Gefördert vom
bmb+f
Bundesministerium für
Bildung, Wissenschaft,
Forschung und Technologie

Verbmobil
Verbundvorhaben

What's in a Word Graph
Evaluation and Enhancement of Word Lattices

Jan W. Amtrup
Henrik Heine
Uwe Jost

Universität Hamburg

Vm
Report 186
Dezember 1996

Dezember 1996

Jan W. Amtrup
Henrik Heine
Uwe Jost

Arbeitsbereich Natürlichsprachliche Systeme
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg

Tel.: (040) 54 94 - 2519/2521/2522
Fax: (040) 54 94 - 2515
e-mail: {amtrup|heine|jost}@informatik.uni-hamburg.de

Gehört zu den Antragsabschnitten: 15.1/4 (Modul-Interaktion und Hypothesenverwaltung) und 15.3 (Phonologisch informierte akustische Analyse)

Das diesem Bericht zugrundeliegende Forschungsvorhaben wurde mit Mitteln des Bundesministers für Bildung, Wissenschaft, Forschung und Technologie unter dem Förderkennzeichen 01 IV 101 A/O gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei den Autoren.

Abstract

During the last few years, word graphs have been gaining increasing interest within the speech community as the primary interface between speech recognizers and language processing modules. Both development and evaluation of graph-producing speech decoders require generally accepted measures of word graph quality. While the notion of recognition accuracy can easily be extended to word graphs, a meaningful measure of word graph size has not yet surfaced.

We argue, that the number of derivation steps a theoretical parser would need to process all unique sub-paths in a graph could provide a measure that is both application oriented enough to be meaningful and general enough to allow a useful comparison of word recognizers across different applications.

This paper discusses various measures that are used, or could be used, to measure word graph quality. Using real-life data (word graphs evaluated in the 1996 Verbmobil acoustic evaluation), it is demonstrated how different measures can affect evaluation results.

Finally, some algorithms are presented that can significantly improve word graph quality with respect to specific quality measures.

Contents

Chapter 1

Introduction

The success of language processing modules within a speech understanding (or interpreting) system depends heavily on the quality of the outcome of the first stage in processing, the speech recognizer. The most widely used interface between a word recognizer and subsequent language understanding modules consists of a chain of words representing the sequence of words best matching the acoustic input signal (*best chain* recognition). It is highly advantageous to be used as add-on to already existing language processing systems, since the input has the same form as written input (save for punctuation and capitalization).

When the recognition rate of a speech recognizer is sufficiently high, the information provided by the *best chain* may suffice for the language processing modules. In many cases, however, these modules either require perfect input or the average number of errors in the *best chain* is simply to high. In these cases, the recognizer has to pass more information to subsequent processing steps. The simplest way to do this is to present a list of different word sequences, which represent the *n-best chains* the recognizer was able to detect. Language processing modules may thus choose among the set of possible utterances presented by the recognizer.

However, it is usually not enough to deliver just the best 10 or 20 utterances, at least not for reasonable sized applications given todays speech recognition technology. To significantly increase overall system performance, n (the number of utterance hypotheses) has to be quite large. It increases exponentially with the length of the sentence (Oerder and Ney, 1993).

Word graphs offer a simple and efficient way to represent an extremely high number of competing hypotheses and have therefore become very popular as the primary interface between speech recognizers and language processing modules (Oerder and Ney, 1993), (Aubert and Ney, 1995).

In order to improve speech recognition systems, a reliable measure of system performance is needed. Most well known evaluations (e.g. Resource Management, Wall-Street Journal, Switchboard) only consider the *best chain*. It is often assumed that the best system with respect to the best chain will also be the best system to produce word graphs. We think it may be misleading to rely on a secondary measure of system performance (provided we consider word graphs as primary output), because it is not at all clear why the assumption mentioned above should hold. The evaluation results of the 1996 Verbmobil accoustic evaluation (Reinecke, 1996) provide some empirical evidence to question this assertion.

Previous attempts to directly evaluate word graphs have been hampered by the lack of a meaningful measure of word graph quality (Lehning, 1996). While the notion of word accuracy can easily be extended to the best-fitting path through a word graph (Paulus and Lehning, 1995), it is much harder to find a meaningful measure of word graph size. In the following chapters, various measures of the size and quality of word graphs are discussed and some methods to improve word graph quality are presented.

In the second chapter, a formal definition of word graphs is given and some properties of word graphs are explained. Chapter 3 presents the "classical" methods of evaluation of word graphs together with some new results from the latest Verbmobil speech evaluation, which took place in September 1996. We will argue that the measures used there do not fully account for the criteria of languague processing modules regarding problem size and thus tractability of processing. Chapter 4 tries to motivate a different kind of measurement for the problem size and gives some examples, again derived from the latest Verbmobil evaluation. Finally, chapter 5 presents a number of methods that modify word graphs with the aim of reducing word graph complexity in order to simplify language processing.

Chapter 2

Definition and properties of word graphs

To set up some terms we will use for the rest of the paper, we start by defining word graphs and some of their graph-theoretic properties. We do not start with fundamental definitions of graphs, but instead directly begin with word graphs.

Definition 1 *[Word graph]*
A word graph is a directed, acyclic, weighted, labelled graph with distinct root and end vertices. It is a quadruple $G = (\mathcal{V}, \mathcal{E}, \mathcal{W}, \mathcal{L})$ with the following components:

- *A nonempty set of vertices $\mathcal{V} = \{v_1, \ldots, v_n\}$. Each vertex may be associated with additional information, say the point in time represented by a certain vertex (see below for functions that return specific properties of vertices).*
- *A nonempty set of weighted, labelled, directed edges $\mathcal{E} = \{e_1, \ldots, e_m\} \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{W} \times \mathcal{L}$. The edges are directed, i.e. in general $(v, v', w, l) \in \mathcal{E} \not\Longrightarrow (v', v, w, l) \in E$. In particular, this would be forbidden by the acyclicity of the graph, too (see below).*
- *A nonempty set of edge weights $\mathcal{W} = \{w_1, \ldots, w_p\}$. Edge weights normally represent a score modelling the degree of similarity between the acoustic signal and a model for a particular word.*[1]

[1]Usually, edge weights are modelled as a function $w : \mathcal{V} \times \mathcal{V} \times \mathcal{L} \longrightarrow R$, but this would prevent edges identical except for acoustic score, which will arise from time to time.

- *A nonempty set of Labels* $\mathcal{L} = \{l_1, \ldots, l_o\}$, *which represents information attached to an edge. Commonly, each edge bears the word label, representing the lexicon entry that was recognized.*

We define some relations to access several properties of vertices and edges:

- Accessor functions for vertices
  Generally, vertices are attached to certain points in time representing the progress of the utterance. We measure this time in $10ms$ frames, *and define a function* $t : \mathcal{V} \longrightarrow N$, *that returns the frame number for a given vertex.*

- Accessor functions for edges
  To allow for easier reading of formulae we will use in the following, we define seperate functions to access components of an edge. In particular, let $e = (v, v', w, l) \in \mathcal{E}$ *be an edge of the word graph. Then we define:*

  – *The accessor function for start vertices of edges as*
  $$\alpha : \mathcal{E} \longrightarrow \mathcal{V}, \alpha(e) := v \tag{2.1}$$
  – *The accessor function for end vertices of edges as*
  $$\beta : \mathcal{E} \longrightarrow \mathcal{V}, \alpha(e) := v' \tag{2.2}$$
  – *The accessor function for edge weight as*
  $$w : \mathcal{E} \longrightarrow \mathcal{W}, w(e) := w \tag{2.3}$$
  – *The accessor function for edge labels as*
  $$l : \mathcal{E} \longrightarrow \mathcal{L}, l(e) := l \tag{2.4}$$

- Reachability of vertices
  We define the relation of reachability *for vertices* ($\rightarrow$) *as follows:*
  $$\forall v, w \in \mathcal{V} : v \rightarrow w \Leftrightarrow \exists e \in \mathcal{E} : \alpha(e) = v \wedge \beta(e) = w \tag{2.5}$$
  The transitive hull of the reachability relation $\rightarrow$ *is denoted by* $\stackrel{*}{\rightarrow}$.

- Paths through graphs
  A path through a graph (or a subgraph) is the sequence of edges traversed on that path. We denote the sequence as $\mathrm{e}_{[1;K]}$, *the ith element of the sequence as* ${\mathrm{e}_{[1;K]}}^{(i)}$. *Each edge must be adjacent to the next one, i.e.* $\forall i \in \{1, \ldots, k-1\} : \beta({\mathrm{e}_{[1;K]}}^{(i)}) = \alpha({\mathrm{e}_{[1;K]}}^{(i+1)})$. *We call the number of edges in the sequence* (k) length *of the sequence. To express that two vertices are connected by a sequence of edges, we use* $v \xrightarrow{E} w$.

- Degree functions for vertices
  For several purposes, we want to know how many edges start from (are incident from) or end at (are incident to) a given vertex.

  To this behalf, we define the function $\#_{in} : \mathcal{V} \longrightarrow N$ to return the number of edges incident to a vertex:

  $$\#_{in}(v) := \#\{e \in \mathcal{E} | \beta(e) = v\} \tag{2.6}$$

  Analogously, we define $\#_{out} : \mathcal{V} \longrightarrow N$ to return the number of edges incident from a vertex:

  $$\#_{out}(v) := \#\{e \in \mathcal{E} | \alpha(e) = v\} \tag{2.7}$$

Additionally to the properties stated above, for a graph to be a word graph, the following must hold true:

- *The graph must be* acyclic, *meaning that there is no sequence of edges emerging from one vertex and ending in the same vertex:*

  $$\forall v \xrightarrow{*} w : v \neq w \tag{2.8}$$

- *The graph contains a distinct root vertex. This vertex represents the start of the utterance:*

  $$\exists v^{(r)} \in \mathcal{V} : \#_{in}(v^{(r)}) = 0 \wedge \forall v \in \mathcal{V} : \#_{in}(v) = 0 \Longrightarrow v = v^{(r)} \tag{2.9}$$

  Note that the property of being acyclic and rooted entails the connectivity of the graph, i.e. every vertex is reachable from the root vertex.

- *The graph contains a distinct end vertex. This vertex represents the end of the utterance:*[2]

  $$\exists v^{(f)} \in \mathcal{V} : \#_{out}(v^{(f)}) = 0 \wedge \forall v \in \mathcal{V} : \#_{out}(v) = 0 \Longrightarrow v = v^{(f)} \tag{2.10}$$

An important property of word graphs that has consequences for the complexity (and execution time) of several algorithms is the possibility to impose a topological ordering on them.

[2]Note that this property does not hold when dealing with incremental word graphs.

Definition 2 (Topological ordering) *A topological ordering is a function $\tau : V \longrightarrow N$ on the vertices of a directed graph having the property that*

$$\forall e \in \mathcal{E} : \tau(\alpha(e)) < \tau(\beta(e)) \tag{2.11}$$

The natural topological ordering for a word graph is given by time stamps of vertices. Acyclicity and topological ordering are consequences of the fact that words are uttered linearily in time. A word graph can additionally be viewed as a lattice; more precisely, the set of vertices is a lattice, if we consider $\rightarrow$ to be the ordering relation for vertices (cf. Carreé (1979, p. 23)). The topological ordering can be established in $O(|\mathcal{E}| + |\mathcal{V}|)$ time (cf. Cormen, Leiserson, and Rivest (1990, p. 486).

However, word graphs in general are not planar. This can be either shown by homeomorphism of subgraphs to special nonplanar graphs (cf. McHugh (1990, p. 33)) or simpler by using a theorem on the linear bound on the number of edges (cf. McHugh (1990, p. 38)). Thus, several algorithms applying to planar graphs only can not be used.

The most advantegous property of word graphs is their compactness. Even very small graphs (in terms of number of edges, the most commonly used size measure for word graphs) can contain a huge number of different utterance hypotheses, which makes this representation superior to n-best interfaces. Consider the graph in Figure 2.1, one of the early Verbmobil toy graphs. It consists of only 97 edges, yet contains 1380 paths.

Coarticulation poses a problem for any kind of modelling of speech, which can be partly overcome by using a graph as representation. If a german speaker utters trag's (trage and es glued together), it is at best problematic to retrieve the relevant words. The reduced form is in general not part of the system's lexicon. Both unreduced lexems may be in the graph, but most certainly they overlap in time, meaning there is no path through the graph which covers both edges. There are methods to handle edges with small overlaps, the simpest of them abandoning word graphs and retreating to sets of words without a connection condition. But this solution poses too much other problems to be adequate for these cases. Below, we will present an algorithm which unifies vertices that are close together in time under certain circumstances; for a large number of coarticulated cases, this might help.

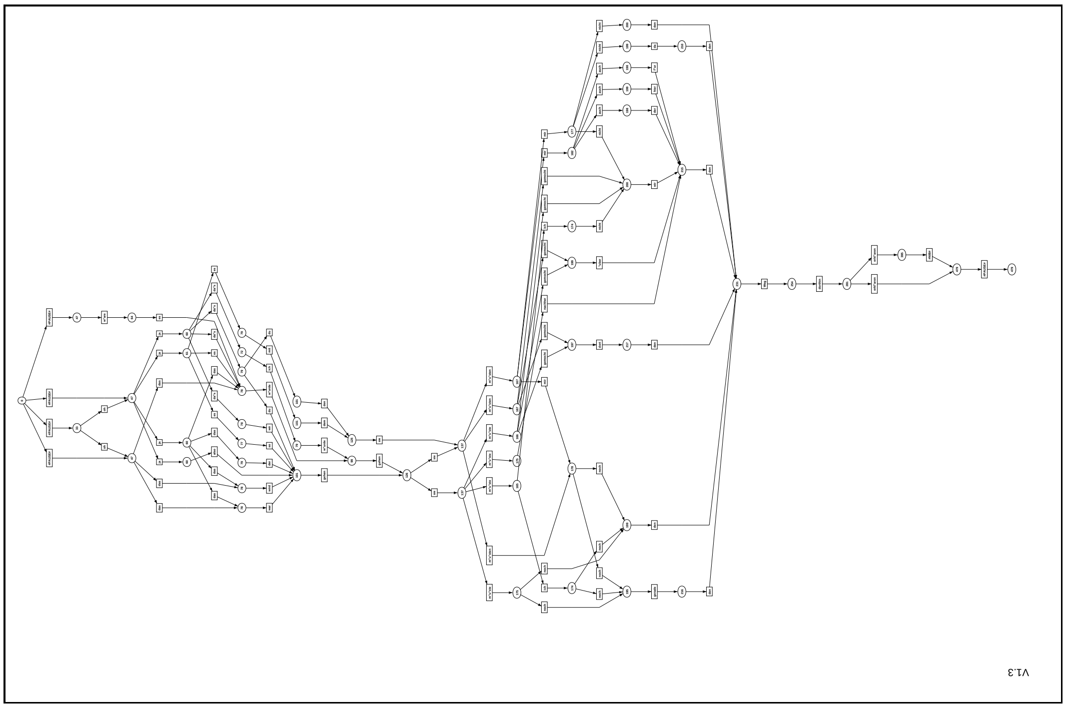

Figure 2.1: A simple Verbmobil wordgraph representing the utterance "ja das wuerde gehen sie muessten mir nur noch grade den weg dorthin erklaeren"

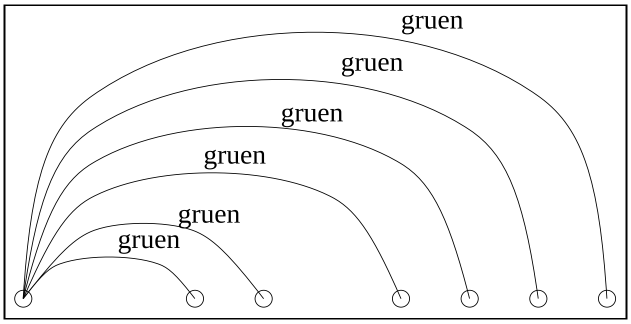

Figure 2.2: A family of edges for the word gruen

But there are problems present with word graphs, too. The worst, from a point of view of speech recognition evaluation and language processing, are:

- The *density of graphs* is sometimes too high to allow reasonable processing times for language modules. Density is measured as the mean number of edges spanning a certain point in time. The reason to hunt for a high density is quite clear: The higher the number of edges, the higher the probability of the right edge being present where it was actually spoken. On the other hand, language processing modules work best when given sparse graphs.

  Theoretically, word graphs (and, in general, sets of word hypotheses) can be parsed in cubic time complexity, assuming context free grammars (cf. Weber (1995, p. 29)). For example, given an augmented Tomita Parser (Kipps, 1991), the complexity is $O(n^3|G|^2)$, where n denotes the number of vertices in the graph and $|G|$ denotes the number of symbols in the grammar given. Ambiguity present in a graph does not change the complexity either. Even if a word hypothesis e_{w1} spans several other, there is no need to handle this case through a grammar rule that maps e_{w1} (or, rather, the preterminal category of it) to the sequence of edges covered. You could simply add the category at the proper point in the derivation matrix.

  In practical, dense graphs are not efficiently computable. The first reason for this is that language modules are not constrained to context free grammars, but often use more powerful models, like complex feature based formalisms. Second, although the theoretical bound holds even in case of lexical ambiguity, the linear burden on a parser is tremendous and must be avoided if you want to achieve reasonable computing times.

- *Families of edges* form the last difficulty we want to present here. They arise due to the nature of speech recognizers which try to recognize words wherever they can. A family of edges consists of edges with identical labelling (word form) that end at vertices being close together in time. Figure 2.2 shows an example of a family of edges belonging to the word gruen. Note that each edge starts at the same vertex, but the endpoints are distant from one another for one or two frames.

  To treat those edges individually in a parser results in a huge processing overhead. The goal again is to unite vertices to reduce the number of edges to be used (ideally, one for each family). Note that the definition of families (as used by Weber (1995)) can be extended to cover edges with close start vertices, too. The treatment taken in Weber (1995) then has to be extended to use directed hypergraphs as a representation for word graphs (cf. Gondran and Minoux (1984, p. 30)).

Chapter 3

Conventional graph evaluation methods

If graphs are evaluated at all, the quality measure used is usually a straightforward extension of the word accuracy measure as used for the best hypothesis:

$$\text{word accuracy} = 100 - 100 \cdot \frac{\#\text{errors}}{\#\text{words in transcription}}$$

$$\#\text{errors} = \#\text{substitutions} + \#\text{deletions} + \#\text{insertions}$$

The word accuracy of the graph is then just the word accuracy of the path through the graph with the best rating according to this measure (Paulus and Lehning, 1995), (Reinecke, 1996), (Oerder and Ney, 1993), (Tran, Seide, and Steinbiss, 1996).

The size (or density) of the graph is usually defined as the average number of edges per (transcribed) word (e.g. (Aubert and Ney, 1995), (Woodland et al., 1995)). This measure is rather vague, as figures 3.1 and 3.2 demonstrate. Even without any formal definition of graph size or complexity, it seems intuitively clear that the graph in figure 3.2 is somehow "bigger". However, both graphs have the same number of edges.

To account for this, the average number of incoming or outgoing edges per vertex is sometimes taken into account (Reinecke, 1996). As pointed out by Lehning (1996), these two measures don't allow to directly judge the complexity of a graph. As an illustration, consider figures 3.3 and 3.4. The first one represents the results

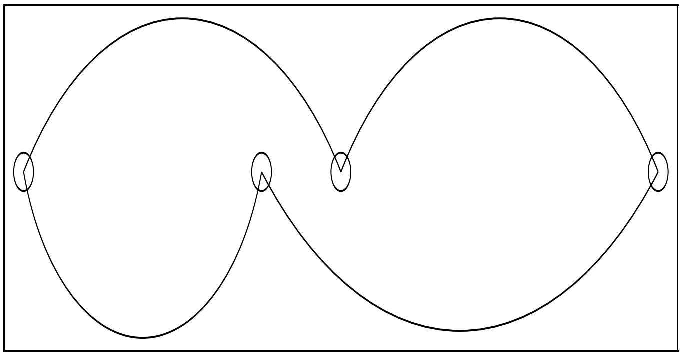

Figure 3.1: example graph

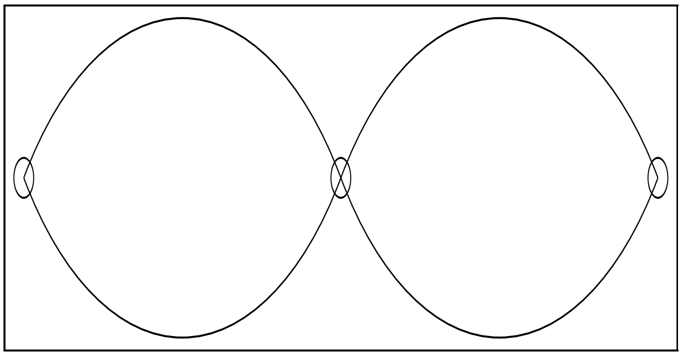

Figure 3.2: example graph

of the 1996 Verbmobil acoustic evaluation in the main category (see Reinecke (1996) or `http://www.sbvsrv.ifn.ing.tu-bs.de/eval.html` for more details). The word accuracies are plotted against the graph densities (edges per word) in figure 3.3, while the same accuracies are plotted against the average number of edges incident from a vertex in figure 3.4. The comparison[1] shows that the two measures might suggest different ratings (compare, for instance, the curves for "TUM2" with "FP2" or "DB" with "UHH"). It would also be hard to find a sensible way to combine the two measures.

It should be noted that both charts would not support the assumption that the recognizer that is better on the best hypotheses will also be better on graphs (e.g. compare "DB" with "FP2").

[1]The "edges per vertex" charts were not part of the official evaluation in 1996.

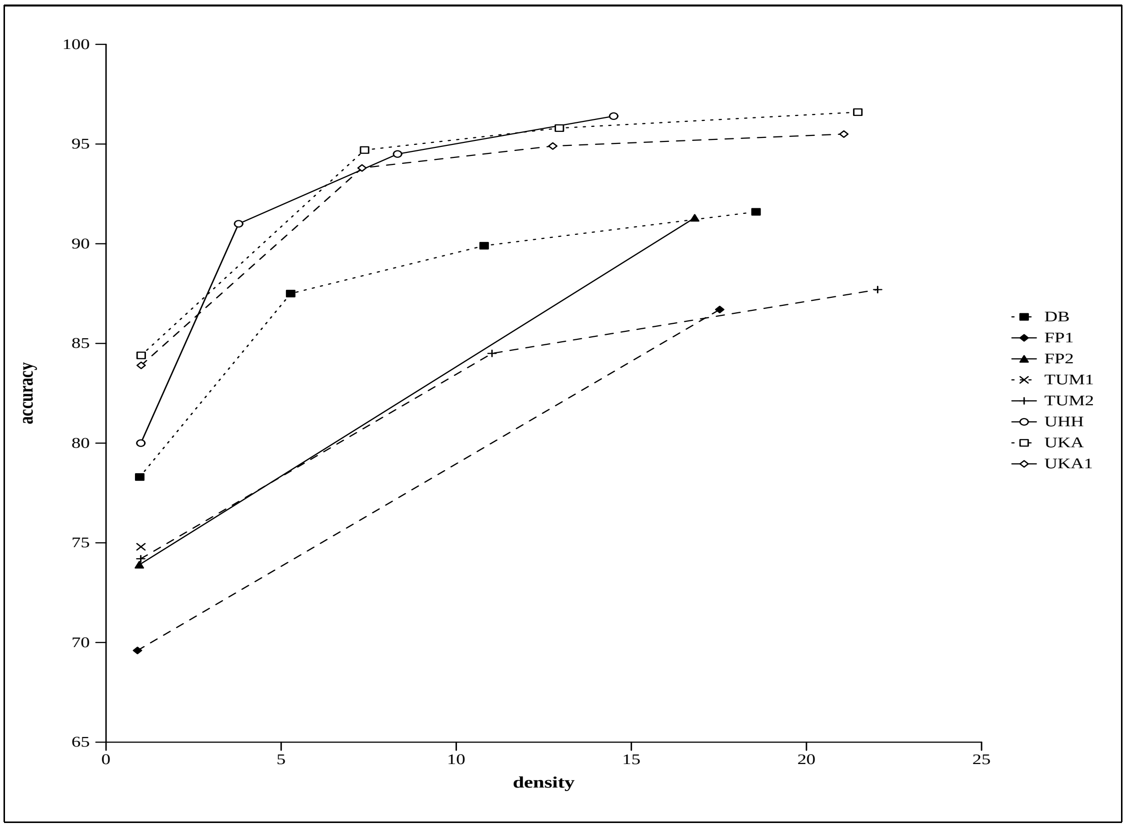

Figure 3.3: Results of the 1996 Verbmobil acoustic evaluation (cat. t1)

Three alternative measures of graph complexity are proposed in Lehning (1996):

1. number of all paths through the graph
2. some combination of the two measures: number of edges and average number of outgoing edges
3. error rates of randomly selected paths

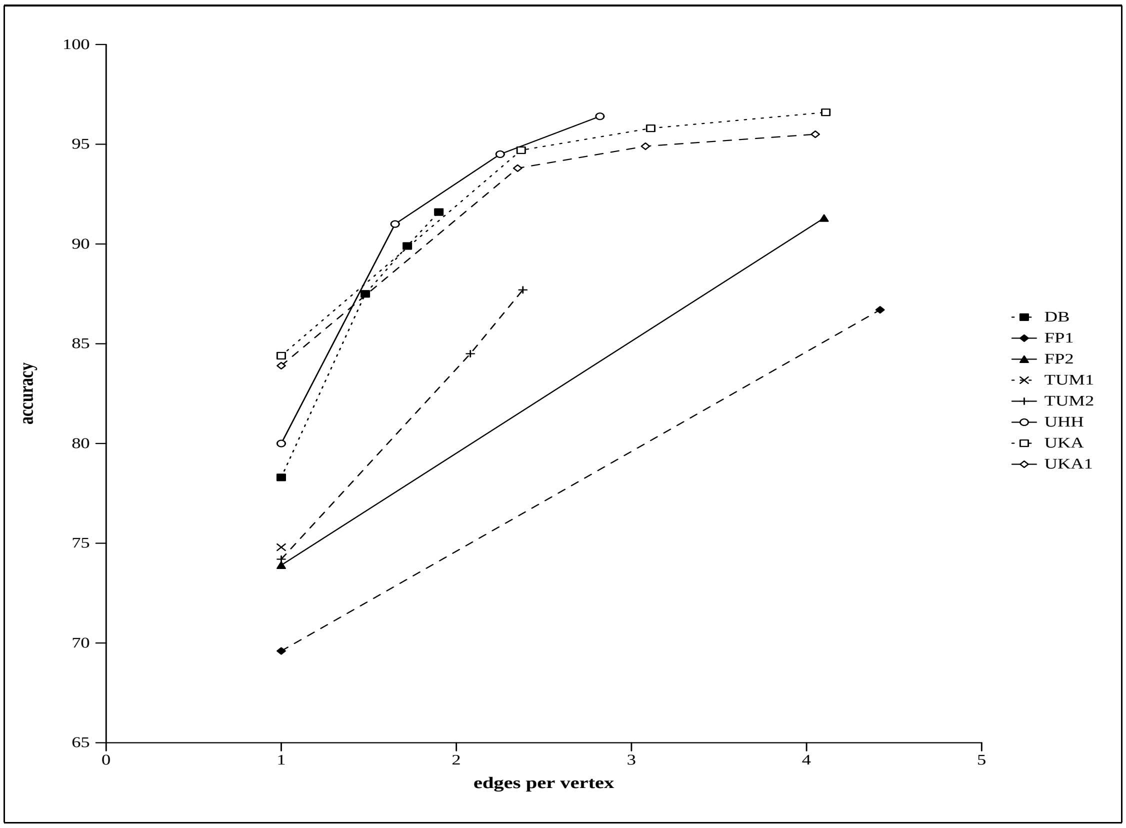

Figure 3.4: Inofficial evaluation results

Chapter 4

Application-oriented evaluation of word graphs

In the last chapter, we have presented conventional methods to evaluate the performance of word recognition. The size of graphs was measured using edge-based counts, e.g. the number of edges per word in a reference transliteration. In this chapter we will set out a new measure for graph size that — as we will argue — is more suitable if one wants to evaluate speech recognition in the broader context of a large speech understanding system.

4.1 Integrated measures

Provided the correct word sequence is in a graph, one could ask: "How much information is needed to select the correct path among all the paths in the graph?". There are (at least) three measures that could be used to answer this question[1]:

- the (renormalised) probability of the correct path according to the graph
- the rank of the correct path in the graph
- the number of processing steps a component (e.g. a probabilistic parser) would have to carry out before it arrives at the correct path

[1]In sections 4.3.3 and 4.3.4 we present algorithms to compute two of them.

Problems with these measures include:

- The correct path is often not in the graph at all.
- The results can not be compared with the evaluation results for best-chain evaluations.

4.2 New measures for word graph evaluation

The assumption we will use here is that the output of a word recognizer cannot be evaluated isolated from the components that use that output. We assume that a parser is the primary module using word graphs for further analysis. In particular, we assume that it uses a formalism based on complex features. This prevents the use of algorithms with a cubic time complexity, as we may assign different structures to each two edges or constituents covered by a rule of the syntax (e.g. by defining a feature that holds the sequence of word labels attached to the words the edge spans). We constrain ourselves to grammars that use a context-free backbone for convenience, and further constrain the rules to have at most two right-hand side nonterminals, i.e. the grammars are in Chmomsky normal form. We do not, however, apply constraints regarding search strategy or pruning mechanisms.

These assumptions have serious consequences for parsing complexity, as we cannot hold a fixed size set of nonterminal categories for each interval of graph vertices (which leads to polynomial complexity, see above). Instead, we have to produce one derivation step for each pair of edges (word hypotheses or complex edges covering more than one word) to simulate the processing of a parser.[2]

A first attempt to establish a measure for word graph size relevant for the overall performance of a speech understanding system is to investigate into the number of paths in a graph. This measure is motivated by the fact that a parser may construct an analysis for each sequence of word hypotheses covering the whole utterance. A wordgraph may have as many as $(\frac{|\mathcal{E}|}{|\mathcal{V}|})^{|\mathcal{V}|-1}$ paths in it, if the edges

[2]Since a grammar may contain ambiguities, there may be more than one derivation step for each pair of edges. We abstract from this fact as well as from the fact that not every pair of edges may be combined due to grammar restrictions. Thus, we do only take into account the consequences obtainable from the raw input, the word graph, and set aside properties of grammars and such.

are distributed evenly among the graph. Determining the number of paths in a graph can be done efficiently in $O(|\mathcal{E}| + |\mathcal{V}|)$ time given the acyclicity of word graphs. The simple algorithm is outlined in figure 4.4 below. Figure 4.1 shows results obtained for two sets of word graphs where the size of graphs is measured as the number of paths in the graph.

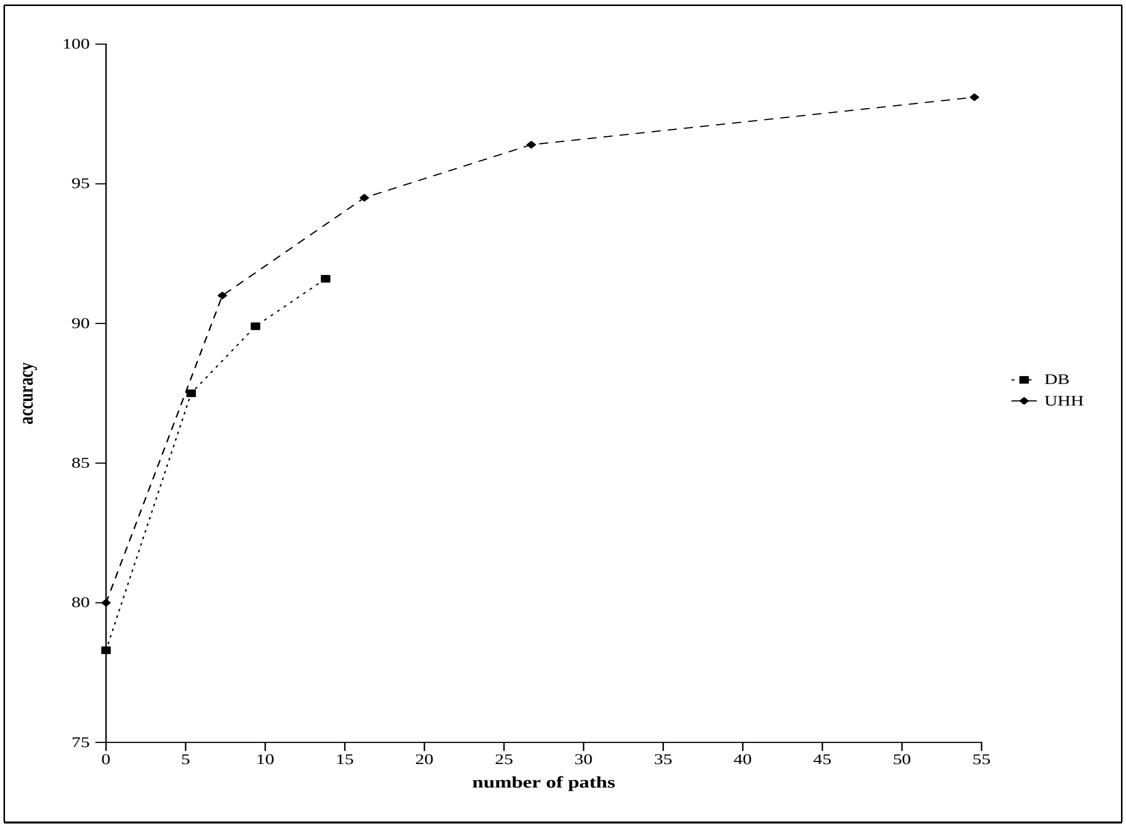

Figure 4.1: Recognition accuracy vs. number of paths (in 10^x) (cat. t1)

The next possible extension is to reduce the graph to only contain unique word sequences. The motivation behind this modification of a graph, which is described below in section 5.1.3, is the observation that two identically labelled yet different paths through the graph can only differ regarding two pieces of information[3]:

- The vertices the paths visit. This should not bother a parser, since the mapping from words to exact intervals in time may be irrelevant. Note that this is not the case if information other than the words given by a speech

[3] For an account of two paths identical except for silence, cf. section 5.1.2.

recognizer is to be taken into account, e.g. prosodic information which may well be bound to specific time intervals.

- The acoustic scores the words are annotated with. The algorithm should preserve the order in which different paths are ranked according to acoustic score.[4] If that is not possible (as with our algorithm), the least it should do is always to retain better word scores, thus only augmenting the overall score of a path. Given this schema, we only need to hold one path for each unique word sequence.

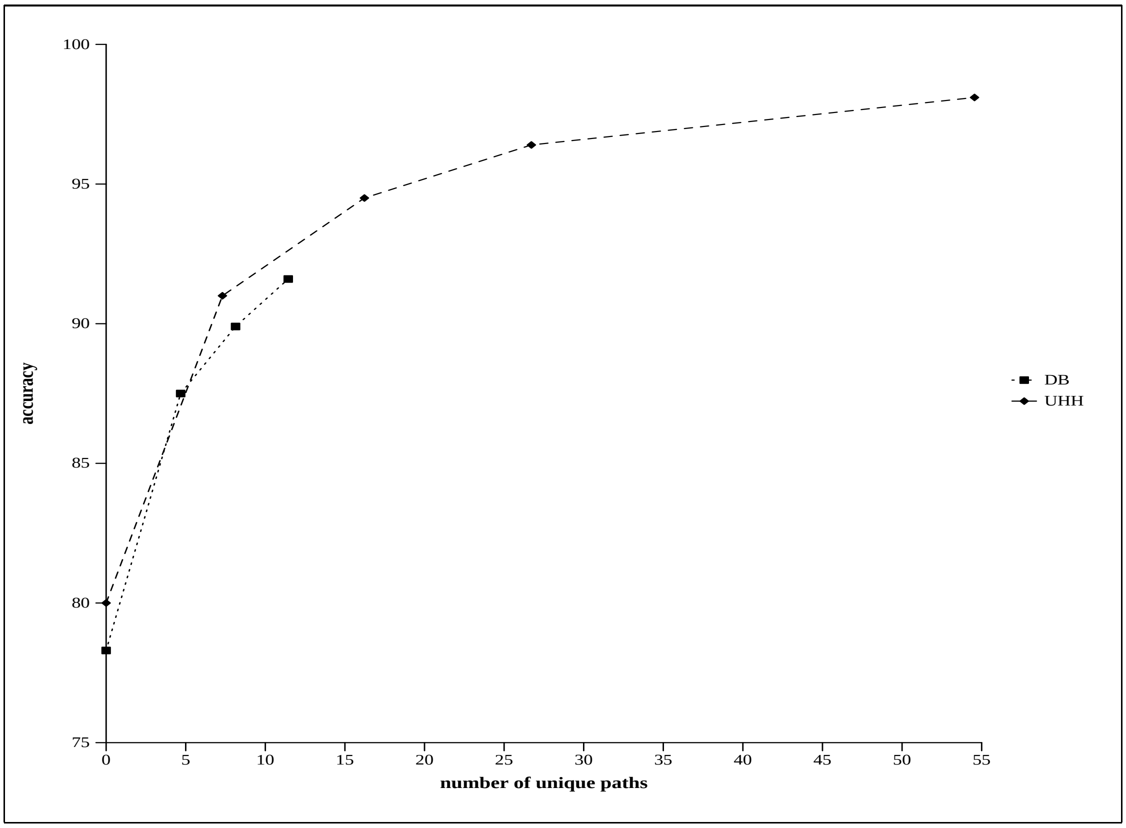

Figure 4.2: Recognition accuracy vs. number of unique paths (in 10^x) (cat. t1)

The algorithm to achieve unique label sequences comes in two flavors: The first version acts information-preserving, as defined in section 5.1. It has proven to be unpractical, as it creates a very large number of new vertices and edges, yielding a graph that cannot be efficiently handled. The second version abandons with the information-preserving property, i.e. the scores of partial paths may be

[4] Cf. the exact definition of information-preserving operations in section 5.1.

changed and, according to parametrisation of the algorithm, new paths may be introduced. It preserves, however, the property to modify only the scores of paths. This version has been used for graph evaluation purposes.

Unfortunately, the information-preserving algorithm has exponential time complexity in the worst case (cf. section 5.1.3), so it is not applicable in the general case. By using the version that does not preserve information and by introducing some optimization, we could reduce run time for most graphs to a reasonable amount of time[5]. That way, we can at least compare graph sizes using that method. The outcome of an evaluation based on the number of unique paths to determine the size of a graph is shown in figure 4.2.

The next step in the direction of a suitable size measurement is to account for the number of derivations a parser has to carry out in order to fully parse a word graph. We first consider the analysis of one path and extend our argument to derivations over a full graph.

The number of derivation steps ($d^{(p)}$) for a full analysis of one path (p) through the graph is

$$d^{(p)} = \frac{n^3 - n}{6} \tag{4.1}$$

, where n denotes the length of the path, i.e. the number of edges covered by it. The number of derivation steps directly corresponds to the number of processing steps needed to fill the derivation matrix of the CKY-algorithm (cf. Mayer (1986, p. 107)). Note again that (4.1) does not entail that parsing with complex feature based grammars is cubic. The only property extending over context-free parsing we use in our argument (namely not to guarantee a fix-sized set of hypotheses at any vertex) prevents us from incorporating many paths into one operation.

If we assume that all paths through the graph are independent of each other, the total number of derivations is

$$d^{(G)} = \sum_{p \in G} d^{(p)} \tag{4.2}$$

[5]Nevertheless, it is still exponential in the worst case.

, which gives a linear dependency between the number of paths and the number of derivations in a graph. However, subparts of a graph are shared among different paths. Thus, the formula above is only an upper bound. To account for subgraph sharing, we have to use a slightly more complex algorithm, given in figure 4.5 below. The complexity of this algorithm yields $O(|\mathcal{E}||\mathcal{V}|)$. The method used to compute the number of derivation assumes some kind of chart parser which does not compute partial analyses twice. Shared left contexts are preserved, thus only adding once to the desired number.

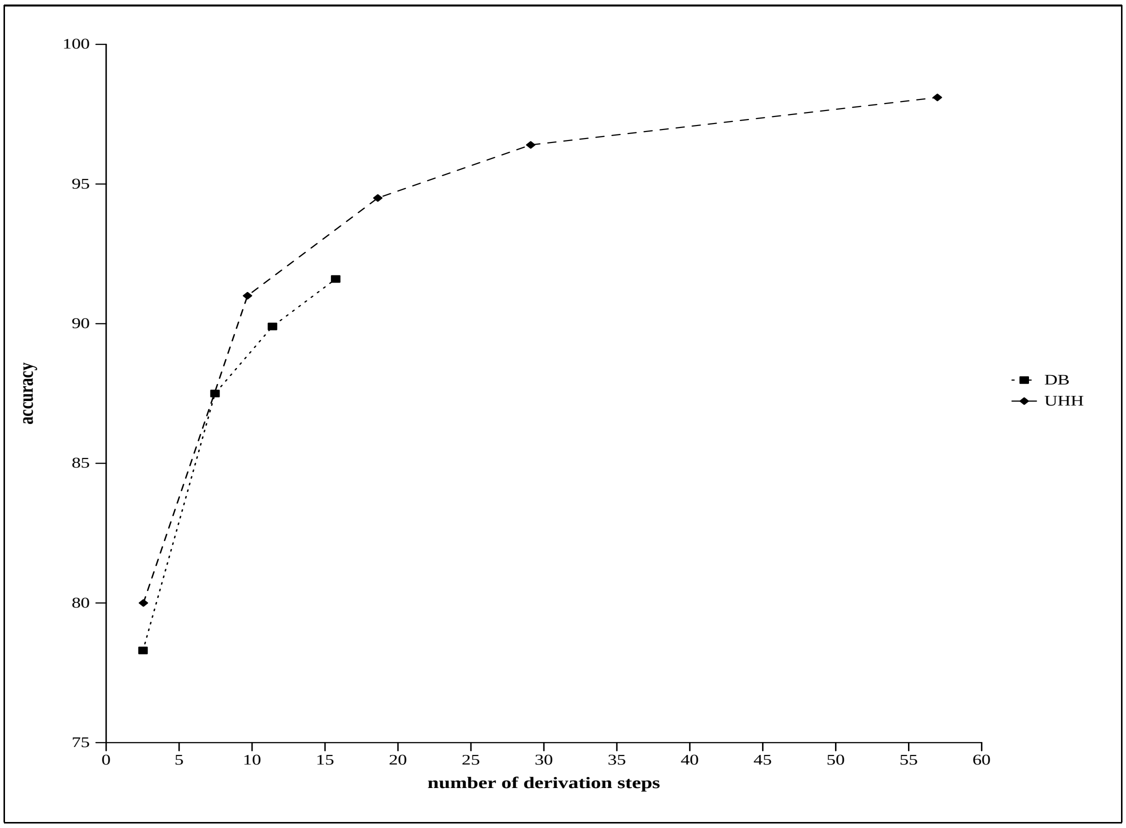

Figure 4.3: Recognition accuracy vs. number of derivation steps (in 10^x) (cat. t1)

By using the number of derivations generated by this algorithm we take into account the impact of different graph shapes onto parsing effort (cf. chapter 3). Thus, given two graphs with identical number of paths, the graph that has the largest amount of subgraph sharing in it will be the best one to parse. To give an example, consider the graph shown in figure 5.11. It consists of 37 vertices and 260 edges. Without any modification, there are $1.4 \cdot 10^9$ paths in it. If all paths are treated independently of each other, a parser would build $9.48 \cdot 10^{11}$

derivations, while sharing only yields $8.91 \cdot 10^{10}$ derivations. Constraining the graph to unique label sequences reduced the graph to 172 edges (the number of vertices rises to 57). After modification, the graph contains 8221 (different) paths, and a parser with sharing would construct $5.88 \cdot 10^5$ derivations only. An example of an evaluation based on the number of derivation steps is shown in figure 4.3

If we are to evaluate more than one graph, we use a geometric average instead of simple arithmetic average. This prevents an overly large influence from the largest graph in the set.

4.3 Evaluation procedures

In this section, we describe in detail the algorithms outlined in the sections before. For each method mentioned, we give an abstract definition of the algorithm and also state some complexity observations.

4.3.1 Number of paths

The algorithm in figure 4.4 determines the number of paths in a word graph. The input to the algorithm is a word graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{W}, \mathcal{L})$ as defined in definition 1. The output is the number of paths through that graph, $p^{(G)}$.

The complexity of the algorithm yields $O(|\mathcal{E}|+|\mathcal{V}|)$. Although we use two cascaded loops, the outer loop (line [1]) solely determines the order in which edges are accounted for. Since each edge ends at a single vertex, no edge is considered twice.

4.3.2 Number of derivations

The algorithm in figure 4.5 determines the number of derivation steps a reasonable chart parser has to compute in order to fully parse a word graph. The assumptions made are

- There is no ambiguity present in the underlying grammar.

begin

Handle each vertex

[1] **for** each vertex $v \in \mathcal{V}(G)$, taken in topological order, **do**

Compute the number of paths ending at v

$p^{(v)} := \sum p^{(w)} : (w, v, x, y) \in \mathcal{E}$

[2] $p[v] \longleftarrow 0$

[3] **for** each edge $e = (w, v, x, y)$ ending in v **do**

[4] $p[v] \longleftarrow p[v] + p[w]$

The number of paths in the graph is the number of paths up to the final vertex

$p^{(G)} \longleftarrow p^{(v^{(f)})}$

[5] **return** $p^{(G)}$

end

Figure 4.4: Determining the number of paths in a word graph

- There is no restriction to the number of partial analyses the parser holds at each vertex. This, of course, is nothing a real parser would do. All parsers for large graphs known to the authors perform some kind of pruning.
- There is no restriction to the applicability of rules. The algorithm thus computes the largest number of derivations possible with the least restrictive grammar.

The complexity of this algorithm is $O(|\mathcal{E}||\mathcal{V}|)$. The outermost (line [2]) and innermost (line [5]) loops ensure that each edge is considered and steer the order of evaluation of the edges. The remaining loop (line [4]) increments the number of derivations up to the current vertex, taking into account all shorter derivations of precedent vertices. The sequences can be at most of length $|\mathcal{V}|$, thus the bound on the complexity.

begin

Initialization

[1] totalderiv $\longleftarrow 0$

Handle each vertex

[2] **for** each vertex $v \in \mathcal{V}(G)$, taken in topological order, **do**

Adjust the number of rule applications for this vertex and the total number of derivations so far.

[3] $\text{deriv}_v[1] \longleftarrow \#_{in}(v)$

[4] **for** all $i \in \{2, \ldots, |\mathcal{V}|\}$ do

[5] **for** each edge $e = (w, v, x, y)$ ending in v **do**

[6] $\text{deriv}_v[i] \longleftarrow \text{deriv}_v[i] + \text{deriv}_w[i-1]$

[7] $\text{totalderiv} \longleftarrow \text{totalderiv} + \text{deriv}_v[i]$

totalderiv holds the number of derivations

[8] **return** totalderiv

end

Figure 4.5: Determining the number of derivations in a word graph

4.3.3 Rank of path

The following algorithm determines the rank of a path within a word graph. More exactly, it determines the rank of a path with a given total acoustic score among the scores of all paths in the graph. The input thus consists of a word graph and a reference score s_{ref}.

---

begin

 Compute paths in the graph as given in figure 4.4

 Compute minimum and maximum prefix scores

 for each vertex $v \in \mathcal{V}(G)$, taken in topological order, **do**

  $s_{v,min} \longleftarrow \infty$

  $s_{v,max} \longleftarrow 0$

  for each edge $e = (w, v, s_e, y)$ ending in v **do**

   $s_{v,min} \longleftarrow \min(s_{w,min} + s_e, s_{v,min})$

   $s_{v,max} \longleftarrow \max(s_{w,max} + s_e, s_{v,max})$

 Compute rank of path with score s_{ref} recursively, starting at the final vertex

 $r \longleftarrow$**ComputeRank** (v_f, s_{ref})

 r is the rank of a path with score s_{ref} through the graph

 return r

end

Figure 4.6: Determining the rank of a given path

The algorithm operates by stripping edges from the end of a path and comparing maximal and minimal scores at vertices. This should be superior to simply performing a best-first search based on acoustic scores in order to enumerate the number of paths until the desired is found.

Unfortunately, the algorithm to find the rank of a path with a given score is exponential in the number of vertices in the worst case. This is because the whole graph may be enumerated path by path.

Consider a graph with $|\mathcal{V}|$ vertices and $n(|\mathcal{V}| - 1)$ edges for some odd n (An

function ComputeRank (v, s_{ref})

If the reference score is lower or equal than the minimum score, all paths left from here are worse and need not be considered

if $s_{ref} \leq s_{v,min}$

  return 0

If the reference score is greater or equal than the maximum score, all paths left from here are better and need not be considered

if $s_{ref} \geq s_{v,max}$

  return $p^{(v)}$

Now comes the hard part. Recursively consider all vertices incident to v

$r_{tmp} \longleftarrow 0$

for each edge $e = (w, v, s_e, y)$ ending in v **do**

  $r_{tmp} \longleftarrow r_{tmp}$+**ComputeRank** $(w, s_{ref} - s_e)$

return r_{tmp}

end

Figure 4.7: Determining the rank of a given path (part 2)

example of such a graph with 4 vertices and $n = 5$ is given in figure 4.8). The edges are evenly distributed between the vertices, n edges between each neighbouring pair. Let the vertices be numbered from 1 to $|\mathcal{V}|$ according to the topological position. The weights of the edges between vertex i and vertex $i+1$ are assigned as follows:

- The edge belonging to the reference path has the median weight, say 0.
- Half of the remaining edges have weight $\frac{1}{2^i}$.
- The remaining edges have weight $-\frac{1}{2^i}$.

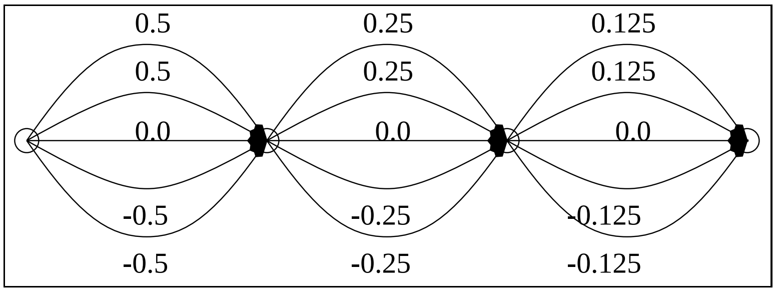

Figure 4.8: A hard graph for rank computation

Now consider the function **ComputeRank()** shown in figure 4.7. It is first called at the final vertex of the graph with a reference score of 0. Since this is neither greater than the maximum score nor lower than the minimum score, n subsequent calls to **ComputeRank()** are issued. As no partial path ending at the final vertex can have a total score with an absolute value larger than any but one edge ending at the start vertex of this particular partial path, it follows that with every call to **ComputeRank()** n additional calls are generated, save the root vertex.

In total, this yields

$$\prod_{i=1}^{|\mathcal{V}|-1} n = n^{|\mathcal{V}|-1} \tag{4.3}$$

calls, which results in a worst case complexity of $\Theta(n^{|\mathcal{V}|})$.

4.3.4 Number of derivations until best path is found

```
begin
          Perform best first search, thereby marking traversed edges
[1]       Push v_s, nil, nil, 0
[2]       while reference path not found do
[3]            Pop v, p, e, s with best score s
               Check success
[4]            if v = v_f ∧ p = p_ref do
[5]                 Reference path is found, end loop
[6]            else do
[7]                 Mark edge e as processed
                    Construct new states
[8]                 for all edges e_vw = (v, w, s_vw, l)incident from v do
[9]                      Push w, p ∪ e_vw, e_vw, s + s_vw

[10]      Remove all unmarked edges from graph
[11]      return the number of derivations from figure 4.5
end
```

Figure 4.9: Determining the number of derivations until the best matching path is found

Chapter 5

Operations on word graphs

This chapter focuses on algorithms for modifications of word graphs. We distinguish between operations on word graphs that preserve all the information in the input graph and operations that may lead to an information loss.

To qualify as being information-preserving, an operation has to fulfil the following criteria:

- All paths in the input graph must be contained in the output graph, with one exception: When the input graph contains any two paths that represent exactly the same sequence of words, the one with the lower (i.e. worse) score may be deleted.
- There must be no path in the output graph that was not in the input one.
- The quotient of the scores of any two complete paths through the graph must stay the same.

Information-preserving operations might for instance remove duplicate paths from the graph.

5.1 Information-preserving operations

5.1.1 Score normalisation

The acoustic score a HMM-based word recognizer might attach to an edge as the edge weight ($w^{(i)} \in \mathcal{W}$) might have different meanings, e.g. probability of the acoustic signal given the corresponding label ($l^{(i)} \in \mathcal{L}$) or probability per frame of it, etc. In this section, we want to define an edge weight measure, that allows (as far as possible) to compare the likelihood of edges ($e^{(i)} \in \mathcal{E}$) or edge sequences ($E_{(t^{(i)},t^{(j)})}$) that cover different intervals ($[i;j]$) of the observation sequence ($O_{[i;j]}$).

To achieve such a comparability, the acoustic scores have to be normalised with respect to the properties of the acoustic signal, i.e. the corresponding observation interval. We do this by dividing the acoustic score by an approximation of the probability of the observation sequence. Other attemts to renormalize scores (e.g. Young (1994)) usually require specific changes in the acoustic recognizer. Our recognizer, for instance, uses a global triphone model, trained on all phones, to recognize out-of-vocabulary words. The score this model assigns to an observation sequence could be used for the normalisation step.

However, we want to define a normalisation procedure that is applicable to any word lattice and does not rely on some specific model.

In all further discussions, we will refer to the resulting measure ($m(e_{(x,y)})$), an approximation of the mutual information between the label and the corresponding observation sequence, as "score".

Using Bayes formula, the probability that a specific model (λ) was the source of a sequence of observations O can be calculated as follows:

$$P(\lambda|O) = \frac{P(O|\lambda) \cdot P(\lambda)}{P(O)}$$

For classification purposes, the quantity $P(O)$ is usually ignored, since for all hypotheses that cover the same O, the probability of it is obviously the same. However, when hypotheses covering different observation sequences are to be compared (to sort an agenda of a probablisitic chart parser, for instance) or the cost function of a classification error is not symmetric (i.e., the exact a-posteriori probability influences the classification decision), an estimate of $P(O)$ is needed.

In theory, it could be calculated using:

$$P(O) = \sum_{\lambda \in models} P(O|\lambda) \cdot P(\lambda)$$

The remaining question is: How can we extract the information needed to solve the above equations from a given word graph? Obviously, a word graph will not contain the scores for *all* possible ways in which *all* possible models (e.g. sentences in a language) could have produces the given observations. For a relatively big word graph, this problem should not be too severe, since we can assume that all hypotheses not in the word graph would have such a low score that they would not contribute much to the overall sum. More difficult problems with word graph score normalisation include:

conditional model instance probabilities: Edge labels are generally not (HMM) model names – there might be different models that carry the same label (e.g. due to pronunciation variants or context dependent versions of the same model). Even if the model itself was known, the a-priory probability of the model $P(\lambda)$ would not be readily available.

local score normalisation: We don't just want to calculate $P(O_{(t(v^{(r)}),t(v^{(f)}))})$ for the complete observation seqence, since this would only allow to renormalise the acoustic score of full paths (from root vertex $v^{(r)}$ to the final vertex $v^{(f)}$) through the graph. Instead, we want to attach renormalised scores to each edge in the graph separately. In order to normalise the score of an edge that starts at frame x and ends at frame y, we need $P(O_{(x,y)})$, the probability of the observation sequence starting at frame x and ending at frame y. To calculate it, we might need to take into account acoustic scores of edges that cover the interval $[x; y]$ only partially and may cover neighbouring observations as well.

As these considerations indicate, we see no way to calculate the exact value of $P(O_{(x,y)})$ from the information contained in the word graph. Instead, we first try to calculate a normalisation constant (an approximation of $P(o^{(i)})$) for each frame i and then divide each edge probability by the product of the normalisation constants of all frames covered by it, in order to get an approximation of the mutual information (m) between edge and corresponding observation sequence. From this score, the a-posteriori probability of a edge sequence $P(E|O)$ can easily be calculated, possibly using sophisticated language models to calculate $P(E)$.

More formally:

$$
\begin{aligned}
m(e_{(x,y)}) &= \mathrm{ld}(\frac{P(e_{(x,y)}, O_{(x,y)})}{P(e_{(x,y)})P(O_{(x,y)})}) && (5.1)\\
&= \mathrm{ld}(\frac{P(O_{(x,y)}|e_{(x,y)})}{P(O_{(x,y)})}) && (5.2)\\
&\approx \mathrm{ld}(\frac{P(O_{(x,y)}|e_{(x,y)})}{\prod_{i=x}^{y} P(o^{(i)})}) && (5.3)\\
\text{using:} & \\
P(O_{(x,y)}|e_{(x,y)}) &= w(e_{(x,y)}) && (5.4)\\
P(o^{(i)}, e^{(i)}) &\approx \sqrt[\beta(e)-\alpha(e)]{w(e_{(x,y)})P(l(e_{(x,y)}))} && (5.5)\\
P(o^{(i)}) &\approx \sum_{l \in \mathcal{L}} max_{\{e \in \mathcal{E} | l(e)=l \wedge t(\alpha(e)) \leq i < t(\beta(e))\}} P(o^{(i)}|e^{(i)}) && (5.6)\\
P(l(e)) &= \text{a-priori probability according to language model}
\end{aligned}
$$

These formulae imply the following assumptions:

1. $P(O_{(x,y)}) = \prod_{i=x}^{y} P(o^{(i)})$ (independence)
2. $P(O_{(x,y)}, l) = max_{\{e \in \mathcal{E} | l(e)=l \wedge t(\alpha(e))=x \wedge t(\beta(e))=y\}} P(O_{(x,y)}|e)$ (Viterbi)
3. $P(O_{(x,y)}, e_{(x,y)}) = \prod_{i=x}^{y} P(o^{(i)}, e^{(i)})$

It is quite obvious that these assumptions do not really hold. It should be noted, however, that the score of each complete path (from $v^{(r)}$ to $v^{(f)}$) is just multiplied with the same constant factor ($\prod_{i=v^{(r)}}^{v^{(f)}} P(o_i)$). The score normalisation is therefore information-preserving according to the above definition. We think that even though the observation-normalisation used is certainly not "correct", comparing different partial paths will at least be much more meaningful after this normalisation than without any normalisation at all. As a side effect, this method also allows us to deal with (discrete) probabilities, while the original scores attached to labels by the recognizer we are using for our experiments (Huebener, Jost, and Heine, 1996) attaches values of density functions to the output edges.

5.1.2 Operations concerning silence

Silence edges deserve some special treatment since they usually are not supposed to carry any meaning that has to be preserved for subsequent language processing modules after recognition. By agreement each word graph starts and ends with silence, but in principle all other silence edges don't count. They are, for example, not considered as errors during evaluation.

We implemented three different methods to simplify graphs containing silence. The first one deals with vertices connected solely by silence in a straightfoward way. The second one tries to join contiguous silence edges[1], while the third simply removes all silence edges from the graph (even at the start and end of the graph).

Naturally, the algorithms removing silence edges can be applied to other types of edges, too. We have been experimenting with the removal of several small filling or hesitation-marking words (äh or ähm in German). The removal of such words can be assumed advantegeous if they are considered as noise in the signal. On the other hand, they may be important for the detection of repairs or restarts in systems processing spontaneous speech.

Removing single silences

A single silence edge being the only edge between two vertices is a rare condition in graphs delivered by a speech recognizer. However, the number of such cases may rise if several other operations are applied to a word graph, such as those described later. The simple idea behind the algorithm shown in figure 5.1 is to join two vertices if a silence edge is the only one between the two and they are otherwise mutually not reachable. In some versions we added a further time constraint. As each vertex is assigned a point in time, we demanded that no edge leaving the earlier vertex may end at a vertex that lies before the other. This was done to ensure that the ordering of edges and vertices in time is preserved. In general, however, this constraint is not mandatory and not shown in figure 5.1. Note that the scores of prolongated edges have to be incremented by the score of the silence edge to preserve the informational content of the graph.

The function to merge two vertices is described seperately, as we will use this function in the algorithms to come, too. Figure 5.2 describes how to merge two vertices, v and w, by copying all edges at w to v. Afterwards, w is normally

[1]Since this algorithm is not information preserving, it is described below in section 5.2.1.

begin
[1] **for** each vertex $v \in \mathcal{V}$, taken in topological order, **do**
Search for silences leaving v
[2] **for** each edge $e = (v, w, s, < SIL >)$ **do**
[3] **if** $v \not\xrightarrow{*} w$, disregarding e, **then**
Adjust scores
[4] **for** each edge $e' = (v, w', s', l'), e \neq e$, **do**
[5] $s' \longleftarrow s' + s$
Merge vertices v *and* w
[6] **Merge**(v, w)
[7] Delete w
end

Figure 5.1: Removing single silence edges

deleted. To reduce overhead, new edges are only constructed if there is no identically labelled edge already present. In that cases, scores are simply updated.

Removing all silence edges

The most effective method to reduce the number of paths in a graph (if not the number of edges) only by operating on silence edges is to remove them all. This can easily be done by copying all edges at the endpoint of a silence edge.[2] The newly introduced edges start at the startpoint of the silence, thereby in general generating much more edges. However, in many cases an identically labelled edge has already been present. This fact reduces the growth in the number of edges. Figure 5.3 describes the algorithm. This algorithm operates information-preserving, provided we assume that silence edges carry no meaning, i.e. paths are considered equal that only differ in the distribution of silences among them.

[2]Silence edges that end at the final vertex of a graph have to be preserved, thus not generating multiple end vertices.

```
procedure Merge( v, w)
        Merge two vertices v and w
        First, copy w's ingoing edges
        for each edge e = (x, w, s, l), do
            if ∃e' = (x, v, s', l), then
                s' ⟵ min(s, s')
            else
                Create a new edge e'' = (x, v, s, l)

        Now, copy w's outgoing edges
        for each edge e = (w, x, s, l), do
            if ∃e' = (v, x, s', l), then
                s' ⟵ min(s, s')
            else
                Create a new edge e'' = (v, x, s, l)
end
```

Figure 5.2: Merge two vertices

```
begin
        for each vertex v ∈ 𝒱, taken in topological order, do
            for each edge e = (v, w, s, <SIL>), do
                Copy edges from the endvertex
                for each edge e' = (w, x, t, l), do
                    if ∃e'' = (v, x, t', l), then
                        t' ⟵ min(t + s, t')
                    else
                        Create a new edge e'' = (v, x, t + s, l)
                Delete e
end
```

Figure 5.3: Removing all silence edges

5.1.3 Reducing paths in a word graph

Constraining to unique word sequences

As mentioned earlier, a parser may not be interested in more than one path through a graph covering a given word sequence. The algorithm shown in figure 5.4 modifies a word graph to ensure just that. It guarantees that no vertex is ever left by two edges with identical labels. This local condition has the effect that from a global point of view no two distinct paths through the graph bear identical word sequences.

---

begin
[1] **for** each vertex $v \in \mathcal{V}(G)$, taken in topological order, **do**
[2] **for** each pair of identically labeled edges e_1, e_2 **do**
Perform edge merging and create new vertex
[3] Create a new vertex v having
$t(v) := \min(t(\beta(e_1), t(\beta(e_2))))$,
inserting v into the topological order
Copy all edges incident from $\beta(e_1)$ *to* v
[4] **for** each edge $e = (\beta(e_1), w, s, y)$ **do**
[5] Create a new edge $e' := (v, w, s, y)$
Copy all edges incident from $\beta(e_2)$ *to* v
[6] **for** each edge $e = (\beta(e_2), w, s, y)$ **do**
[7] Create a new edge $e' := (v, w, s, y)$
Delete e_1, e_2
end

Figure 5.4: Reducing a graph to unique label sequences

The worst case complexity of this algorithm is exponential in the number of vertices of the graph. To see this, consider a graph as shown in figure 5.5. Every vertex except the last two have two edges incident from them, one incident to the next vertex, one incident to the one after next. Both edges are labelled identically. If we apply the algorithm, a new vertex has to be created for each pair of such edges. The new vertex has twice as many incident from it as the original vertex. More generally, if we assume d pairwise identically labelled edges incident from

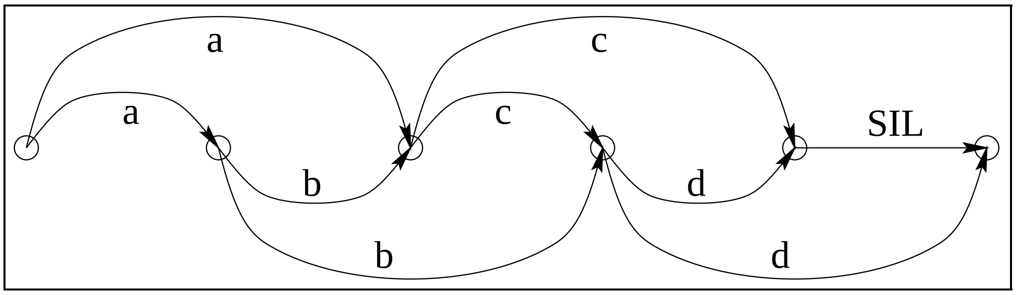

Figure 5.5: A hard graph concerning unique paths

each vertex, we have $2\,|\mathcal{V}| - 1$ edges in the graph. Let us follow a chain of vertex creations and application on the new vertices. For the first vertex, $\frac{d}{2}$ new vertices are created, each of them equipped with $2\,d$ edges incident from it. Operating on one of the new vertices again yields $\frac{d}{2}$ new vertices, now with $4\,d$ edges. We can repeat this cycle $\frac{|\mathcal{V}|}{2} - 1$ times, as we advance two vertices with every step. During the cycle, we create

$$\sum_{i=1}^{\frac{|\mathcal{V}|}{2}-1} 2^i\,d \tag{5.7}$$

new edges, which all have to be considered by the algorithm. This yields at least

$$d(2^{\frac{|\mathcal{V}|}{2}} - 2) \tag{5.8}$$

operations, thus the complexity of our algorithm is $\Theta(\sqrt{2^{|\mathcal{V}|}})$.

The algorithm proposed here has not been applied to any real-world word graphs. This is due to the large number of vertices and edges that are created anew. In section 5.2.2 below we will present a version of this algorithm wich does not preserve information, but is faster and has been used to a larger extent.

5.2 Other operations

5.2.1 Joining contiguous silences

Two paths only differing in the number of silence edges at a particular place are considered different, yet they might not change word recognition rates — silence is usually ignored during evaluation and in general considered worthless during linguistic analysis[3]. Hence, the number of paths can be reduced by concatenating multiple silence edges.

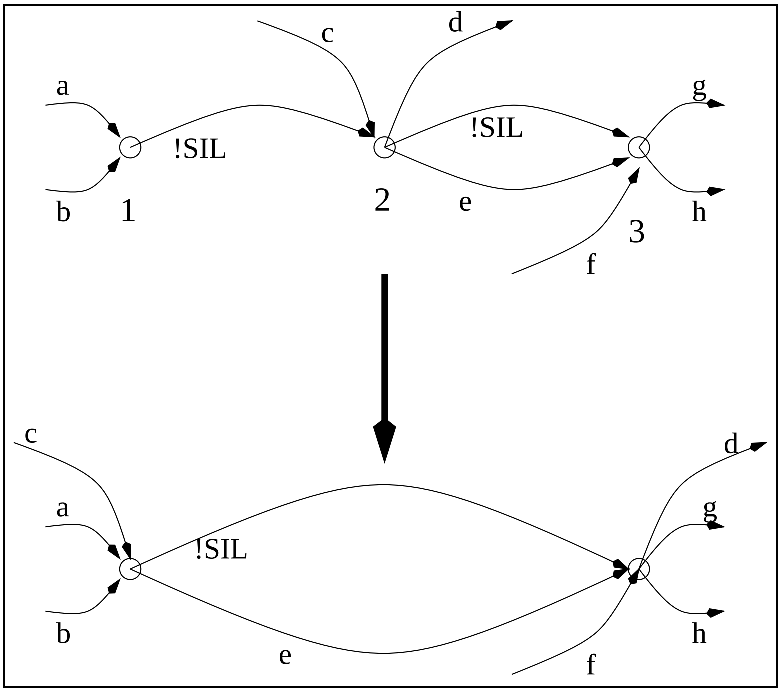

Figure 5.6: Joining two consecutive silence edges

The easy case arises if three vertices are only connected by silence edges. This case can be handled by the algorithm described in figure 5.1. A more complex example is shown in figure 5.6. The difficult part is to decide where to attach edges c and d. If the edges incident to vertex 2 are attached to vertex 1, and the edges incident from vertex 2 attached to vertex 3 (as is shown in the figure) new

[3] Pauses in the speech signal may well be used to constrain the application of certain grammar rules, e.g. in the case of multi-utterance grammars (Kasper and Krieger, 1996). In those cases, however, one should use a dedicated recognizer for prosodic events like (Strom and Widera, 1996).

paths are introduced in the graph (in this case the edge sequence $c - e - d$). If, on the other hand, d is attached to 1 and c to 3, sequences vanish.[4]

The algorithm ist straightforward and defined in figure 5.7. This algorithm is not information-preserving.

5.2.2 Unique word sequences

The algorithm to constrain a word graph to unique label sequences outlined above is correct and preserves the informational content of a word graph, but it can not be efficiently applied to larger graphs. The main augmentation that has a positive effect on run time is to merge vertices of the graph.[5] Vertices should be merged whenever possible to reduce the number of edges in the graph.

Merging of vertices is carried out by copying the edges from one vertex, which is afterwards deleted, to the other (cf. figure 5.2). Edges that bear labels already present at the remaining vertex are not copied, rather the score is possibly updated to reflect the better score of both edges. Naturally, in order to avoid the introduction of cycles into the word graph, only vertices that are mutually not reachable may be merged.

Merging, however, is severly restricted if we constrain ourselves to information-preserving operations. Two vertices can only be merged under the conditions that

- they have the same set of vertices incident to and from them,
- the edges incident to and from them are identically labelled, and
- the edges incident to and from them have identical scores.

These conditions are hardly ever met. To allow for further merging, we departed from an information preserving approach and loosened the constraints somewhat. Some conditions for merging could be:

- Two vertices can be merged if they have the same set of vertices incident to and incident from them. This adds slightly to the number of paths, since there may be new partial paths of length 2 introduced by the merging.

[4] We don't discuss the two remaining possibilities.

[5] Even using the augmentations the algorithm is still exponential in the worst case.

begin
[1] **for** each vertex $v \in \mathcal{V}$, taken in topological order, **do**
Search for two silences
[2] **if** $\exists e_1 = (v_1, v, s, < SIL >), e_2 = (v, v_2, s', < SIL >) \in \mathcal{E}$, **then**

Check applicability
[3] **for** each edge $e = (w, v, x, y) \in \mathcal{E}$, **do**
[4] **if** e falls into the range $[t(v_1), t(v_2)]$, **then**
[5] **continue** with loop in line **[2]**

Move edges incident to v along
[6] **for** each edge $e = (v', v, x, y) \in \mathcal{E}, e \neq e_1$, **do**
[7] **if** $v' = v_1$, **then**
[8] Create new edge $e' = (v_1, v_2, x + s', y)$
[9] **else**
[10] Create new edge $e' = (v', v_1, x - s, y)$

Move edges incident from v along
[11] **for** each edge $e = (v, v', x, y) \in \mathcal{E}, e \neq e_1$, **do**
[12] **if** $v' = v_2$, **then**
[13] Create new edge $e' = (v_1, v_2, x + s, y)$
[14] **else**
[15] Create new edge $e' = (v_2, v', x - s', y)$

[16] Delete v and all incident edges
end

Figure 5.7: Algorithm to join consecutive silence edges

- Two vertices can be merged if they have the same set of vertices incident from them.
- Two vertices can be merged if they belong to the same point in time. This situation can arise given two conditions: First, a recognizer may create more than one vertex for some frame (e.g. to preserve the shape of the search space within the recognizer). Second, the creation of new vertices during the algorithm might lead to vertices belonging to the same point in time.

The merging of vertices is in general carried out each time a new vertex is considered. Additionally, after creating new vertices due to edges with identical labels, one might want to merge all those vertices. They are guaranteed to have the same set of vertices incident to them (exactly one) and are primary candidates for merging.

Using merging of vertices with identical vertices left and right and merging of newly created vertices (but only in case they have identically vertices incident from them), we were able to create reduced graphs in a reasonable amount of time. The run time is shown in figures 5.8 and 5.9 below.

5.2.3 Reducing families

As explained in chapter 2, word graphs usually contain a large number of families, i.e. overlapping edges with a common label. This is a reflection of the fact that word graphs usually contain many paths that represent exactly the same label sequence with slightly different segmentations. Since the exact segmentation of the speech signal is often not really important for word graph processing modules, one might want to reduce graphs so as to contain unique label sequences only. An algorithm to do this is described in sections 5.1.3. Unfortunately, the complexity of this algorithm is exponential.

Furthermore, even a graph that contains unique label sequences only might contain a high proportion of families, namely when family members have different neighbours. A more radical reduction in the number (or size) of families can be reached when the information-preserving requirement of the algorithm is given up and additional paths can be inserted into the graph.

In this section we describe a very fast ($n \log n$ in the number of edges) algorithm,

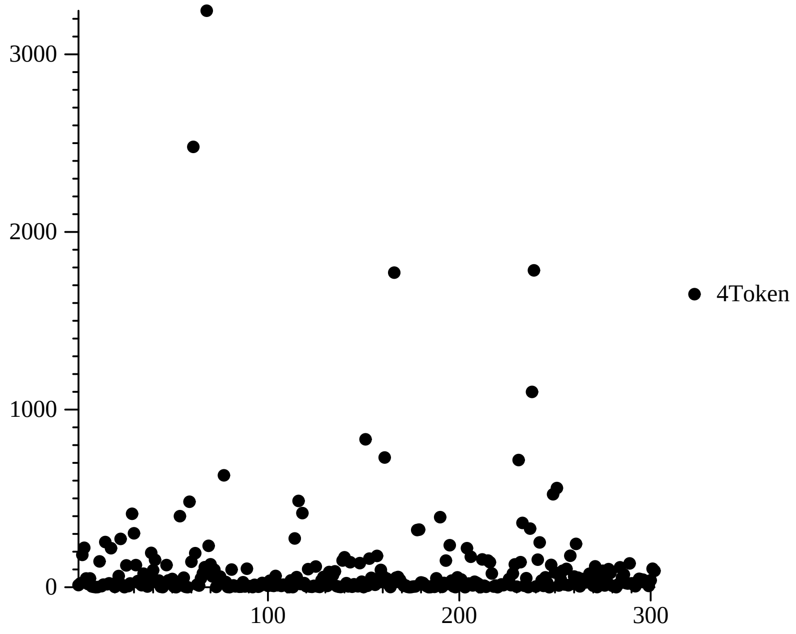

Figure 5.8: Runtime used for unique word sequences (x-axis: turn number, y-axis: runtime in seconds)

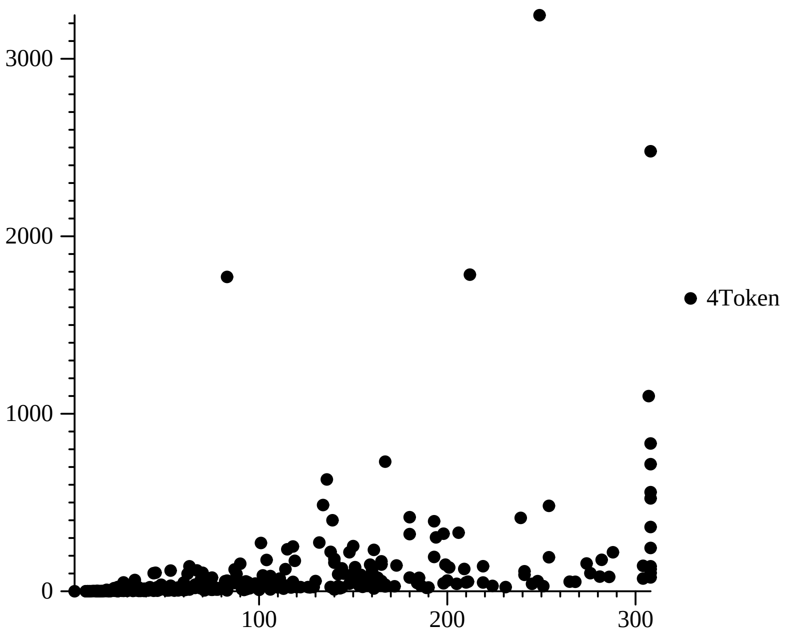

Figure 5.9: Runtime used for unique word sequences (x-axis: log(number of paths), y-axis: runtime in seconds)

that reduces the number of families drastically by joining vertices to "meta vertices" (u).

---

begin

Initialization

[1] $\mathcal{U} = \{u^{(0)}\}$

[2] $\alpha(u^{(0)}) = v^{(r)}$

[3] $\beta(u^{(0)}) = v^{(f)}$

find meta vertices

[4] **for** each edge $e \in \mathcal{E}$, sorted according to increasing length **do**

[5]   **if** $u(\alpha(e)) == u(\beta(e))$

[6]   split $u(\alpha(e))$ at $\frac{t(\alpha(e))+t(v(e))}{2}$

split meta vertices if they are longer than permitted

replace vertex by meta vertex

[7] **for** each edge $e \in \mathcal{E}$ **do**

    $\alpha(e) = u(\alpha(e))$; $\beta(e) = u(\beta(e))$

end

Figure 5.10: Merge vertices

To get some impression of the impact of the algorithm, compare figures 5.11 and 5.12. Note that the number of edges and vertices was reduced from 141 edges and 37 vertices to 91 edges and 22 vertices. At the same time, the number of paths increased from 24409 to 87742 and the nuber of unique paths from 8221 to 34554.

The original graph contains many very short edges with the label "#PAUSE#". If these edges are ignored during the meta-node splitting process, the reduced graph contains only 25 edges, 8 vertices and 1098 paths, that are all unique. The resulting graph is plotted in figure 5.13.

To compare the results of reducing a graph to unique label sequences and of reducing families using the algorithm just described, consider figures 5.14 and 5.15. To achieve better readability, all "#PAUSE#"-edges except the ones starting at the root vertex were removed (see section 5.1.2 for the algorithm). The first graph

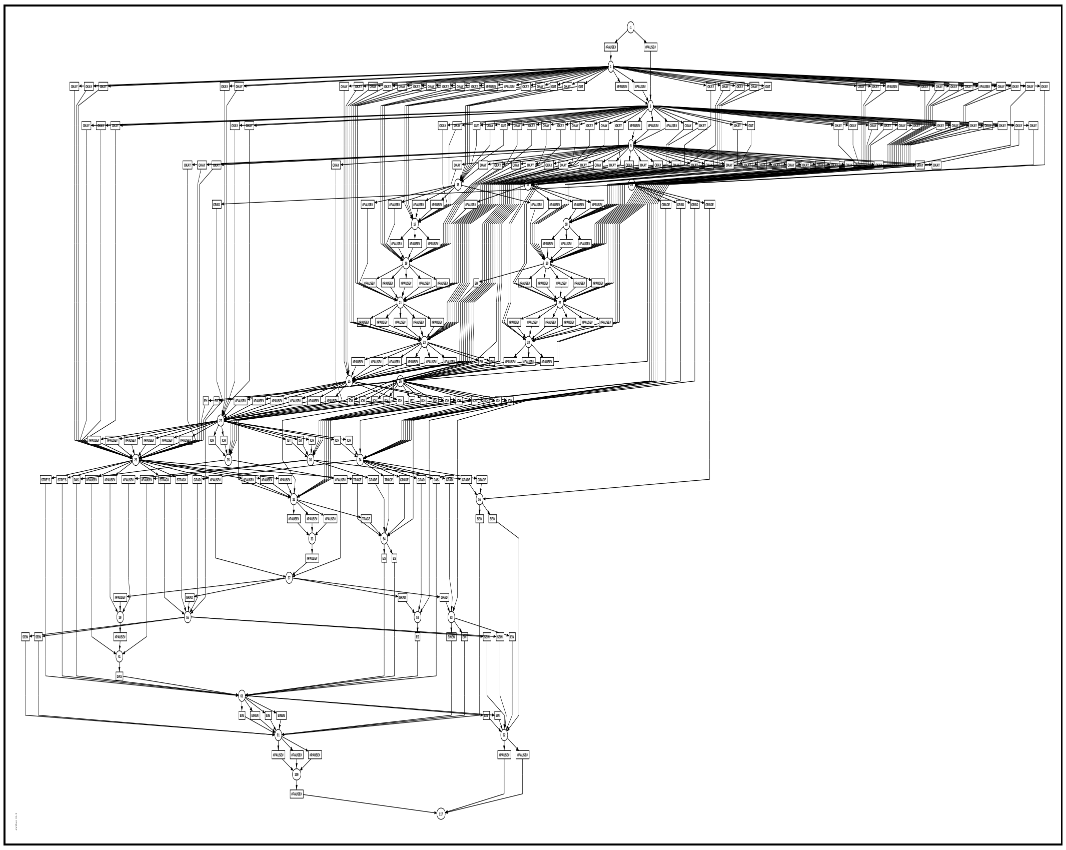

Figure 5.11: Graph for reference utterance: "ich trage es ein"

contains 47 edges, 24 vertices and 300 paths while the second one contains 31 edges, 9 vertices and 756 paths.

As mentioned above, previously unconnected partial paths will be connected by this algorithm. On the one hand, this is a desired effect that can for instance help to deal with the problem of coarticulation (see chapter 2). On the other hand, one could say: "What an HMM has divided, man should not join." In other words: Graph manipulations should ideally compensate for deficiencies in the acoustic and lexical modelling only and otherwise retain all the information that could be extracted from the acoustical signal. Therefore it is advisable to restrict the maximal length of meta-nodes and to punish the joining of paths according to same distance measure between the original nodes in further processing steps.

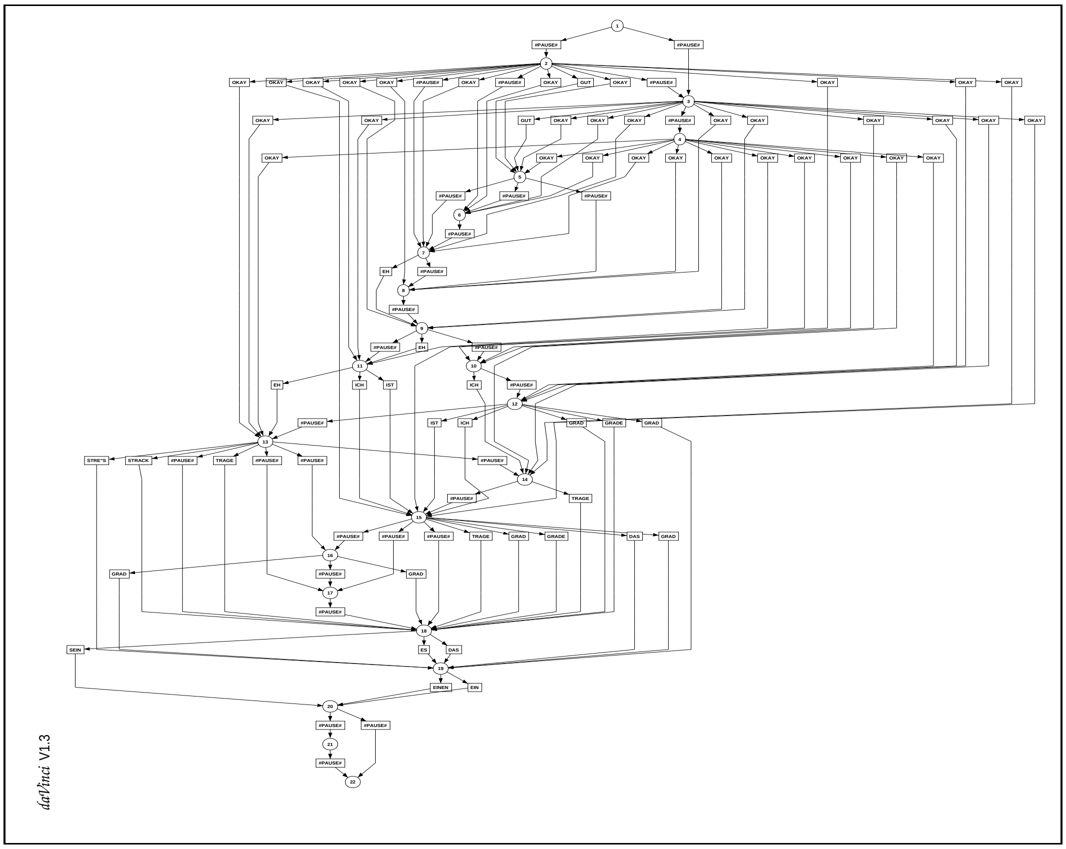

Figure 5.12: Reduced graph for reference utterance: “ich trage es ein”

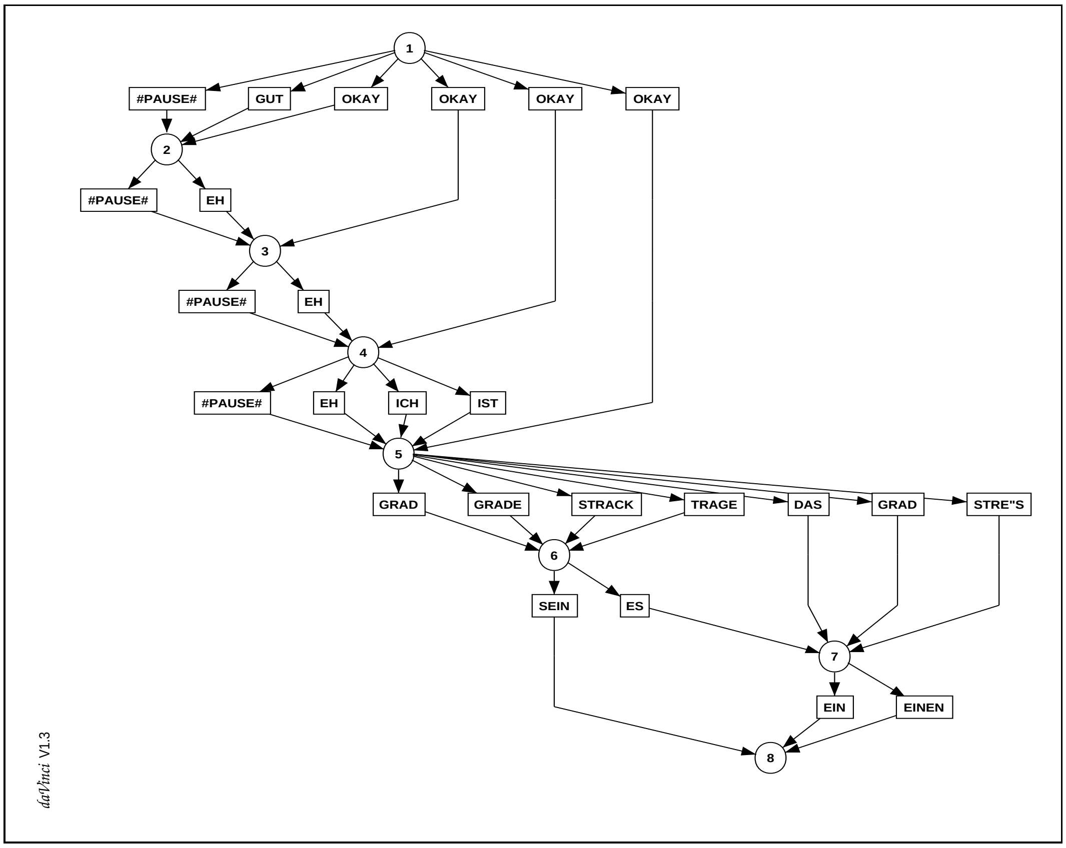

Figure 5.13: "#PAUSE#"-reduced graph for reference utterance: "ich trage es ein"

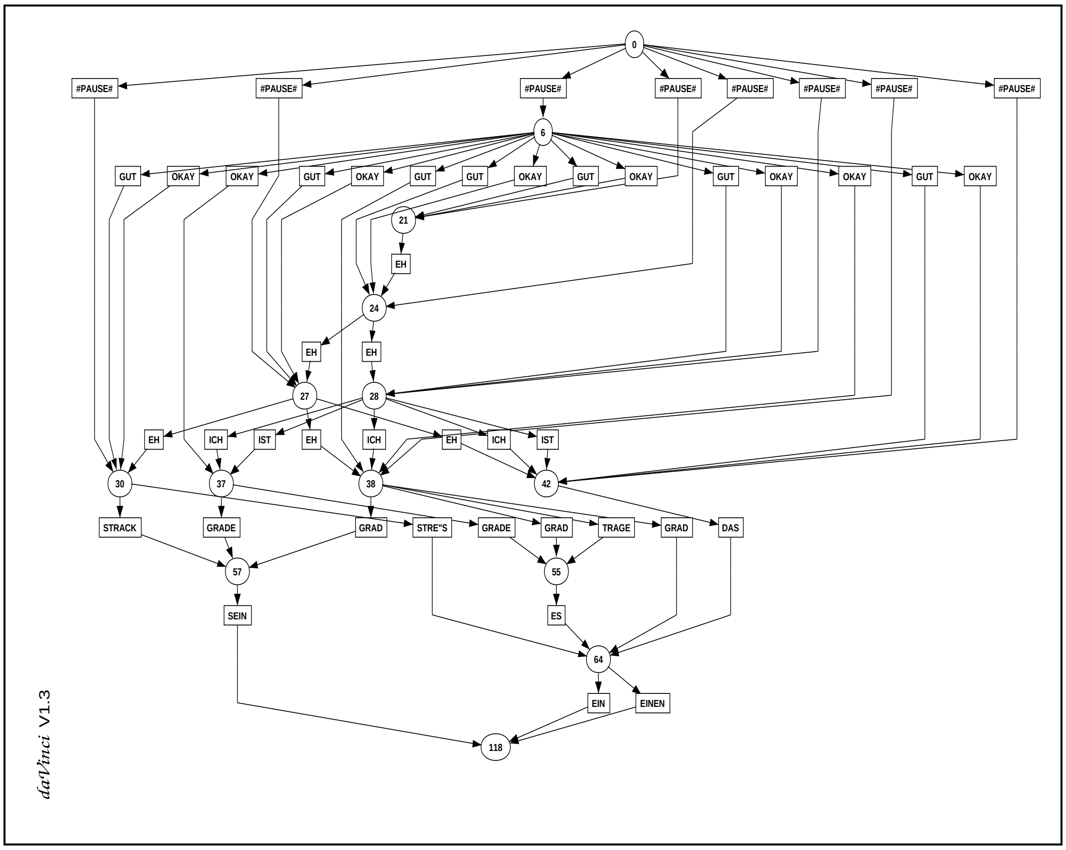

Figure 5.14: Graph for reference utterance: "ich trage es ein" containing unique hypotheses only

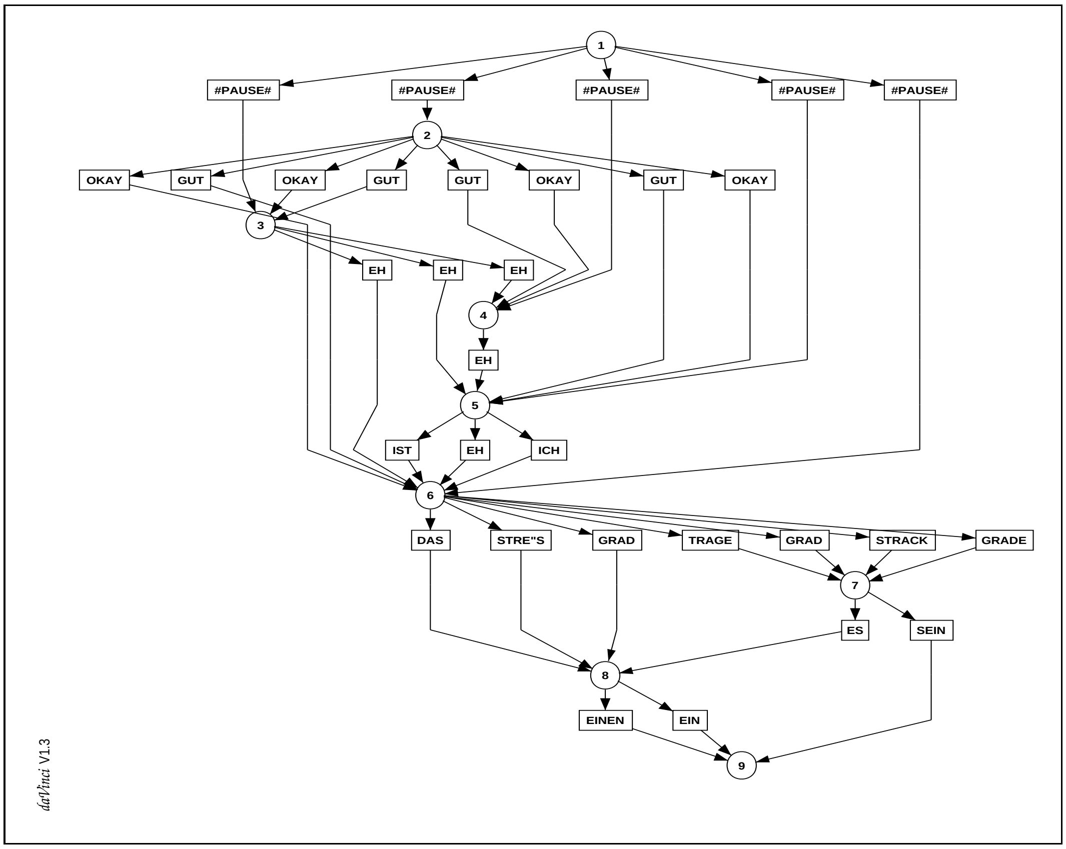

Figure 5.15: Family-reduced graph for reference utterance: "ich trage es ein"

Chapter 6

Conclusion

In this paper, we have proposed a new measure for the problem size represented by word graphs. Starting from conventional evaluation of best-chain recognizers, we argued that a straightforward generalization of well known procedures to graph recognizers may be misleading. While word accuracy can be extended to graph evaluation, a sensible notion of the size of a problem is much harder to find. Measures taken until now, like the number of word hypotheses per reference word or the average fan out of vertices, are insufficient in our view, since the topology and shape of the word graphs are not properly reflected by these measures.

Instead, we propose to choose the amount of processing a hypothetical parser would have to carry out in order to process a graph as the principal measure for graph size. This measure takes account of the number of edges as well as the number of paths, and simultaneously the shape of a word graph is considered. The assumption taken here was that a large amount of subgraph sharing reduces the burden placed on a language processing component considerably. We motivated this measure and gave efficient algorithms to compute it.

The second part of this report was dedicated to algorithms modifying word graphs with different degrees of severity. We introduced the notion of "harmless" (i.e., information-preserving) operations and presented some algorithms concerning the reduction of silence edges and the construction of graphs containing only uniquely labelled edge sequences. Since the run times were sometimes dissatisfying, we showed versions of that algorithm that do not preserve information content, but in general need only a fraction of the processing time. We feel that even while they introduce new paths and modify paths scores, they do not too much "harm" to a graph. Finally, we presented a very fast algorithm that in essence reorganizes a word graph to a large degree, but has several favoring properties, among them that it results in extremly small, compact graphs and partly copes with the problem of coarticulation.

Chapter 7

Bibliography

Aubert, Xavier and Hermann Ney. 1995. Large vocabulary continuous speech recognition using word graphs. In *ICASSP 95*.

Carreé. 1979. *Graphs and networks*. Oxford applied mathematics and computing science series. Oxford: Clarendon Press.

Cormen, Thomas H, Charles E. Leiserson, and Roanld L. Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Gondran, Michel and Michel Minoux. 1984. *Graphs and alogorithms*. Wiley-interscience series in discrete mathematics. Chichester: John Wiley & sons.

Huebener, Kai, Uwe Jost, and Henrik Heine. 1996. Speech recognition for spontaneously spoken German dialogs. In *ICSLP96*.

Kasper, Walter and Hans-Ulrich Krieger. 1996. integration of Prosodic and Grammatical Information in the Analysis of Dialogs. In *KI-96: Advances in Artificial Intelligence. 20th Annual German Conference on Artificial Intelligence*, pages 163–174, Berlin, September. Springer.

Kipps, J.R. 1991. GLR Parsing in time $O(n^3)$. In M. Tomita, editor, *Generalized LR Parsing*. Kluwer, Boston, pages 43–59.

Lehning, Michael. 1996. Evaluierung von signalnahen Spracherkennungssystemen fuer deutsche Spontansprache. Verbmobil Report 161, TU Braunschweig, www.dfki.uni-sb.de/verbmobil.

Mayer, Otto. 1986. *Syntaxanalyse*. Reihe Informatik, number 27. Mannheim: Bibliographisches Institut.

McHugh, James A. 1990. *Algorithmic Graph Theory*. Englewood Cliffs, NJ: Prentice Hall.

Oerder, Martin and Hermann Ney. 1993. Word graphs: an efficient interface between continuous-speech recognition and language understanding. In *ICASSP93*.

Paulus, Erwin and Michael Lehning. 1995. Die Evaluierung von Spracherkennungssystemen in Deutschland. Verbmobil Report 70, TU Braunschweig, www.dfki.uni-sb.de/verbmobil.

Reinecke, Joerg. 1996. Evaluierung der signalnahen Spracherkennung. Verbmobil Memo 113, TU Braunschweig.

Strom, Volker and G. Widera. 1996. What's in the "pure" Prosody? In *Proc. ICSLP 1996*, Philadelphia.

Tran, B.H., F. Seide, and V. Steinbiss. 1996. A word graph based n-best search in continuous speech recognition. In *ICSLP*.

Weber, Hans. 1995. *LR-inkrementelles. probabilistisches Chartparsing von Worthypothesengraphen mit Unifikationsgrammatiken: Eine enge Kopplung von Suche und Analyse*. Ph.D. thesis, Universität Hamburg.

Woodland, P.C., C.J. Leggetter, J.J. Odell, V. Valtchev, and S.J. Young. 1995. The 1994 HTK large vocabulary speech recognition system. In *ICASSP95*.

Young, Sheryl R. 1994. Detecting misrecognitions and out-of-vocabulary words. In *ICASSP95*.