a database of trees, all trees that are "close" to a given query tree. It has been implemented on Sun workstations, and experiments on rather large synthetic tree databases and actual syntactic tree database from the Penn Treebank, indicate that it can perform approximate matches within tenths of a second to about a second depending on the size of the database and the error that the search is allowed to consider.

Acknowledgments

Comments by three anonymous reviewers helped tremendously in improving the content and presentation of this paper. This research was in part funded by a NATO Science for Stability Phase III Project Grant—TU-LANGUAGE.

References

[1] M.W. Du and S.C. Chang, "A Model and a Fast Algorithm for Multiple Errors Spelling Correction," *Acta Informatica*, vol. 29, pp. 281-302, 1992.

[2] H. Maruyama and H. Watanabe, "Tree Cover Search Algorithm for Example-Based Translation," *Proc. Fourth Int'l Conf. Theoretical and Methodologies Issues in Machine Translation*, pp. 173-184, 1992.

[3] E.W. Myers and W. Miller, "Approximate Matching of Regular Expressions," *Bulletin of Mathematical Biology*, vol. 51, no. 1, pp. 5-37, 1989.

[4] S. Nirenburg, S. Beale, and C. Domashnev, "A Full-Text Experiment in Example-Based Translation," *Proc. Int'l Conf. New Methods in Language Processing*, Manchester, UK, pp. 78-87, 1994.

[5] K. Oflazer, "Error-Tolerant Finite-State Recognition With Applications to Morphological Analysis and Spelling Correction," *Computational Linguistics*, vol. 22, no. 1, pp. 73-89, 1996.

[6] S. Sato and M. Nagao, "Towards Memory-Based Translation," *Proc. 13th Int'l Conf. Computational Linguistics*, vol. 3, pp. 247-252, 1990.

[7] S.M. Selkow, "The Tree-to-Tree Editing Problem," *Information Processing Letters*, vol. 6, no. 6, pp. 184-186, 1977.

[8] K.C. Tai, "The Tree-to-Tree Correction Problem," *J. ACM*, vol. 26, no. 3, pp. 422-433, 1979.

[9] T. Utsuro, K. Uchimoto, M. Matsumoto, and M. Nagao, "Thesaurus-Based Efficient Example Retrieval by Generating Retrieval Queries From Similarities," *Proc. 15th Int'l Conf. Computational Linguistics*, vol. 2, pp. 1,044-1,048, 1994.

[10] J.T.-L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A System for Approximate Tree Matching," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 4, pp. 559-570, 1994.

[11] S. Wu and U. Manber, "Fast Text Searching With Errors," Technical Report TR91–11, Dept. of Computer Science, Univ. of Arizona, 1991.