TREE MATCHING FOR EVALUATION OF SPEECH INTERPRETATION SYSTEMS

Matthias Thomae, Tibor Fabian, Robert Lieb, Günther Ruske

Institute for Human-Machine Communication
Technische Universität München, Germany
{tho,fab,lie,rus}@mmk.ei.tum.de

ABSTRACT

Common data-driven evaluation metrics for speech understanding systems are based on automatically comparing sequences of slot–value pairs by dynamic programming (DP) matching. However, for complex hierarchical language models sequence matching based metrics don't seem appropriate as they cannot fully capture structural similarities. For this task we propose a novel evaluation metric, the tree node accuracy. Our approach is founded on a DP-style algorithm that computes the minimum edit distance between pairs of ordered labeled trees and includes the sequence matching problem as a special case. We also extended the basic scheme for our task to support trees consisting of different categories of tree nodes. Experiments carried out on several semantic models confirm that the tree matching based approach displays greater flexibility than conventional sequence matching based metrics and is especially suited for complex hierarchical models.

1. INTRODUCTION

The quantitative and objective evaluation of the 'goodness' of speech interpretation systems, which produce some representation of the meaning of the user's utterances, has several goals: Firstly, although speech interpreters are normally not used in isolation but as a component of a larger system such as a spoken language dialogue system, a *glass box* assessment [1], i.e. measuring the performance of single components, plays an important role in determining the goodness of the whole system. Secondly, the ability to measure a component's performance is often a vital tool in order to purposefully improve the component during its development. A further goal is to enable the comparison of different approaches. However, meaning representations are not 'standardized' and appear with different complexities. Therefore, an evaluation metric for speech interpretation systems must operate on a common basis that other meaning representations can be converted to in order to be comparable.

In [2] we introduced our *One-stage Decoder for Interpretation of Natural Speech* (ODINS), which directly computes a tree-structured semantic representation from an input speech signal. ODINS operates on a generalized uniform knowledge model consisting of a hierarchy of probabilistic transition networks. In order to support the development of such a knowledge model for an airport information system application, the need for a suitable automatic evaluation procedure arose. A commonly used evaluation metric for speech understanding systems is the *concept accuracy*, which was derived from the *word accuracy* metric used for speech recognition systems evaluation [3]. The concept accuracy computation is based on matching sequences of so-called 'concepts' in the shape of slot–value pairs [3, 4, 1] or slot–value–communicative function triples [5, 6]. The same *minimum edit distance* based matching algorithm as for speech recognizer evaluation can be utilized to find the optimum match between reference and hypothesis concept sequences of a test corpus. The concept accuracy is then computed from the counts of correctly and incorrectly matched concepts. However, we think that in order to fully take the hierarchical nature of ODINS' output trees into account, the matching should happen directly on the tree structure. Therefore, our goal was to devise an evaluation method that measures to what degree pairs of reference and hypothesis trees agree with each other.

As ODINS is able to process hierarchical models with an unlimited number of hierarchy levels and also supports the skipping of levels, the structure of its output trees is very variable. Thus, we aimed to impose as little constraints on the trees to be evaluated as possible. The two most fundamental properties of the trees are that they are ordered (due to the sequential nature of the speech signal) and consist of labeled nodes (the labels being e.g. semantic categories or words). For matching this kind of trees, a number of algorithms have been proposed (see [7] for an overview) and applied to problems such as RNA secondary structure analysis. For our task, we selected the algorithm by Shasha and Zhang [7] because it is general, easy to understand and to implement and has limited time and space complexity.

We utilize the tree matching algorithm to compute the best match between reference and hypothesis trees, and then express the accuracy of the match in a novel evaluation metric which we call *tree node accuracy*. Moreover, we extended the basic tree matching approach to support the matching of *typed* tree nodes. Hence, the novel evaluation metric is generally applicable to all speech understanding systems whose output is or can be converted to an ordered, labeled and possibly typed tree structure. The tree node accuracy, which was already briefly introduced in [2], can be seen as a natural generalization of the word accuracy and the concept accuracy. In fact, the problem of tree pattern matching can be seen as a generalization of sequence pattern matching, as it contains the sequential problem as a special case. Moreover, the tree matching algorithm we will discuss in this paper is based on the dynamic programming algorithm which is usually used for sequence matching [8]. Therefore, we will review the sequence matching problem in Section 3. In Section 4, we will derive the basic tree matching algorithm, drawing analogies to the sequence matching case. Our extension of the basic approach to typed tree nodes will be presented in Section 5. In Section 6, reasons will be

This work was funded partly by the German Research Council (DFG) project Ru 301/6-1 and also by the NADIA research project from the Bayerische Motorenwerke (BMW) group.

From: ASRU 2003, Aka Verlag, Berlin, IOS Press, Amsterdam

i	1	2		3	4	5	6
$S_1[i]$	d_i	drei		sieben	drei	von	hamburg
		↓		↓	↓	↓	↓
$S_2[j]$		drei	zwei	sieben	drei	nach	hamburg
j		1	2	3	4	5	6

Fig. 1. *Sequence matching example.*

given why the tree accuracy might be a more appropriate evaluation metric than the concept accuracy. Some experimental results which support this assumption are presented in Section 7. In the following section, we will define the novel evaluation metric.

2. DEFINITION OF TREE NODE ACCURACY

A goal of speech recognizer evaluation is to quantify how well the recognizer's hypotheses of a set of test utterances correspond to the correct (i.e. hand-labeled) reference transcriptions of the test utterances. In order to reach this goal the (optimum) correspondence between pairs of hypothesis and reference utterances must be known. The correspondence function gives rise to three types of errors, namely substitutions, insertions and deletions of words. The *word accuracy* Acc_w is then computed from the counts of correct C_w, substituted S_w, inserted I_w and deleted words D_w:

$$Acc_w = \frac{N_w - S_w - I_w - D_w}{N_w} = \frac{C_w - I_w}{C_w + S_w + D_w} \quad (1)$$

where $N_w = C_w + S_w + D_w$ is the number of words in the reference [9].

Our goal was to devise an evaluation method that measures to what degree pairs of hypothesis trees, produced by a speech interpretation system, and reference trees agree with each other. Assuming that we know the optimum correspondences between pairs of trees, the number of correct C_n, substituted S_n, inserted I_n and deleted D_n *tree nodes* can be counted. Consequently we define the *tree node accuracy* Acc_n analogously to Eq. (1) as:

$$Acc_n = \frac{C_n - I_n}{C_n + S_n + D_n} \quad (2)$$

3. SEQUENCE MATCHING REVISITED

Let's consider an example. Fig. 1 depicts a possible correspondence function between a reference word sequence $S_1 =$ [*d_i, drei, sieben, drei, von, hamburg*] and a hypothesis $S_2 =$ [*drei, zwei, sieben, drei, nach, hamburg*]. Intuitively, the correspondence function can be viewed as a transformation from S_1 to S_2 that consists of three basic *edit operations*: Delete operations, e.g. $(1 \mapsto \varepsilon)$, insert operations, e.g. $(\varepsilon \mapsto 2)$, and map operations, e.g. $(3 \mapsto 3)$ or $(5 \mapsto 5)$. In general, an edit operation is denoted $(i \mapsto j)$, where i and j are indices $1 \leq i \leq |S_1|$ or $1 \leq j \leq |S_2|$ from the symbol sequences $S_1 = [S_1[1], S_1[2], .., S_1[|S_1|]]$ or $S_2 = [S_2[1], S_2[2], .., S_2[|S_2|]$, respectively, or the null index ε. The edit operation $(i \mapsto j)$ is called a *map operation* if $i \neq \varepsilon$ and $j \neq \varepsilon$, a *delete operation* if $i \neq \varepsilon$ and $j = \varepsilon$ and an *insert operation* if $i = \varepsilon$ and $j \neq \varepsilon$. A map operation $(i \mapsto j)$ is termed *correct* if the symbols are identical ($S_1[i] = S_2[j]$), or a *substitution* otherwise ($S_1[i] \neq S_2[j]$). Formally, the ith symbol of a sequence S of length $|S|$ is denoted $S[i]$. A sub-sequence of S that contains consecutive elements from index i' to index i inclusively is denoted $S[i'..i]$. Moreover, $S[i..i] = S[i]$ and $S[i'..i] = \emptyset$ if $i < i'$, where $\emptyset$ denotes an empty sequence.

By associating each edit operation with a cost, we can quantitatively characterize a transformation: The total cost or *edit distance* of the transformation is the sum of the costs of each edit operation involved. Now, the goal of finding the best transformation can be reformulated as to find the transformation with the minimum edit distance. We set the edit costs throughout the paper according to the widely used NIST scoring software [10] to 4 for substitutions, 3 for insertions and deletions and 0 for correct mappings.

A *mapping* M between two sequences S_1 and S_2 is defined as an unordered set of edit operations:

$$M(S_1, S_2) = \{(i_1 \mapsto j_1), (i_2 \mapsto j_2), .., (i_{|M|} \mapsto j_{|M|})\} \quad (3)$$

For the example of Fig. 1 the mapping $M(S_1, S_2) = \{(1, \varepsilon), (2, 1), (\varepsilon, 2), (3, 3), (4, 4), (5, 5), (6, 6)\}$.

Any pair of map operations $(i_1 \mapsto j_1)$ and $(i_2 \mapsto j_2)$ with $1 \leq i_1, i_2 \leq |S_1|$ and $1 \leq j_1, j_2 \leq |S_2|$ of a mapping $M(S_1, S_2)$ has the following properties:

$$i_1 = i_2 \iff j_1 = j_2 \quad \text{(one-to-one mapping)} \quad (4)$$

$$i_1 < i_2 \iff j_1 < j_2 \quad \text{(symbol order preserved)} \quad (5)$$

The *cost* of a single edit operation $(i \mapsto j)$ is denoted $c(i \mapsto j)$ and usually depends only on the symbol identities, so that $c(i \mapsto j) = c(S_1[i] \mapsto S_2[j])$. In order to be a distance metric, c has to meet the following conditions:

$$c(a \mapsto b) \geq 0 \quad \text{(non-negative cost)} \quad (6)$$

$$c(a \mapsto a) = 0 \quad \text{(no cost for identity mapping)} \quad (7)$$

$$c(a \mapsto b) = c(b \mapsto a) \quad \text{(order independent)} \quad (8)$$

$$c(a \mapsto c) \leq c(a \mapsto b) + c(b \mapsto c) \quad \text{(triangle inequality)} \quad (9)$$

The *edit distance* $D_e(M(S_1, S_2))$ of a mapping $M(S_1, S_2)$ that transforms S_1 into S_2 is computed by accumulating the costs of the single edit operations:

$$D_e(M(S_1, S_2)) = \sum_{(i,j) \in M(S_1, S_2)} c(i \mapsto j) \quad (10)$$

Now we can define the *minimum edit distance* $D_e^{min}(S_1, S_2)$ as the edit distance of the mapping that yields the minimum value:

$$D_e^{min}(S_1, S_2) = \min_{M(S_1, S_2)} D_e(M(S_1, S_2)) \quad (11)$$

3.1. Recursive formulation, dynamic programming algorithm

The task of computing Eq. (11) for given sequences S_1 and S_2 and a given cost function c is usually performed by a dynamic programming (DP) algorithm [8]. With this algorithm, the minimum edit distance is computed by accumulating the single costs c step-by-step in a left-to-right manner. Locally, i.e. in each step, decisions are taken to yield the least-cost path. The heart of the algorithm is captured in a recursive formula that computes the (minimum) accumulated cost for transforming the sub-sequence $S_1[1..i]$ into $S_2[1..j]$ from previously computed values:

$$D_e^{min}(S_1[1..i], S_2[1..j]) = \min \begin{cases} D_e^{min}(S_1[1..i-1], S_2[1..j]) & + c(i \mapsto \varepsilon) \\ D_e^{min}(S_1[1..i], S_2[1..j-1]) & + c(\varepsilon \mapsto j) \\ D_e^{min}(S_1[1..i-1], S_2[1..j-1]) & + c(i \mapsto j) \end{cases} \quad (12)$$

j ↓	i →	0	1 d_i	2 drei	3 sieben	4 drei	5 von	6 hamburg
0		**0**	**3**	6	9	12	15	18
1	drei	3	4	**3**	6	9	12	15
2	zwei	6	7	**6**	7	10	13	16
3	sieben	9	10	9	**6**	9	12	15
4	drei	12	13	10	9	**6**	9	12
5	nach	15	16	13	12	9	**10**	13
6	hamburg	18	19	16	15	12	13	**10**

Fig. 2. *Dynamic programming example.*

Fig. 2 illustrates the progression of the DP algorithm for the minimum edit distance computation between the word sequences of Fig. 1. The minimum edit distances $D_e^{min}(S_1[1..i], S_2[1..j])$ are depicted as cells of a table whose columns and rows correspond to the indices of S_1 and S_2, respectively. The table is processed top-to-bottom and left-to-right. According to Eq. (12), the value of a cell (i, j) is computed from the cells to the left $(i-1, j)$, to the top $(i, j-1)$, and to the top-left $(i-1, j-1)$. Transitions from the left are computed with the first term of Eq. (12) and correspond to delete operations; ones from the top correspond to the second term and to insertions; ones from the top-left correspond to the third term and to correct matches if $S_1[i] = S_2[j]$ or to substitutions otherwise. The best transition corresponds to the minimum of the three values and is depicted by an arrow. The costs for deletions, insertions, substitutions and correct matches were set to 3, 3, 4 and 0, respectively, thus fulfilling Conditions (6) – (9).

Generally, more than one transition can result in the same optimum value, e.g. at cell $(1, 2)$ both the transitions from the top and from the top-left yield the minimal cost. Hence, a pair of sequences can have more than one minimum edit distance mapping. The example of Fig. 2 has only one optimum mapping with an edit distance of 10 (the value of the cell $(|S_1|, |S_2|)$). The optimum mappings are retrieved by back-tracking the transitions from cell $(|S_1|, |S_2|)$ to cell $(0, 0)$, depicted in Fig. 2 with bold-faced numbers. In order to allow delete and insert operations at the beginning of the sequences, the table contains an extra column and row corresponding to empty sequences $\emptyset$. The table is initialized by setting the value of cell $(0, 0)$ to $D_e^{min}(\emptyset, \emptyset) = 0$.

4. TREE MATCHING REVISITED

Similar to the case of sequences, the problem of finding the optimum match between a pair of ordered, labeled trees T_1 and T_2 can be tackled by determining the least-cost transformation from T_1 to T_2 utilizing three basic tree editing operations, illustrated in Fig. 3. The map operation converts a tree node i into j; a deletion removes a node i and makes its children the children of its parent k; inserting a node j makes it the child of a certain node k and a consecutive number of k's children become the children of j. Similar to the sequence case, introducing a cost function for the operations enables the definition of a minimum edit distance criterion.

Formally, a labeled ordered rooted tree T consists of $|T|$ labeled tree nodes, exactly one of them being the root node. A labeled tree node of T is referred to via its index i; its label is denoted $t[i]$. Similar to [7], we assume in the following that the index corresponds to the left-to-right postorder numbering of T, which can be computed by performing a left-right depth search through the tree, assigning each node a consecutive number after all of its children have been visited. Hence, $t[1]$ is the leftmost leaf node and $t[|T|]$ is the root node of T. Fig. 4 illustrates two example trees including node labels and postorder numbers.

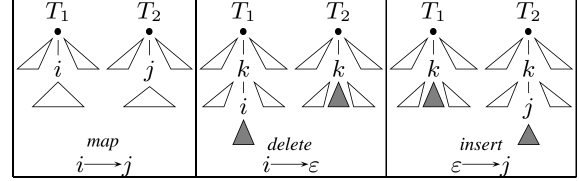

Fig. 3. *Basic edit operations on tree nodes; a triangle depicts a subtree of the node above it.*

Similar to mappings between sequences (see Section 3), we define a mapping M between two labeled, ordered trees T_1 and T_2 as an unordered set of tree edit operations:

$$M(T_1, T_2) = \{(i_1 \mapsto j_1), (i_2 \mapsto j_2), .., (i_{|M|} \mapsto j_{|M|})\} \quad (13)$$

The indices $i_1, i_2, .., i_{|M|}$ and $j_1, j_2, .., j_{|M|}$ are either tree node indices in the ranges $1..|T_1|$ or $1..|T_2|$, respectively, or the null index ε. The tree edit operation $(i \mapsto j)$ is called a *map operation* if $i \neq \varepsilon$ and $j \neq \varepsilon$, a *delete operation* if $i \neq \varepsilon$ and $j = \varepsilon$ and an *insert operation* if $i = \varepsilon$ and $j \neq \varepsilon$. Fig. 4 illustrates a mapping between two trees T_1 and T_2 by depicting the map operations with gray arrows. For this example, the mapping is:

$$\begin{aligned} M(T_1, T_2) = \{&(1, \varepsilon), (2, \varepsilon), (3, 1), (4, 2), (\varepsilon, 3), (\varepsilon, 4), (5, 5), \\ &(6, 6), (7, 7), (8, 8), (9, 9), (10, \varepsilon), (11, 10), \\ &(12, 11), (13, 12), (14, 13), (15, 14)\} \end{aligned}$$

Any pair of map operations $(i_1 \mapsto j_1)$ and $(i_2 \mapsto j_2)$ with $1 \leq i_1, i_2 \leq |T_1|$ and $1 \leq j_1, j_2 \leq |T_2|$ of a mapping $M(T_1, T_2)$ must meet the following conditions:

$$i_1 = i_2 \iff j_1 = j_2 \quad \text{(one-to-one mapping)} \quad (14)$$

$$i_1 \text{ left sibling of } i_2 \iff j_1 \text{ left sibling of } j_2 \quad \text{(sibling order preserved)} \quad (15)$$

$$i_1 \text{ ancestor of } i_2 \iff j_1 \text{ ancestor of } j_2 \quad \text{(ancestor order preserved)} \quad (16)$$

Similar to the mapping Conditions (4) and (5) of the sequence case, Condition (14) states that a node may not occur in more than one map operation, and Condition (15) ensures that the transformation preserves the left-to-right order. Additionally, Condition (16) retains the top-to-bottom *structure*. This means e.g. for Fig. 4 that if $t_1[14] =$ *AOrigin* is mapped to $t_2[13] =$ *ADestination*, a descendant of *AOrigin* such as *von* may only be mapped to a node 'below' *ADestination*. More precisely, we mean 'below' $t_2[13]$ in terms of structure (i.e. the descendants $t_2[10]$, $t_2[11]$, and $t_2[12]$) and not 'below' in terms of node depth (that would allow e.g. *von* to be mapped to $t_2[7] =$ *drei*).

Similar to the sequence case, we define a *cost function* $c(i \mapsto j)$ for a single tree edit operation $(i \mapsto j)$. As the cost function should be a distance metric, we apply Conditions (6) – (9). The *tree edit distance* $D_t(M(T_1, T_2))$ of the mapping $M(T_1, T_2)$

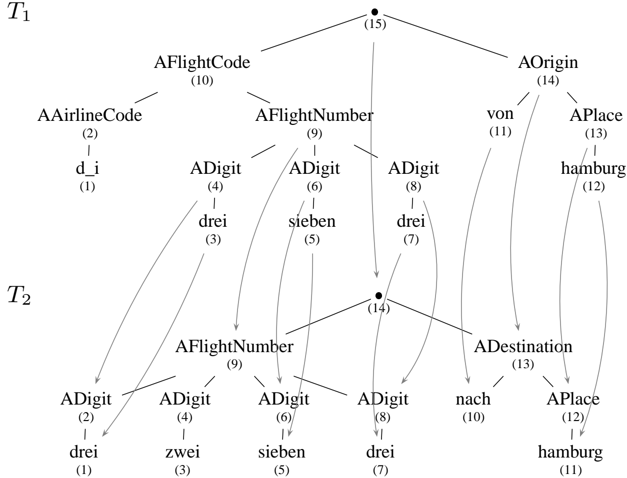

Fig. 4. *Mapping between labeled, ordered, rooted trees T_1 and T_2 with left-to-right postorder numbered nodes.*

that transforms T_1 into T_2 and the *minimum tree edit distance* $D_t^{min}(T_1, T_2)$ are thus expressed by:

$$D_t(M(T_1, T_2)) = \sum_{(i,j) \in M(T_1,T_2)} c(i \mapsto j) \tag{17}$$

$$D_t^{min}(T_1, T_2) = \min_{M(T_1,T_2)} D_t(M(T_1, T_2)) \tag{18}$$

4.1. Recursive formulation

As in Section 3.1, we want to utilize a dynamic programming algorithm to compute the minimum edit distance. Hence we need a recursive formula to compute in a step-by-step, left-to-right manner the accumulated costs for transforming 'sub-structures' from T_1 to 'sub-structures' from T_2.

Similar to [7], we use the postorder numbering to define the order of a tree T. A 'sub-structure' of a tree is thus composed of consecutively numbered tree nodes. Generally, such a structure does not consist of a single subtree of T but of several subtrees, called an *ordered subforest* of T. The subforest is ordered in the sense that its subtrees appear in the same order as they do in T. Formally, $T[i'..i]$ denotes the subforest consisting of the subtrees of T that contain the nodes with indices i' to i, inclusively. If $i' > i$, $T[i'..i] = \emptyset$. Fig. 5 illustrates an example for a subforest of tree T_1 of Fig. 4 that contains nodes 6..13 of T_1.

The computation of forest distances is thus required as an intermediate step for the computation of the tree distance. Our desired formula should determine how to compute the minimum distance between subforests $T_1[i'..i]$ and $T_2[j'..j]$ of the trees T_1 and T_2. We denote this as *minimum forest distance* $D_f^{min}(T_1[i'..i], T_2[j'..j])$ or shorter $D_f^{min}(i'..i, j'..j)$. The desired recursive formula should similar to the sequence case determine the (minimum) forest distance from previously computed distances and the costs for the three tree edit operations.

For delete and insert operations, the computation is equivalent to the sequence case. The new forest distance is composed of the old value and the cost for deletion or insertion, respectively. For map operations, however, special care must be taken to ensure that the formulation obeys the top-to-bottom structure of the trees (see

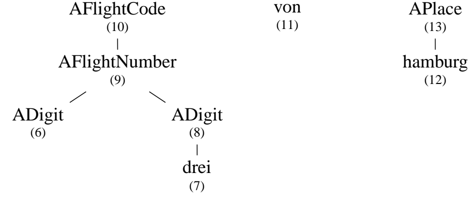

Fig. 5. *Ordered subforest $T_1[6..13]$ of tree T_1 of Fig. 4.*

Section 4). Note that the left-to-right ordering is automatically preserved by the dynamic programming procedure. Thus we need to decompose the forest(s) into two parts:

1. The subtree rooted at the current tree node i (or j)
2. All the other trees of the forest, located left of the current tree node

In order to perform this decomposition, the postorder number of the node at the 'boundary' needs to be known. Let $T[i]$ denote the subtree of T rooted at i. The index of the *leftmost leaf descendant* of $T[i]$ is denoted $l(i)$. For leaf nodes, $l(i)$ equals i. The expression for $T[i]$ in terms of forests is $T[l(i)..i]$. Thus, the part of the forest $T[i'..i]$ located to the left of i can be expressed as $T[i'..l(i) - 1]$ if $i' < l(i)$.

Since our ultimate goal is the computation of tree distances, we restrict the value of the left bound i' to be a leftmost leaf descendant $l(i_1)$ of some tree node i_1. Obviously, the right bound i of the forest $T[l(i_1)..i]$ must then be a value from the set of descendants $desc(i_1)$ of i_1, i.e. part of the subtree rooted at $T[i_1]$.

Under the assumption that $i \in desc(i_1)$ and $j \in desc(j_1)$ we can formulate the recursive forest distance computation similar to [7] as:

$$D_f^{min}(l(i_1)..i, l(j_1)..j) = \min \begin{cases} D_f^{min}(l(i_1)..i-1, l(j_1)..j) & + c(i \mapsto \varepsilon) \\ D_f^{min}(l(i_1)..i, l(j_1)..j-1) & + c(\varepsilon \mapsto j) \\ D_f^{min}(l(i_1)..l(i)-1, l(j_1)..l(j)-1) & \\ \quad + D_f^{min}(l(i)..i-1, l(j)..j-1) & + c(i \mapsto j) \end{cases} \tag{19}$$

The first two terms of Eq. (19) correspond to delete and insert operations, respectively. The third term corresponds to a map operation and consists of the distance of the forests left of i and j, of the distance of the subtrees rooted at i and j and of the costs for the map operation $(i \mapsto j)$ itself. For the case that $l(i_1) = l(i)$ and $l(j_1) = l(j)$ the forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$ are proper trees, so that there are no forests left of i and j. Thus Eq. (19) can be split into two cases. Again we assume that $i \in desc(i_1)$ and $j \in desc(j_1)$:

(Case 1) If $l(i_1) = l(i)$ and $l(j_1) = l(j)$:

$$D_f^{min}(l(i_1)..i, l(j_1)..j) = D_t^{min}(i, j) = \min \begin{cases} D_f^{min}(l(i_1)..i-1, l(j_1)..j) & + c(i \mapsto \varepsilon) \\ D_f^{min}(l(i_1)..i, l(j_1)..j-1) & + c(\varepsilon \mapsto j) \\ D_f^{min}(l(i_1)..i-1, l(j_1)..j-1) & + c(i \mapsto j) \end{cases} \tag{20}$$

(Case 2) Otherwise, i.e. $l(i_1) \neq l(i)$ or $l(j_1) \neq l(j)$:

$$D_f^{min}(l(i_1)..i, l(j_1)..j) = \min \begin{cases} D_f^{min}(l(i_1)..i-1, l(j_1)..j) & + c(i \mapsto \varepsilon) \\ D_f^{min}(l(i_1)..i, l(j_1)..j-1) & + c(\varepsilon \mapsto j) \\ D_f^{min}(l(i_1)..l(i)-1, l(j_1)..l(j)-1) & + D_t^{min}(i, j) \end{cases} \tag{21}$$

```
01: def compute_Dtree(T1,T2):
02:    Dt = array(1,|T1|,1,|T2|)
03:    LRKR1 = compute_LRKR(T1)
04:    LRKR2 = compute_LRKR(T2)
05:    for i1 in LRKR1:
06:       for j1 in LRKR2:
07:          Dtmin = compute_Dsubtree(i1,j1,Dt)
08:    return Dtmin
09:
10: def compute_Dsubtree(i1,j1,Dt):
11:    Df = array(l(i1)-1,i1,l(j1)-1,j1)
12:    initialize_Df(Df)
13:    for i in range(l(i1),i1):
14:       for j in range(l(j1),j1):
15:          if l(i) == l(i1) and l(j) == l(j1):
16:             Df[i,j] = compute_Df1(l(i1),i,l(j1),j)
17:             Dt[i,j] = Df[i,j]
18:          else:
19:             Df[i,j] = compute_Df2(l(i1),i,l(j1),j)
20:    return Dt[i1,j1]
```

Fig. 6. *Pseudo-Python code for minimum tree distance algorithm.*

where $D_t^{min}(T_1[i], T_2[j])$ or shorter $D_t^{min}(i, j)$ denotes the minimum tree distance between the subtrees rooted at i and j, respectively.

Our goal is to compute $D_t^{min}(T_1[|T_1|], T_2[|T_2|])$ which is equal to $D_t^{min}(T_1, T_2)$. As we can see from Eq. (21) this involves the computation of the tree distances $D_t^{min}(i, j)$ for all pairs of subtrees $(T_1[i], T_2[j])$, $1 < i < |T_1|$, $1 < j < |T_2|$. However, some of these distances are already available from the computation of $D_t^{min}(i_1, j_1)$. These are the pairs of subtrees whose roots i and j are in the paths of $l(i_1)$ to i_1 and $l(j_1)$ to j_1, respectively. Thus only the root node and all nodes with left siblings need separate computations. Formally, this set of nodes is called the *left-right keyroots* $LRKR(T)$ of T, and is defined as in [7] as:

$$LRKR(T) = \{k \mid \nexists\, k' > k \text{ such that } l(k') = l(k)\} \qquad (22)$$

For the example of Fig. 4, $LRKR(T_1) = \{6, 8, 9, 13, 14, 15\}$ and $LRKR(T_2) = \{4, 6, 8, 12, 13, 14\}$.

4.2. Dynamic programming style algorithm

Fig. 6 depicts some Pseudo-Python code for the minimum tree edit distance computation with a dynamic programming style algorithm. The main procedure is `compute_Dtree(T1, T2)`, where the values for D_t^{min} are stored in the permanent array `Dt`, which is allocated in line 02. Then the left-right keyroots are computed for both trees via `compute_LRKR(T)` and stored in ascending order. The `for` loops of lines 05 and 06 loop over all pairs of left-right keyroots, computing the minimum tree edit distance for the subtrees rooted at `i1` and `i2` via `compute_Dsubtree(i1, j1, Dt)`. In this procedure, first a temporary array `Df` for the minimum forest distances is allocated (see line 11). The column and row indices of `Df` correspond to the numbers of the nodes contained in the subtrees rooted at `i1` and `i2`. The additional first column and row correspond to the empty forests $\emptyset$, and are initialized in line 12 similar to Fig. 2. The `for` loops of lines 13 and 14 loop over all nodes of the subtrees in ascending order, and calculate in lines 16 and 19 the minimum forest distances after Equations (20) and (21), respectively. In line 17, forest distances that correspond to tree distances are stored in the permanent array `Dt`. Finally, the mapping (or mappings) leading to the minimum tree edit distance can be constructed similar to the sequence case by back-tracking the least-cost transitions from cell

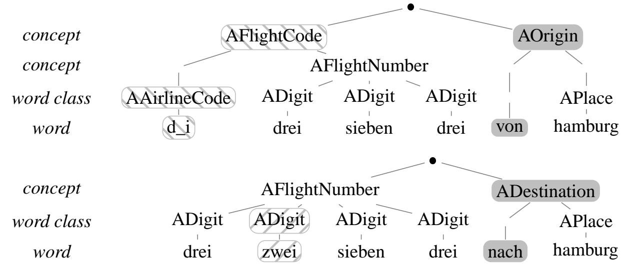

Fig. 7. *Semantic trees of Fig. 4, redrawn to reflect the node types.*

($|T_1|$,$|T_2|$) to cell (0,0) of the forest distance matrix `Df` during the last call of `compute_Dsubtree`. As [7] shows, the complexity of this algorithm is $O(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$.

5. EXTENSION TO TYPED TREES

As mentioned in the introduction, we extended the basic tree evaluation scheme to support trees consisting of labeled and additionally *typed* nodes. This need arises from the structure of the semantic model we employ, which consists of a hierarchy of probabilistic transition networks [2], whose labels correspond to the node labels of the trees that ODINS decodes. The network hierarchy is subdivided into a number of *hierarchy levels*, e.g. phonemes, words, word classes and semantic concepts. All networks of a hierarchy level share the same properties, which can be structural restrictions or search parameters. Fig. 7 shows the semantic trees of Fig. 4, redrawn so that a horizontal line through the tree only touches tree nodes of the same type (i.e. the hierarchy level). In Fig. 7 two important properties of the hierarchy can be seen, namely that a hierarchy level may consist of several sub-levels, and that sub-levels or even entire levels may be skipped.

We modified the cost function for the edit operations in order to avoid that tree nodes of different types are mapped, called *type substitutions*. Thus we only set the substitution cost to 4 (see Section 3) if the types of the mapped nodes are identical. Otherwise, the substitution cost is set to ∞. This modification has the effect that an insertion and a deletion is always preferred over a type substitution. It is also required to ensure that the 'dummy' root node, which we add to a semantic tree to yield a rooted tree and which is assigned its own type, is always mapped to the root node of the other tree. Note that this modification also implies that the tree distance is no longer a distance metric in the strict sense, as the triangle inequality (see Eq. (9)) does no longer hold.

6. COMPARISON WITH CONCEPT ACCURACY

In order to compute the concept accuracy Acc_c, we first have to convert the semantic trees into sequences of slot–value pairs. This is done by taking each leaf node as a value for the slot given by the concatenation of its ancestor nodes. The tree of Fig. 7 (top) would e.g. be converted to (slot,value) pairs *(AFlightCode.AAirlineCode, d_i), (AFlightCode.AFlightNumber.ADigit, drei), . . . , (AOrigin.APlace, hamburg)*, where a dot denotes a concatenation. The resulting sequences are then matched as described in Section 3 and the concept accuracy is computed from

# concepts	0	5	10	15	27	47
# concept sublevels	0	2	2	2	2	3
test-set perplexity	25.4	24.6	28.2	29.8	30.4	28.9
Acc_w	83.3%	83.4%	81.8%	81.9%	81.2%	82.9%
Acc_c	82.1%	82.3%	78.8%	77.3%	74.1%	58.0%
Acc_n	84.6%	85.9%	85.6%	85.5%	84.3%	82.6%
$Acc_n - Acc_c$	2.5%	3.6%	6.8%	8.2%	10.2%	24.6%

Table 1. *Tree and sequence matching evaluation results for models of different semantic complexity.*

the (slot,value) pair counts similar to Eq. (1):

$$Acc_c = \frac{C_c - I_c}{C_c + S_c + D_c} \tag{23}$$

Both the slots and the values of concepts must match to be counted as correct C_c. A substitution S_c is given if only the slot matches, if it doesn't a deletion D_c or insertion I_c is counted. For the example of Fig. 7 this would mean that *all* of the (slot,value) pairs would be rated as insertions and deletions because none of the slots match. Thus, the concept accuracy would be $Acc_c = \frac{0-6}{0+0+6} = -100\%$, whereas the tree node accuracy would be $Acc_n = \frac{9-2}{9+2+3} = 50\%$.

This example suggests that the two approaches can produce largely different results (which are admittedly extreme in the given example). Although both approaches honor partially correct interpretations, the tree matching approach displays greater flexibility in finding correspondences between partially correct substructures, whereas the concept accuracy metric always requires fully matching 'vertical' interpretations. Consequently, we expect the difference between the two evaluation metrics to be related to the complexity of the hierarchical model.

7. EXPERIMENTAL RESULTS

Based on a hierarchically annotated set of 1446 spoken utterances from 17 different speakers that were collected in a wizard-of-oz setup simulating a spoken dialogue system for an airport information system, we built 6 different hierarchical transition networks that incorporate varying degrees of semantic information. The most complex hierarchical model consists of 574 words, 11 word classes and 47 semantic concepts, and speaker-independent tied intra-word triphone HMMs with about 25k Gaussian mixture components, trained as described in [2]. For the global sentence model at the top of the hierarchy, backing-off bigrams were estimated. In order to reduce the degree of semantic information, we deleted varying numbers of semantic concepts from the annotation trees, and repeated the model building procedure. Thus all models have identical phoneme, word and word class levels, whereas the concept levels and the global sentence models differ. We built reduced complexity models with 27, 15, 10, 5 and 0 semantic concepts. The evaluations were performed on a test set of 233 utterances from 3 different speakers which are not contained in the training set. Table 1 displays the resulting word, tree node and concept accuracies along with the test-set perplexities of the different models. Note that the output trees of the least complex model (first column of Table 1) are only composed of words and word classes, but no semantic concepts. The bottom row contains the absolute differences between the tree node accuracy and the concept accuracy. The experimental data confirm our expectation, as the two approaches' difference continually rises with increasing model complexity, from 2.5% for the model without semantic concepts to 24.6% for the most complex model with 47 concepts and 3 concept hierarchy sub-levels.

8. CONCLUSION

We proposed the use of tree pattern matching methods for the evaluation of speech interpretation systems that produce hierarchically structured output. We reviewed the fundamental matching principle which is based on minimizing the tree edit distance, and outlined a suitable dynamic programming style algorithm. A modification of the cost function leads to an extension of the basic scheme, making it applicable to typed trees. An example was given in order to illustrate the argument that the tree node accuracy metric displays greater flexibility by taking the *structural correspondence* of trees into account, whereas the concept accuracy only rates fully matching interpretation slots as correct. We also presented experimental data confirming that the discrepancy of the two metrics rises with the complexity of the hierarchical model. Hence, we conclude that the tree node accuracy is especially suited for evaluation of speech interpretation systems that utilize complex hierarchical models. However, in practice the appropriateness of an evaluation metric also greatly depends on the capabilities of the component which further processes the semantic representation.

9. REFERENCES

[1] A. Simpson and N. M. Fraser, "Blackbox and glass box evaluation of the SUNDIAL system," in *Proc. Eurospeech*, Berlin, Germany, 1993, vol. 2, pp. 1423–1426.

[2] M. Thomae, T. Fabian, R. Lieb, and G. Ruske, "A One-Stage Decoder for Interpretation of Natural Speech," in *Proc. NLP-KE'03*, Beijing, China, October 2003.

[3] M. Boros, W. Eckert, F. Gallwitz, G. Görz, G. Hanrieder, and H. Niemann, "Towards Understanding Spontaneous Speech: Word Accuracy vs. Concept Accuracy," in *Proc. ICSLP*, Philadelphia, PA, 1996, vol. 2, pp. 1009–1012.

[4] W. Minker, "Evaluation Methodologies for Interactive Speech Systems," in *Proc. LREC'98*, Granada, Spain, May 1998, pp. 199–206.

[5] R. Bod, "Spoken Dialogue Interpretation with the DOP Model," in *Proc. COLING/ACL98*, San Francisco, California, 1998, pp. 138–144, Morgan Kaufmann Publishers.

[6] Gertjan van Noord, G. Bouma, R. Koeling, and M.-J. Nederhof, "Robust Grammatical Analysis for Spoken Dialogue Systems," *JNLE*, vol. 5, no. 1, pp. 45–93, 1999.

[7] D. Shasha and K. Zhang, "Approximate Tree Pattern Matching," in *Pattern Matching Algorithms*, A. Apostolico and Z. Galil, Eds., chapter 14. Oxford University Press, 1997.

[8] D. Sankoff and J. B. Kruskal, Eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison Wesley, 1983.

[9] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland, *The HTK Book (for HTK Version 3.2)*, CUED, 2002.

[10] William M. Fisher and Jonathan G. Fiscus, "Better Alignment Procedures for Speech Recognition Evaluation," in *Proc. ICASSP*, Minneapolis, Minnesota, 1993, pp. 59–62.