N-Best Search Methods Applied to Speech Recognition

Diploma Thesis
(Diplomarbeit)

Heiko Purnhagen

Universitetet i Trondheim
Norges Tekniske Høgskole
Institutt for Teleteknikk

Universität Hannover
Institut für Theoretische Nachrichtentechnik
und Informationsverarbeitung

Trondheim, May 1994

N-Best Search Methods Applied to Speech Recognition

Diploma Thesis
(Diplomarbeit)

Heiko Purnhagen

Trondheim, May 1994

UNIVERSITETET I TRONDHEIM
NORGES TEKNISKE HØGSKOLE
FAKULTET FOR ELEKTRO- OG DATATEKNIKK

H O V E D O P P G A V E

This copy of the thesis description is included in this document for completeness only. It is not the original!

Kandidatens navn: Heiko Purnhagen

Fag: Teleteknikk

Oppgavens tittel (engelsk): **N-best search methods applied to speech recognition**

Oppgavens tekst:

Automatic recognition of speech has come a long way from the first serious attempts at machine recognition of a few isolated words in the 1950's. Today, commercial recognizers capable of recognizing several tens of thousands words spoken as isolated utterances are available on a PC platform and the first speaker independent recognizers are being deployed in the telephone networks.

The main emphasis of current research in automatic speech recognition is on speaker independent recognition of large vocabulary continuous speech. A number of issues are vital to the success of the present challenge. For instance, the feature extraction must be robust to speaker variations, yet sensitive to changes in phonetic content. Also, detailed modeling of the natural acoustic phonetic variations is essential to the accuracy in recognition.

Most successful systems are based on Hidden Markov models. This powerful probabilistic model has proved to be efficient as well as versatile and has the nice capability that the speech process can be modeled at several levels using the same type of building blocks. This simplifies the incorporation of grammar in the speech recognizers.

The Viterbi algorithm has been widely used when recognizing continuous speech. The algorithm is efficient but has the drawback that only the best path (i.e. word string) is found. This gives rise to sub-optimal performance in cases when e.g. a single word is erroneously recognized, but where a simple semantic or syntactic analysis could correct the misrecognized word.

Recently, methods for efficiently finding the N best paths have been proposed. In addition to simplifying the incorporation of syntactic and semantic information in a speech recognizer, these methods also give rise to other interesting possibilities.

In an HMM-based speech recognizer, all parameters specifying the recognizer are found through careful optimization of an object function. There is only one exception. The

lexicon used in a sub-word based recognizer for building word and sentence models from the basic recognition units are normally generated by the use of a pronouncing dictionary or by an experienced phonetician. It has been suggested [1] that a modified N-best algorithm can be used to generated a lexicon that is optimal with respect to the likelihood of the training data.

The thesis should consist of the following components:

1) Study several proposed N-best algorithms with respect to optimality and efficiency of implementation. Simulate one of the methods in the context of a continuous speech recognizer.

2) Modify the selected N-best algorithm to provide a basis for automatically generating optimal lexical entries for a sub-word based continuous speech recognizer. Study the generated lexicon with respect to performance and to their relation to standard baseforms as provided by pronouncing dictionary.

Supervisor: Associate professor Torbjørn Svendsen, NTH

[1] T.Svendsen: "Optimal acoustic baseforms", Internal technical memo, 1991

Oppgaven gitt: 1. desember 1993

Besvarelsen leveres innen: 31. mai 1994

Utført ved: NTH

Trondheim, den 25. mai 1994

Torbjørn Svendsen
faglærer

Acknowledgements

I would like to thank all the people who enabled me to do my diploma thesis here at the Norwegian Institute of Technology (NTH) in Trondheim. First of all, I would like to thank my supervisor at the NTH, Mr. Svendsen, who immediately was willing to offer me a thesis work when I got in contact with him for the first time in summer '92. He always found time to discuss my questions. I would also like to thank Mr. Musmann and Mr. Edler at the University of Hannover who made it possible for me to do this diploma thesis externally.

I am grateful to the people from the *Studienstiftung des deutschen Volkes* who made me start thinking about a diploma thesis abroad and who supported my stay in Norway ideally and financially. I am also grateful to Mr. Laschet at NTH's International Department who gave me the same assistance as if I would have come here within a student exchange program.

I also would like to thank my parents who encouraged and supported my plans for this stay in Norway from the beginning. Finally, I would like to thank all the other people who made this stay an experience I will never want to have missed.

I, the undersigned, certify that this diploma thesis is entirely my own work and that all the sources of information used in it are acknowledged.

Trondheim, May 1994

Contents

Chapter 1

Introduction

The research on automatic recognition of human speech by machines started more than four decades ago. Systems capable of recognising up to several thousand words spoken as isolated utterances are already commercially available. The speaker independent recognition of continuous speech is much more complex and represents a main field of todays research in speech recognition [1]. Most of the successful systems are based on Hidden Markov Models (HMMs). These statistical models have shown to be powerful and versatile and allow to model the speech process on different levels using the same concept [3]. Thus, also language models like a grammar can be easily included in the recogniser.

When developing a speech recognition system, it is normally not sufficient to provide only models of the words that are to be recognised. To obtain reasonable performance, it is common to include additional knowledge sources in the recognition process. These can be models of the natural language or also knowledge about the task the recognition system will be used for. If all knowledge sources are simultaneously included, the search for the most likely word string (hypothesis) for a spoken utterance might become very complex. Recently, techniques that allow to apply the different knowledge sources sequentially and thus reduce computation have been proposed [7]. These techniques are based on the N-best search paradigm. The Viterbi algorithm, which is normally used in HMM based recognition systems, is only able to find the best hypothesis. Now, an N-best algorithm generating the list of the N most likely hypotheses is required. These N-best hypotheses then can be rescored according to additional knowledge sources. In this way, also complex knowledge sources can easily be included in the recognition process.

Several different N-best algorithms for generating the list of the best N hypotheses have been proposed recently [7, 8, 9]. Some of these algorithms generate an exact list of the N most likely hypotheses while others use different approximations to reduce computation. Besides for the N-best search paradigm, N-best algorithms can also be used for other new applications.

The parameters specifying an HMM based recogniser are found in training procedures that optimise these parameters in order to maximise an object function (e.g. likelihood of a training utterance). An important exception is the lexicon that is required in a subword based recogniser to build word models from the models of basic recognition units. This lexicon contains transcriptions in terms of basic recognition units for each word in the vocabulary and is normally generated by using a pronunciation dictionary or by an experienced phonetician. Different techniques for the automatic generation of entries in

such a lexicon have been proposed [15, 16, 17, 18, 19]. They require training utterances of the word and can also take into account the word's spelling. A modified version of one of the proposed N-best algorithms can be used to find the lexicon entries (transcriptions) that are optimal with respect to their likelihood for the training utterances [15].

The work done in this thesis can be divided into two main parts:

In the first part, different N-best algorithms were studied. They will be described and compared in this report. The tree-trellis algorithm, an exact and efficient N-best algorithm, was implemented as a part of this thesis work. This implementation is based on an existing continuous speech recogniser which is part of the HMM Toolkit (HTK). HTK is a collection of programs that allows to build and test HMM based recognisers in an efficient and flexible way [4]. The implementation, which required several adaptations of the original tree-trellis algorithm, and the optimisations that were done in order to obtain maximum performance, will be described. This new N-best recogniser was tested on the 1000 word DARPA resource management task using different HMMs and different grammars. The test results and the computation and memory requirements will be reported.

In the second part, the automatic generation of lexica for a phoneme based recogniser was studied. The implementation of the tree-trellis algorithm was modified and extended to perform a search for the most likely hypotheses given a set of utterances of the same word. The modifications and the optimisation of this program will be described. The search for a new lexicon entry was in several cases too complex for the available hardware, although memory requirements were minimised. Therefore, different approximative techniques to find a new lexicon entry using a search with reduced complexity were developed and examined. Finally, several lexica were generated using these different techniques. The performance of these lexica and their effect on the results of recognition tests will be presented.

This thesis report is organised as follows: In Chapter 2, a short overview of the fundamentals of speech recognition is given. The concept of Hidden Markov Models is reviewed and the HMM Toolkit (HTK) is described. In Chapter 3, different N-best algorithms are presented and their features are compared. An implementation of the tree-trellis algorithm based on HTK's Viterbi recogniser is described and the results of recognition tests are reported. In Chapter 4, concepts for automatic lexicon generation are presented and the implementation of a modified tree-trellis algorithm for multiple utterances is described. Details and problems of the lexicon generation process are discussed and finally, the performance of different automatically generated lexica is examined. In Chapter 5, this report is summarised and conclusions are drawn. Appendix A contains the user manuals for the different programs that were written as a part of this thesis work.

Chapter 2

Speech Recognition

In the beginning of this chapter, a short overview of the fundamentals of speech recognition is given. Then the concept of *Hidden Markov Models* (HMM) is reviewed. HMMs allow powerful and flexible modeling of human speech and represent the basis for the speech recognition methods used in this thesis. Finally the *Hidden Markov Model Toolkit* (HTK) is described. The HTK is a collection of C programs that allows the quick and efficient design of an HMM based speech recogniser.

2.1 Fundamentals of Speech Recognition

First systems for the automatic recognition of spoken speech by machines were developed more the four decades ago. Since then automatic speech recognition has become a big field in research. But the desired goal of a machine that can understand spoken discourse on any subject by all speakers in all environments is still far from being achieved. A good description of the "state of the art" in speech recognition is given by [1]. Some fundamentals of speech recognition will be summarised now.

2.1.1 The Speech Signal

Spoken speech is a means of communication between human beings. The speech production / speech perception process which enables a listener to understand the message spoken by a talker is illustrated in Figure 2.1.

When speaking an utterance, a sequence of different sounds is generated in the speaker's vocal tract [1, 2]. First, the stream of air from the lungs is converted into an excitation for the system of cavities that make up the vocal tract. This excitation can be quasi-periodic (vibrating vocal cords), noise-like (turbulent flow trough a constriction) or transient (opening a total closure in the vocal tract). The resonating cavities in the vocal tract act as a filter and modify the spectral shape of the excitation. By moving tongue, lips etc., the size and shape of these cavities can be changed and different sounds can be articulated. In the English language, there are about 40 to 50 linguistically distinct speech sounds, so-called *phonemes*. The set of phonemes used later in this thesis is shown in Table 2.1. Most speech sounds can be characterised by certain spectral properties. Vowles for example can be characterised by the frequencies of their 2, 3 or 4 biggest resonances, their *format*

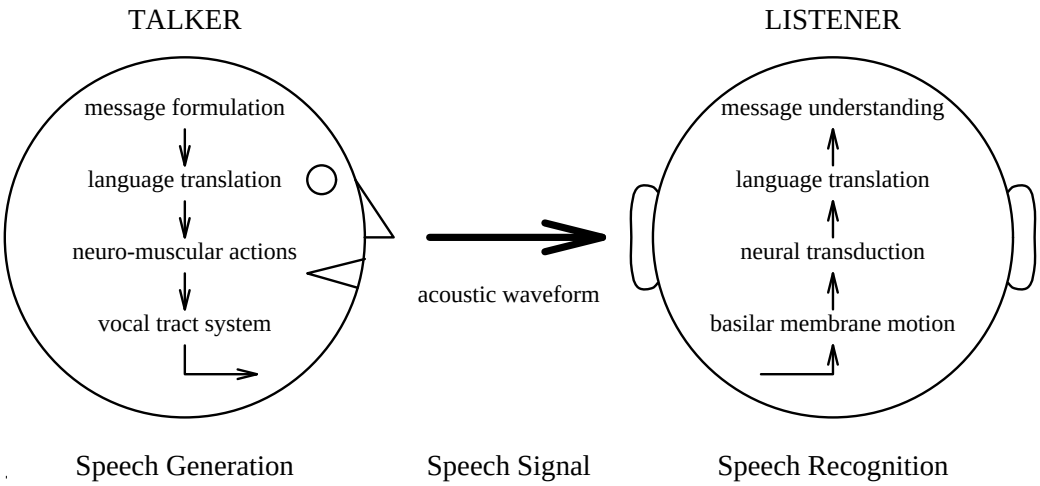


Figure 2.1: Schematic diagram of the speech production / speech perception process (after [1]).

frequencies.

The speech signal generated in this way is a slowly time varying signal. This means that, when examined over a sufficient short period of time (between 5 and 100 msec), its characteristics are fairly stationary. Over longer periods of time (on the order of 200 ms and more) the signal characteristics change, reflecting the different speech sounds being spoken. In normal speech, on the order of 10 phonemes per second are articulated in average.

2.1.2 Speech Recognition Systems

The task of speech recognition can be defined in many different ways. Recognition can be performed either on isolated words or utterances or on continuous speech. It is also distinguished between speaker dependent and speaker independent systems. Speaker independent recognition of continuous speech represents the most difficult and interesting form of speech recognition. To be able to build reasonable performing speech recognition system, it is often necessary to include some knowledge about the task for which the system will be used. In a first step, this is typically done be defining a limited set of words and a syntax describing possible word sequences. In more complex systems, also semantics and other knowledge is included.

A general model of a speech recognition system is shown in Figure 2.2. The model begins with a user creating a speech signal (speaking) to accomplish a given task. The speech signal is recognised by decoding it into a series of words according to the syntax of the task and other higher level knowledge. The meaning of these words is obtained by a higher level processor that uses a dynamic knowledge representation to take into account also the context of what it has previously recognised. Finally, the recognition system responds to the user for example in form of a voice output or a requested task being performed. If necessary, the user can react upon the system's output and thus continue in a dialogue-like manner.

In speech recognition it is generally assumed that speech is a realisation of some message encoded in a speech signal. The task of a speech recogniser can be defined as the task of

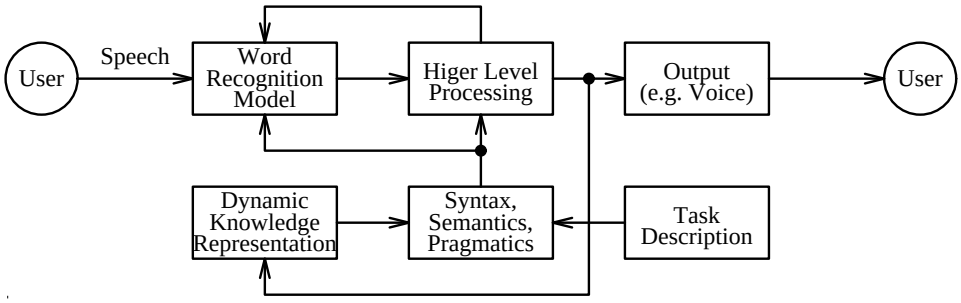


Figure 2.2: General block diagram of a task-oriented speech recognition system (after [1]).

finding an estimate $\hat{W}$ of the spoken message W given the observed speech signal O. This can be written as

$$\hat{W} = \arg\max_{W} P[W|O], \tag{2.1}$$

where

$$P[W|O] = \frac{P[O,W]}{P[O]} = \frac{P[O|W]\,P[W]}{P[O]}. \tag{2.2}$$

Since $P[O]$ is not a function of W, the maximisation in Equation 2.1 can be written as

$$\hat{W} = \arg\max_{W} P[O|W]\,P[W]. \tag{2.3}$$

Assuming that the a priori message probabilities $P[W]$ are known, the most likely spoken message $\hat{W}$ depends only on the likelihood $P[O|W]$. In the following, different methods for calculating $P[O|W]$ will be reviewed.

2.1.3 Modeling and Recognition of the Speech Signal

To be able to calculate the likelihood $P[O|W]$ in Equation 2.3, it is necessary to have models of the different speech signals observed when different messages are spoken. To simplify the process of creating good models in speech recognition systems with a high number of possible messages, a message is often divided up into smaller recognition units that can occur in several different messages. A sentence for example can be divided up into a sequence of words, a word into a sequence of phonemes, etc..

Before a speech signal actually is modelled, the features that are characteristic for the speech signal are extracted from its waveform. Then, models for all the different recognition units (messages, words, phonemes, etc.) used by the recognition system are created. A model of a recognition unit should describe the features or the sequence of features that is typical for the speech signal representing that recognition unit.

The whole speech recognition process is illustrated in Figure 2.3. Generalising it is assumed that the message is a sequence of symbols where each symbol represents a recognition unit. The Figure shows also that it is necessary to find the boundaries between the different recognition units if a message is not modelled as a whole.

2.1.3.1 Parameterising the Speech Signal

To perform speech recognition, it is necessary to extract features from the speech signal that allow to distinguish clearly between the different speech sounds. On the other hand,

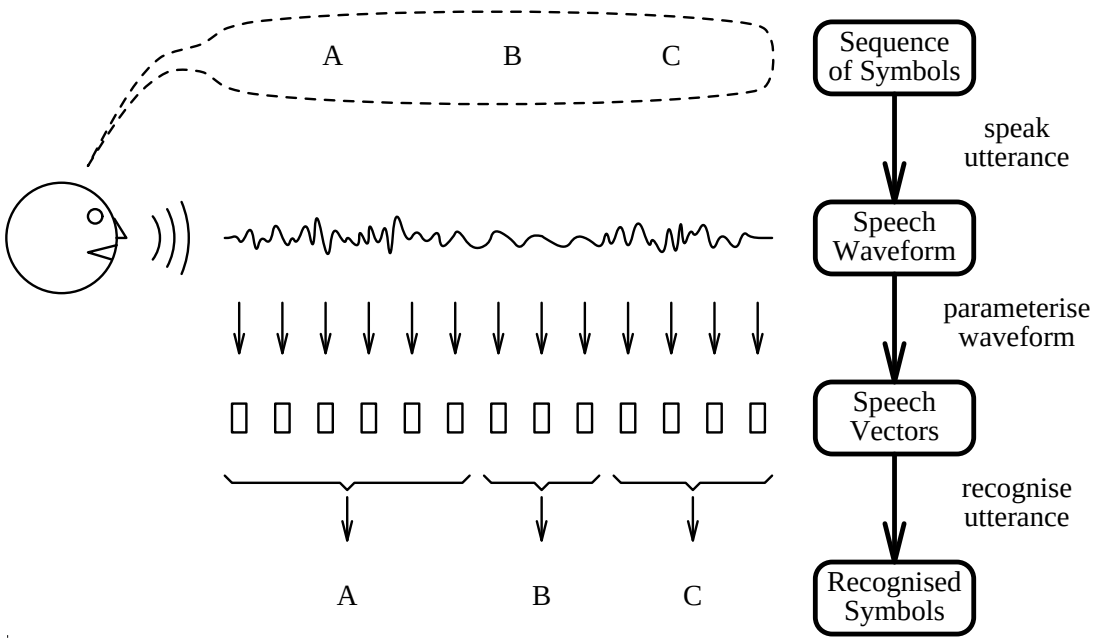


Figure 2.3: The speech recognition process (after [4]).

these features should vary as little as possible if the same sound is spoken by different speakers and under different circumstances. The extracted features are normally presented in form of a parameter vector. Since speech is a slowly time varying signal and its characteristic properties can change significantly a small fraction of a second, the features should be extracted periodically. Therefore it is common to calculate a new parameter vector typically every 5 to 25ms. The segments (frames) of the speech signal that are used to calculate adjacent parameter vectors normally overlap.

Most feature extraction methods are based on the (short-time) spectrum of the speech signal in the frame being parameterised. Typical parametric representations are smoothed spectra or linear prediction coefficients plus various other representations derived from these. The techniques for modeling speech signals used in this thesis are nearly independent from the way in which the actual parameter vector is obtained. Hence, only a short overview of the parameterisation used in this thesis is given here:

The speech waveform is first preemphasised and then divided into a sequence of overlapping frames. Each frame is 25 ms long and the frame centers are 10 ms apart. A Hamming window is applied and the FFT is calculated. Then a mel-scale triangular filterbank with 24 filters is applied to the magnitude spectrum. The mel-scale is a critical band frequency scale that takes into account the frequency perception in the human auditory system. A DCT is applied to the log mel-scale filter outputs and thus 12 *mel-frequency cepstral coefficients* (MFCC) are obtained. Then cepstral liftering is performed. The log energy of the frame is appended as a 13th element to the MFCC vector. Finally, the delta and acceleration coefficients (i.e. first and second order regression coefficients) are appended to the parameter vector. Thus, a 39 element parameter vector is calculated for each 10ms speech frame.

2.1.3.2 Modeling the Speech Signal

The simplest way of modeling a recognition unit is to take the sequence of parameter vectors that were calculated from a single utterance of that recognition unit as a reference pattern. To obtain a more reliable reference pattern, it is common to take several utterances of the same recognition unit into account. Then either the most typical sequence of parameters vectors or a sequence found by averaging the parameter vectors for several utterances can be used as a reference pattern.

In this thesis, probabilistic models, so-called Hidden Markov Models (HMM), are used to represent the different recognition units. An HMM is specified by a set of parameters that describe the typical features of the speech signal, their variation and how these change during the duration of the recognition unit. To estimate the parameter set of an HMM, its necessarry to have several utterances of the recognition unit to be modelled. A more detailed summary of the theory of HMMs is given in Section 2.2.

There are also other ways of modeling speech signals. E.g. also the concept of Neural Networks has been applied to speech recognition. If the spoken messages that are to be recognised are not modelled as a whole but were divided up into several recognition units, it is also necessary to have an overlaying model that describes how the models of the recognition units have to be combined to make up a model of a full message. When the recognition units are words, this overlaying model can e.g. be a grammar.

2.1.3.3 Recognising the Speech Signal

When a spoken utterance is to be recognised, it has to be compared with the known models of the different possible spoken messages. As a result of this comparison, the likelihood $P[O|W]$ from Equation 2.3 is found for every possible message. Instead of this likelihood, a score measuring the similarity between the spoken utterance and the known models can be used.

If a message (or recognition unit) is modelled by a reference pattern, two problems have to be solved during the comparison. First, a distance measure is needed so that the dissimilarity between to parameter vectors can be measured. The other problem is to determine which parameter vectors in the two sequences (reference pattern and spoken utterance) have to be compared with each other. This problem arises since the total duration of a recognition unit and also the relative duration of the different segments within a recognition unit can vary. It is normally solved by a dynamic programming algorithm called *Dynamic Time-Warping* (DTW). This algorithm tries to find an optimal time-alignment of the two sequences and returns a measure of the dissimilarity between the two sequences. Finally, the model that gives the lowest dissimilarity is determined. The message modelled by this model is assumed to be the message spoken.

When HMMs are used to model the messages, the likelihood $P[O|W]$ can be calculated for each HMM. It specifies how likely it is that the message modelled by an HMM could have led to the spoken utterance that was observed. Finally, the most likely model is determined and thus the most likely message is found.

If the messages are modelled as a sequence of recognition units, the complexity of the recognition process increases significantly. This is caused by the fact that many different combinations of recognition units have to be tested and that also the boundaries between

these units have to be determined. HMMs offer clear advantages at this point since their concept allows to model the speech process also on higher levels, e.g. in form of a statistical word grammar.

2.2 Hidden Markov Models

The speech signal can be characterised as parametric random process. It can be modelled by a Hidden Markov Model (HMM), a statistical model which offers a well-defined way of estimating the process parameters. A good review of the theory of HMMs and their application in speech recognition can be found in [3]. In the following, a short overview of this area is given.

2.2.1 The Discrete Markov Process

A discrete Markov process is a system that at any discrete time instance $t = 1, 2, \ldots$ is in one of a set of N distinct states $\{S_1, S_2, \ldots, S_N\}$. The actual state at time t is denoted by x_t. When going from one time instance to the next, the system changes its state (possibly back to the same state) according to a set of probabilities associated with that state. Here, only first order Markov processes are considered, where the transition probabilities depend only on the preceding state and are independent of time. Hence, the probability of transition from state S_i to state S_j is

$$a_{ij} = P[x_t = S_j | x_{t-1} = S_i], \quad 1 \leq i, j \leq N \tag{2.4}$$

with

$$\sum_{j=1}^{N} a_{ij} = 1, \quad 1 \leq i \leq N. \tag{2.5}$$

This set of transition probabilities a_{ij} can be written as matrix $A = \{a_{ij}\}$. An example of a discrete Markov process is illustrated in Figure 2.4.

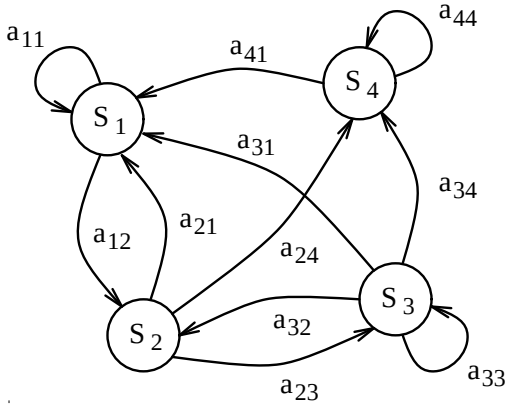


Figure 2.4: Discrete Markov process with $N = 4$ states and selected state transitions.

Since the output of such a Markov process is the sequence of states for the consecutive time instances, it could be called a observable Markov model. To avoid handling the initial state distribution as a special case, it is simply assumed here that the model initially (at

$t = 0$) is in state $x_0 = S_1$. If necessary, S_1 can be used as a special entry state to model the initial state distribution. The probability of the state sequence $X = \{x_1, x_2, \ldots, x_T\}$ can now be calculated as

$$P[X|\lambda] = \prod_{t=1}^{T} a_{x_{t-1}x_t}, \tag{2.6}$$

where λ denotes a Markov model with a specified matrix A.

2.2.2 Extension to Hidden Markov Models

In the observable Markov models considered so far, each state can be seen as a deterministically observable event. To extend this model to a hidden Markov model, it is now assumed that the observable event is a probabilistic function of the state. This means, that the underlying stochastic process is not directly observable (it is hidden) but can be observed only through another set of stochastic processes that produce the sequence of observations.

The probability that the event o is observed in state S_j is

$$b_j(o) = P[o|S_j]. \tag{2.7}$$

The event observed at time t is denoted by o_t. If o can only have discrete values, $b_j(o)$ is a discrete probability distribution. Otherwise, $b_j(o)$ is a continuous probability density. The set of output probability distributions can be written as $B = \{b_j(o)\}$.

Figure 2.5 shows an example of an HMM together with a possible observation sequence that could be generated by this model. This HMM is a special *left-right* model that has no state transitions to earlier states. It also has non-emitting entry and exit states — a concept that is used by HTK to simplify the construction of bigger composite models.

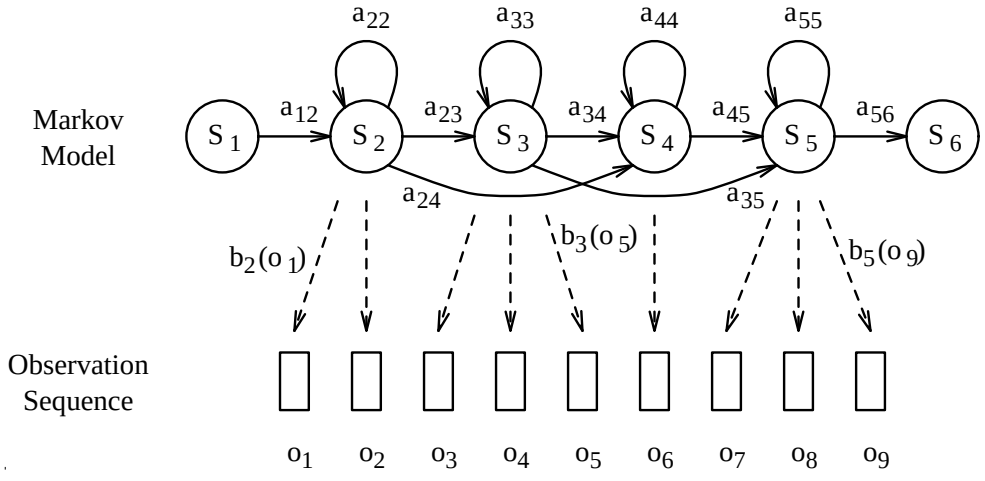


Figure 2.5: Hidden Markov model and a possible observation sequence.

To specify an HMM, the number of states N and the two probability measures A and B must be specified. For convenience, the compact notation $\lambda = (A, B)$ can be used.

2.2.3 The Three Basic Problems for HMMs

For HMMs of the form defined here, there are three basic problems that must be solved for the model to be useful in real-world applications like speech recognition. These problems are:

- **Problem 1:** Given the observation sequence $O = \{o_1, o_2, \ldots, o_T\}$ and the model $\lambda = (A, B)$, how can $P[O|\lambda]$, the probability of the observation sequence, given the model, be calculated efficiently?
- **Problem 2:** Given the observation sequence $O = \{o_1, o_2, \ldots, o_T\}$ and the model $\lambda = (A, B)$, how can a corresponding state sequence $X = \{x_1, x_2, \ldots, x_T\}$ be found that is optimal in a meaningful sense (i.e., explains the observations best)?
- **Problem 3:** Given the observation sequence $O = \{o_1, o_2, \ldots, o_T\}$, how can the model parameters A and B be adjusted so that $P[O|\lambda]$ is maximised?

In the following, possible solutions to these three problems are summarised.

2.2.3.1 Problem 1

The most straightforward way of calculating $P[O|\lambda]$ is to take in account all possible state sequences X of length T individually. For a given state sequence $X = \{x_1, x_2, \ldots, x_T\}$, the probability of the observation sequence O is

$$P[O|X, \lambda] = \prod_{t=1}^{T} P[o_t|x_t, \lambda] = \prod_{t=1}^{T} b_{x_t}(o_t). \tag{2.8}$$

The probability $P[X|\lambda]$ of such a state sequence X is given in Equation 2.6. The joint probability of O and X is now the product

$$P[O, X|\lambda] = P[O|X, \lambda]\, P[X|\lambda]. \tag{2.9}$$

The probability of O given the model λ is obtained by summing this joint probability over all possible state sequences X, thus giving

$$P[O|\lambda] = \sum_{X} P[O, X|\lambda] = \sum_{x_1, x_2, \ldots, x_T} \prod_{t=1}^{T} a_{x_{t-1}x_t} b_{x_t}(o_t). \tag{2.10}$$

The direct definition of $P[O|\lambda]$ in Equation 2.10 requires on the order of $2TN^T$ calculations, which is computationally unfeasible. The *forward-backward procedure* offers a much more efficient way to calculate $P[O|\lambda]$. It makes use of the forward variable

$$\alpha_t(i) = P[\{o_1, o_2, \ldots, o_t\}, x_t = S_i|\lambda]. \tag{2.11}$$

Assuming a known initial state x_0 and using the initialisation

$$\alpha_0(i) = \begin{cases} 1 & \text{if } x_0 = S_i \\ 0 & \text{otherwise} \end{cases}, \quad 1 \leq i \leq N, \tag{2.12}$$

$\alpha_t(i)$ can be calculated iteratively as

$$\alpha_t(j) = \left(\sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} \right) b_j(o_t), \quad 1 \leq j \leq N, \quad t = 1, 2, \ldots, T. \tag{2.13}$$

Finally, $P[O|\lambda]$ can be calculated as

$$P[O|\lambda] = \sum_{i=1}^{N} \alpha_t(i). \tag{2.14}$$

This forward procedure, where Equation 2.12 is computed iteratively for $t = 1, 2, \ldots, T$, is based upon a lattice (or trellis) structure. It only requires on the order of N^2T calculations.

The backward variable

$$\beta_t(i) = P[\{o_{t+1}, o_{t+2}, \ldots, o_T\} | x_t = S_i, \lambda]. \tag{2.15}$$

can be calculated in a similar manner. With the initialisation

$$\beta_T(i) = 1, \quad 1 \leq i \leq N, \tag{2.16}$$

$\beta_t(i)$ can be calculated iteratively as

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(i) a_{ij} b_j(o_{t+1}), \quad 1 \leq i \leq N, \quad t = T-1, T-2, \ldots, 1. \tag{2.17}$$

Now, $P[O|\lambda]$ can be expressed in terms of the forward-backward variables as

$$P[O|\lambda] = \sum_{i=1}^{N} P[O, x_t = S_i|\lambda] = \sum_{i=1}^{N} \alpha_t(i) \beta_t(i) \tag{2.18}$$

which is valid for all $1 \leq t \leq T$.

2.2.3.2 Problem 2

The solution to problem 2 depends on the definition of the "optimal" state sequence. For example, one possible optimality criterion is to choose the states x_t that are *individually* most likely. To implement this solution, the probability measure

$$\gamma_t(i) = P[x_t = S_i|O, \lambda] = \frac{P[x_t = S_i, O|\lambda]}{P[O|\lambda]} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)} \tag{2.19}$$

is defined. Using $\gamma_t(i)$, the individually most likely state x_t at time t is found as

$$x_t = \arg \max_{1 \leq i \leq N} \gamma_t(i), \quad 1 \leq t \leq T. \tag{2.20}$$

Although Equation 2.20 maximises the expected number of correct states, the resulting state sequence X might not be a valid state sequence. This can happen if some state transitions have zero probability ($a_{ij} = 0$ for some i and j).

Because of this problem, the most widely used criterion is to find the *single* best state sequence (path) $X = \{x_1, x_2, \ldots, x_T\}$ for the given observation sequence $O = \{o_1, o_2, \ldots, o_T\}$.

This means that $P[X|O,\lambda]$ is to be maximised, which is equivalent to maximising $P[X,O|\lambda]$. The *Viterbi algorithm* is an efficient dynamic programming method to perform this maximisation and thereby finding the best state sequence.

The Viterbi algorithm makes use of the variable

$$\delta_t(i) = \max_{x_1,x_2,\ldots,x_t} P[\{x_1,x_2,\ldots,x_t = S_i\},\{o_1,o_2,\ldots,o_t\}|\lambda] \tag{2.21}$$

which gives the highest probability (best score) along a single path which accounts for the first t observations o_t and ends in state $x_t = S_i$. To retrieve the best state sequence X^*, the backpointer array $\psi_t(j)$ is used. Assuming a known initial state x_0 and using the initialisation

$$\delta_0(i) = \begin{cases} 1 & \text{if } x_0 = S_i \\ 0 & \text{otherwise} \end{cases}, \quad 1 \le i \le N, \tag{2.22}$$

$\delta_t(j)$ and $\psi_t(j)$ can be calculated iteratively as

$$\delta_t(j) = \left(\max_{1\le i\le N} \delta_{t-1}(i)a_{ij}\right) b_j(o_t), \quad 1 \le j \le N, \quad t = 1,2,\ldots,T, \tag{2.23}$$

$$\psi_t(j) = \arg\max_{1\le i\le N} \delta_{t-1}(i)a_{ij}, \quad 1 \le j \le N, \quad t = 1,2,\ldots,T. \tag{2.24}$$

Finally, the best score P^* and the last state x_t^* of the best path are calculated as

$$P^* = P[X^*,O|\lambda] = \max_{1\le i\le N} \delta_T(i), \tag{2.25}$$

$$x_T^* = \arg\max_{1\le i\le N} \delta_T(i). \tag{2.26}$$

The full best path (state sequence) $X^* = \{x_1^*, x_2^*, \ldots, x_T^*\}$ is found by backtracing:

$$x_t^* = \psi_{t+1}(x_{t+1}^*), \quad t = T-1, T-2, \ldots, 1. \tag{2.27}$$

The Viterbi algorithm is similar to the forward calculation described above. The major difference is the maximisation in Equation 2.23 which is used instead of the summation in Equation 2.12. The lattice (or trellis) structure used by the Viterbi algorithm is illustrated in Figure 2.6. There, the same HMM and observation sequence as in Figure 2.5 is used.

2.2.3.3 Problem 3

The last and most difficult problem is the estimation of the model parameters A and B in such a way, that they maximise the probability of the observation sequence given the model. Since the HMMs used in this thesis already were trained, the procedure of model parameter estimation (i.e. training the HMMs) is not described very detailed here.

The HTK system used in this thesis employs continuous density HMMs, where the probability densities $b_j(o)$ for the observation vectors o are represented by Gaussian mixtures. The formula for computing $b_j(o)$ is

$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} b_{jm}(o_t) = \sum_{m=1}^{M} c_{jm} \mathcal{N}(o_t;\mu_{jm},\Sigma_{jm}), \tag{2.28}$$

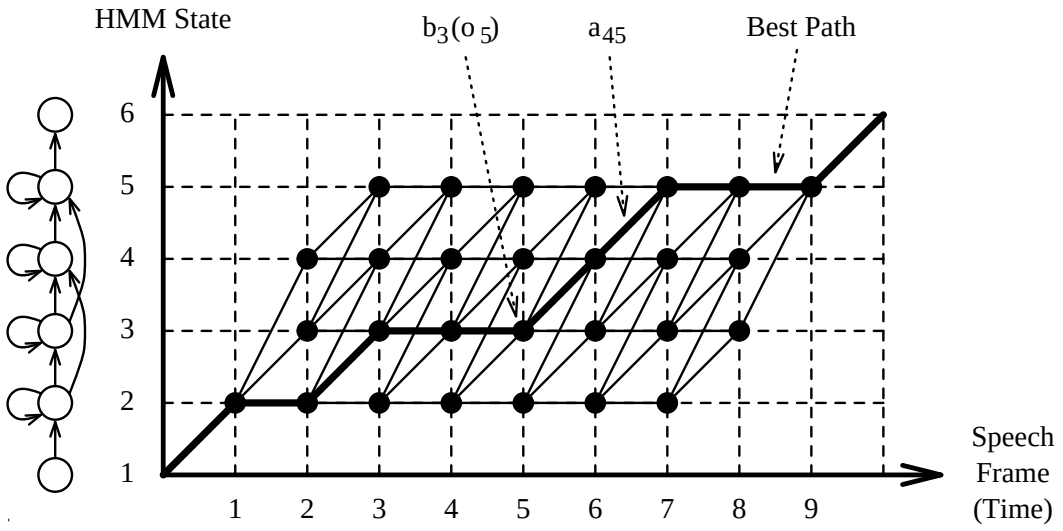


Figure 2.6: Lattice (or trellis) structure used by the Viterbi algorithm.

where M is the number of mixtures, c_{jm} is the weight of the m'th mixture component and $b_{jm}(o_t) = \mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm})$ is the Gaussian probability density

$$\mathcal{N}(o; \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{-\frac{1}{2}(o-\mu)' \Sigma^{-1} (o-\mu)}, \tag{2.29}$$

with the mean vector μ, the covariance matrix Σ and n being the dimensionality of the observation vector o.

To optimise the model parameters, commonly the *Baum-Welch* method, an iterative reestimation procedure, is used. In every iteration, new model parameters $\hat{A}$ and $\hat{B}$ are estimated from the previous parameters A and B so that the modeling of the given training observation sequence O is improved. For this reestimating, the probability measure

$$\begin{aligned} \xi_t(i,j) &= P[x_t = S_i, x_{t+1} = S_j | O, \lambda] \\ &= \frac{P[x_t = S_i, x_{t+1} = S_j, O|\lambda]}{P[O|\lambda]} \\ &= \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)} \end{aligned} \tag{2.30}$$

is used. It can be related to $\gamma_t(i)$ in Equation 2.19 by

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j). \tag{2.31}$$

Now the transition probabilities a_{ij} can be reestimated as

$$\begin{aligned} \hat{a}_{i,j} &= \frac{\text{expected number of transitions from } S_i \text{ to } S_j}{\text{expected number of transitions from } S_i} \\ &= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}. \end{aligned} \tag{2.32}$$

If a set of R training observations O^r, $1 \leq r \leq R$, is used instead of only one training observation O, the a_{ij} can be reestimated as

$$\hat{a}_{i,j} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T^r-1} \xi_t^r(i,j)}{\sum_{r=1}^{R} \sum_{t=1}^{T^r-1} \gamma_t^r(i)}. \tag{2.33}$$

To reestimate the parameters B of the observation probability densities $b_j(o)$, the probability of occupying the m'th mixture component in state S_j at time t for the r'th observation

$$\begin{aligned}\gamma_t^r(j,m) &= P[y_j = m, x_t = S_j | O^r, \lambda] \\ &= P[y_j = m | x_t = S_j, O^r, \lambda] \frac{P[x_t = S_j, O^r | \lambda]}{P[O^r|\lambda]} \\ &= \frac{c_{jm} b_{jm}(o_t)}{b_j(o_t)} \gamma_t^r(j)\end{aligned} \tag{2.34}$$

is needed. y_j denotes the mixture component occupied in state j. Now the parameters c_{jm}, μ_{jm} and Σ_{jm} can be reestimated as

$$\hat{c}_{jm} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j,m)}{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j)}, \tag{2.35}$$

$$\hat{\mu}_{jm} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j,m) o_t^r}{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j,m)}, \tag{2.36}$$

$$\hat{\Sigma}_{jm} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j,m) (o_t^r - \mu_{jm})(o_t^r - \mu_{jm})'}{\sum_{r=1}^{R} \sum_{t=1}^{T^r} \gamma_t^r(j,m)}. \tag{2.37}$$

2.2.4 Speech Recognition with HMMs

To explain how HMMs are applied in speech recognition, the simplest case is used, where every possible spoken message is modelled by its own HMM. To train the HMM for one given message, a set of training utterances of this message is needed. Also the type of HMM (number of states N, set of allowed state transitions $a_{ij} > 0$, number of mixture components M) has to be specified. Then, a first estimate of the parameters A and B of the HMM is found e.g. by uniform segmentation into N states along the time axis and subsequently calculation of the means and variances of the observation vectors in the different segments. Finally, the HMM parameters are recursively reestimated as described in the solution to problem 3. The recursion is aborted if a maximum number of iterations is reached or if the parameters have converged. Thus, distinct HMMs λ_i are generated for all possible messages W_i used in the speech recognition system.

To recognise a spoken utterance (represented by its observation sequence O), the total likelihood $P[O|\lambda_i]$ in Equation 2.3 could be calculated for all models by the forward-backward algorithm described in the solution to problem 1. But in practice, it is preferable to base the recognition on the maximum likelihood state sequence since this can easily be generalised to the continuous speech case. To find this state sequence and its likelihood, the Viterbi algorithm described in the solution to problem 2 is used.

To avoid numerical problems (underflow) during the calculation of the different likelihoods, it is common to use $\log P$ instead of P. This implies, that the multiplications needed to calculate the likelihood of a state sequence are substituted by a simple summation of the corresponding log probabilities.

2.2.5 Continuous Speech Recognition

In the case of continuous speech recognition, a whole message is modelled as a sequence of different recognition units. This means that every distinct recognition unit is modelled by its own HMM. To train these HMMs, it is necessary to have labelled training utterances where the beginning and end of the recognition units is known. Thus, the segments of the observation sequences that correspond to a given recognition unit can be "cut out" of the training utterances and the HMM for that recognition unit can be trained as above. Since the (manual) segmentation and labeling of the training utterances requires a lot of time, this is normally done only for a part of the training utterances.

The preliminary HMMs generated in this way can then be further refined by *embedded reestimation*. For this process, only the sequence of recognition units that make up the training utterance has to be known. Then, a big HMM representing the whole message can be constructed by connecting the HMMs for the different recognition units in that message. The special non-emitting entry and exit states used by HTK (see Figure 2.5) make it easy to "glue" the HMMs together. Finally, the parameters of all HMMs can be reestimated recursively by an algorithm similar to the one used for reestimation of the parameters of a single HMM. The procedure of embedded reestimation can be seen as including a kind of implicit segmentation of the training utterances according to the "knowledge" contained in the preliminary HMM parameters.

To perform speech recognition based on the HMMs for the recognition units, it is necessary to know in which way these units can be combined to make up legal messages. This information is commonly represented as a network where the nodes correspond to recognition units. The network can e.g. represent a grammar (or syntax). Figure 2.7 shows an example of such a network. Contrary to this example, normally whole words would be used as recognition units for such a small vocabulary (10 words an 2 silence models). But for bigger vocabularies (like the 1000 word DARPA resource management task used later in this thesis), it is common to use sub-word units like phonemes as basic recognition units. This allows a more reliable estimation of the HMM parameters due to the greater number of training examples. On the other hand, a transcription in terms of the basic recognition units is needed for all words in the vocabulary. This can e.g. be a lexicon of phoneme transcriptions.

The whole network can be seen as a big HMM consisting of the HMMs for the basic recognition units glued together by means of their entry and exit states. To recognise a spoken utterance, the Viterbi algorithm is used to find the best path through this big HMM. Finally, the names of the words and word-external models passed by this best path are used to build a hypothesis of the spoken message.

2.3 The Hidden Markov Model Toolkit

In this thesis, the *Hidden Markov Model Toolkit* (HTK) has been used [4]:

> HTK is a toolkit for building continuous density hidden Markov model (HMM) based recognisers. It is primarily intended for building sub-word based continuous speech recognisers and can be used in a wide range of pattern classification problems. HTK is built on an extensible, modular library that simplifies the

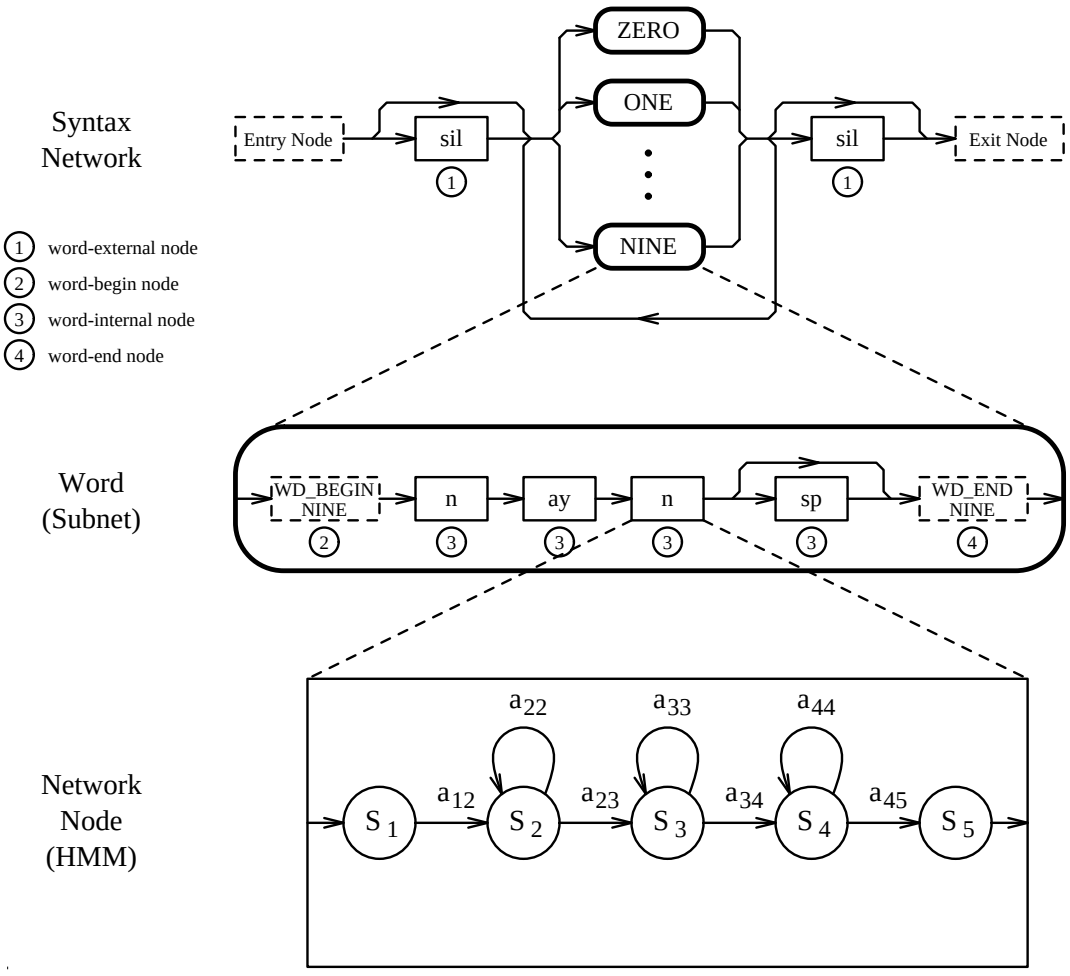


Figure 2.7: A simple syntax network.

development of user-written tools. The toolkit includes signal processing functions, HMM training and testing tools, language modeling support and scoring software. HTK is ideal for research in HMM modeling and recognition.

In the following, a short overview of HTK is given and its Viterbi recogniser is explained. Finally, the 1000 word DARPA resource management task used in this thesis is described.

2.3.1 Overview of HTK

HTK consists of a set of tools that perform the different tasks in an HMM based recognition system. These tools are written in the programming language C [22] and make use of a library of basic functions for handling HMMs. Different data files are used to transfer data between the different tools. These files can contain speech data (as a waveform or as sequence of observation vectors), speech labeling data, HMMs (the parameters that define them) or recognition networks. HTK allows to use nearly arbitrary HMM structures. It is also possible that HMM parameters can be shared by different mixtures, states, HMMs, etc.. In HTK, the basic recognition unit (e.g. a phoneme or a word) modelled by a single HMM is called a *phone*. To model coarticulation effects in sub-word based recognisers, HTK offers *context dependent* HMMs. This means that a distinct biphone or triphone model is used for every possible context of phones.

These are the main tools in HTK:

- **HCode:** This program is used to encode a speech waveform file into a parameterised form (a sequence of observation vectors).
- **HInit:** This tool is used to initialise an HMM on a set of labelled training data segments.
- **HRest:** This is the basic Baum-Welch reestimation tool that works on a set of labelled training data segments.
- **HERest:** This is the embedded training Baum-Welch reestimation tool. It trains a set of HMMs simultaneously using a training data set consisting of continuously spoken sentences and their corresponding transcriptions.
- **HVite:** This is a continuous speech Viterbi recogniser with syntax constraints and beam search.
- **HResults:** This program takes a pair of transcription files, performs a dynamic programming (DP) match between them and outputs recognitions statistics.
- **HSLab:** This program is a simple interactive speech label editor that displays speech waveforms and transcription files graphically.

2.3.2 The Viterbi Recogniser HVite

The tool HVite is a general-purpose Viterbi recogniser. It matches a network of HMMs against one ore more parameterised speech files and outputs a transcription file for each. The recognition network is constructed from a network definition file. Each network node

name refers to a HMM except for the two reserved names WD_BEGIN and WD_END. They are used to delimit the boundaries of a *composite word model* for word recognition systems based on sub-word units. Nodes between a WD_BEGIN / WD_END pair are called *word-internal* and all others nodes are called *word-external*. It is possible to specify external node names that are different from the (internal) node names referring to the HMMs. These external names are separated by a **%** in the network file and are used when the transcription file is generated. Word-external nodes don't appear in the transcription file if they have a null external name (denoted by **%%**). All WD_BEGIN / WD_END pairs must have an external name which is used to label that composite word model in the transcription file. Word-internal nodes never appear in the transcription file. An example of a network definition file is shown in Figure 2.8. It defines the network shown in Figure 2.7.

```
$ZERO = WD_BEGIN%ZERO z iy r ow [sp] WD_END%ZERO;
$ONE  = ...
  .
  .
  .
$NINE = WD_BEGIN%NINE n ay n [sp] WD_END%NINE;
$digit = $ZERO | $ONE | ... | $NINE;
( [sil%%] <$digit> [sil%%] )
```

Figure 2.8: A simple network file defining the network in Figure 2.7.

When reading the network definition file, an equivalent representation is build in memory. The network is represented by a set of nodes with explicit pointers to all successor and all predecessor nodes. Like the HMMs themselves, the network contains a single entry node and a single exit node called ENTER and EXIT respectively. The internal representation of a recognition network in HTK is illustrated in Figure 2.9.

Since the recognition network is regarded as a big HMM, it is also necessary to specify the transition probabilities from a network node M_i to its successor nodes M_j. To be precise, these transitions are performed from the non-emitting exit state of the HMM attached to the node M_i to the non-emitting entry states of the HMMs attached to the successor nodes M_j. The WD_BEGIN and WD_END nodes are represented by a kind of degenerated HMM that consist only of a single non-emitting state which is entry and exit state at the same time. For the word-external transitions (i.e. from a word-external or WD_END node to a word-external or WD_BEGIN node), the log transition probability is defined as

$$s \log P[M_j|M_i] + p \tag{2.38}$$

where s is a grammar scale factor and p is a fixed transition penalty (lower value of p gives higher penalty). Their default values are $s = 1$ and $p = 0$. If a matrix of a priori transition probabilities was specified in a special *bigrammar* file, then

$$P[M_j|M_i] = \text{bigram}[i, j] \tag{2.39}$$

is used. Otherwise

$$P[M_j|M_i] = 1/N_{\text{succ}}(i) \tag{2.40}$$

is used, where $N_{\text{succ}}(i)$ is the number of successors of node M_i. For the word-internal transition (i.e. from a word-internal or WD_BEGIN node to a word-internal or WD_END node), always the log transition probability $-\log N_{\text{succ}}(i)$ is used.

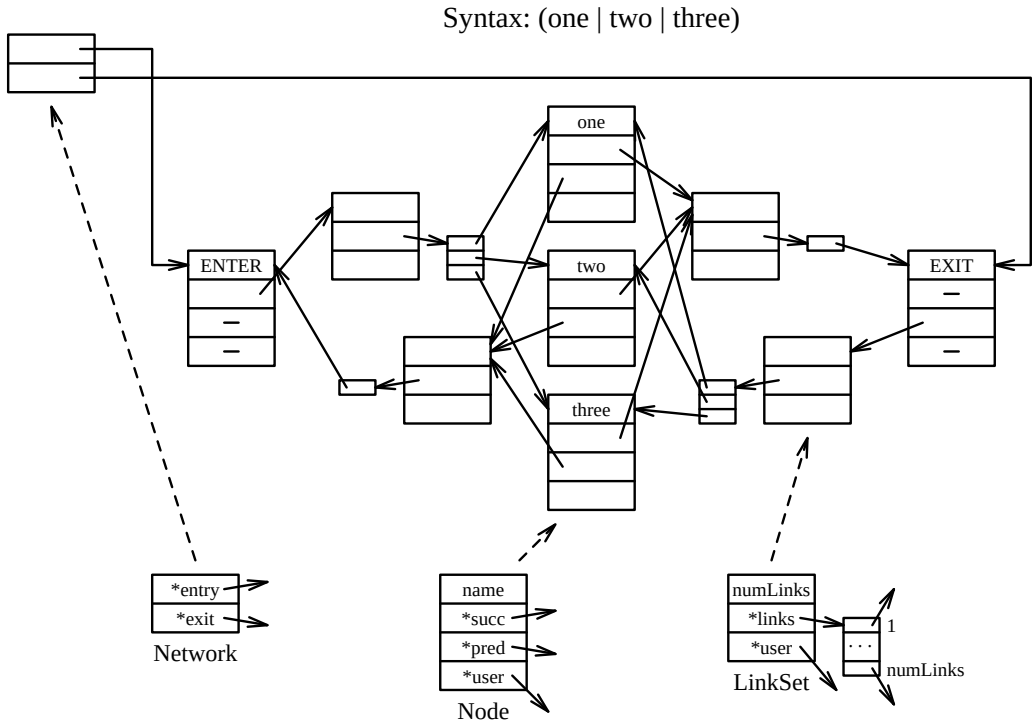


Figure 2.9: Internal representation of a recognition network in HTK (simplified).

The Viterbi algorithm in HVite is implemented using a concept called *token passing paradigm* [5]. It makes use of a *PhoneInstance* data structure attached to each network node. For all nodes except WD_BEGIN and WD_END nodes the PhoneInstance points to the corresponding HMM definition and contains an array which for each HMM state can hold a *token*. The PhoneInstance attached to a WD_BEGIN or WD_END nodes can only hold a single token. A token represents an alignment path and its probability up to the current time frame given the HMM system and the unknown utterance. Every state holds a single token since the best path to the current frame could end in any HMM state.

The basic token passing algorithm simply propagates tokens from each state to every connecting state updating the probability and history information. If more than one token is propagated to a state, only the best (most likely) token is maintained and thus the maximisation characteristic for the Viterbi algorithm is implemented. This propagation is repeatedly performed for every time frame in the utterance. First tokens are passed within each of the HMM instances (PhoneInstances) and then between the PhoneInstances according to the network links. Each best token propagated from a distinct WD_END or word-external node is recored in *phone link record* (PLR). The PLRs allow to trace back the alignment path of the best token after the whole utterance has been matched. The main data structure of HVite is illustrated in Figure 3.10. The structures *RankInfo*, *RankList* and *RankEntry* are not used by HVite and the output probabilities *Bjot* are only stored for the current time frame.

To increase processing speed, HVite can optionally perform a beam search. This means that at every time frame any PhoneInstance whose maximum log probability token falls more than a user specified threshold value below the maximum for all PhoneInstances is deactivated.

2.3.3 The DARPA Resource Management Corpus

The DARPA resource management (RM) corpus is a benchmark task widely used in evaluating large vocabulary continuous speech recognition systems. It consists of more that 2200 different sentences that obey a well defined syntax and has a vocabulary of 991 words. In this thesis, the speaker independent training and test data of the RM corpus was used. The training data consists of 3990 sentences spoken by 109 different speakers. There are 4 different test data sets available. Each of these sets consists of 300 sentences spoken by 10 different speakers. For all sentences, a word level transcription is available. The RM speech data is available on CD-ROM. It is recorded as "laboratory speech" with a good microphone in a silent room and is sampled at 16 kHz and uniformly quantised with 16 bit.

HTK contains also a *Resource Management Toolkit* (RMTK) that consists of script files, utility programs and data files related to the RM task. A basic phoneme-based recogniser for the RM task built according to the recipe in RMTK was available for this thesis. The speech files were parameterised as MFCC with appended energy, delta and acceleration coefficient in frames of 10 ms as described in Section 2.1.3.1. The recogniser uses a set of 47 HMMs, each modelling a phoneme, and has 2 additional silence models (see Table 2.1). The HMMs are context independent 5 state left-right models (monophones) without skip transitions (as in Figure 2.7) and use a single Gaussian mixture ($M = 1$) with diagonal covariance matrix Σ for the output distribution. The silence model "sp" is simpler and has only a single emitting state. Besides this baseline HMM set, here also referred to as "base", a second more advanced HMM set was available. It is part of a small demonstration of HTK and here referred to as "demo". It is based on the monophone HMM set mentioned above but consists of about 2300 context dependent HMMs (triphones) including function-word specific HMMs.

A lexicon containing the phoneme transcriptions for all 991 words in vocabulary is also provided. It is used to build the recognition networks for both HMM sets. Two different recognition networks have been used in this thesis. The first network employs simply *no grammar* so that the words can follow each other in any order. The other network uses a *word-pair grammar* where the set of possible successors is defined for every word in the vocabulary.

Stops

b	bee	b iy
d	day	d ey
g	gay	g ey
p	pea	p iy
t	tea	t iy
k	key	k iy
dx	muddy	m ah dx iy
	dirty	d er dx iy
ts	dates	d ey ts
	knots	n aa ts

Stop Closures

dd	made	m ey dd
	june	dd jh uw n
pd	upgrade	ah pd g r ey dd
	chopped	td ch aa pd t
td	test	t eh s td
	chart	td ch aa r td
kd	hawkbill	hh ao kd b ih l
	yorktown	y ao r kd t aw n

Affricates

jh	joke	dd jh ow k
ch	choke	td ch ow k

Fricatives

s	sea	s iy
sh	she	sh iy
z	zone	z ow n
f	fin	f ih n
th	thin	th ih n
v	van	v ae n
dh	then	dh eh n

Nasals

m	mom	m aa m
n	noon	n uw n
ng	sing	s ih ng
en	reasons	r iy z en z

Semivowels and Glides

l	lay	l ey
r	ray	r ey
w	way	w ey
y	yacht	y aa t
hh	hay	hh ey
el	bottle	b aa t el

Vowels

iy	beet	b iy t
ih	bit	b ih t
eh	bet	b eh t
ey	bait	b ey t
ae	bat	b ae t
aa	bott	b aa t
aw	bout	b aw t
ay	bite	b ay t
ah	but	b ah t
ao	bought	b ao t
oy	boy	b oy
ow	boat	b ow t
uh	book	b uh k
uw	boot	b uw t
er	bird	b er d
ax	about	ax b aw t

Others

sil	silence
sp	inter-word silence

Table 2.1: The RM Baseline Monophone Set: Symbols, example words and possible transcriptions (The set contains 47 phonemes and 2 silence models).

Chapter 3

N-Best Algorithms

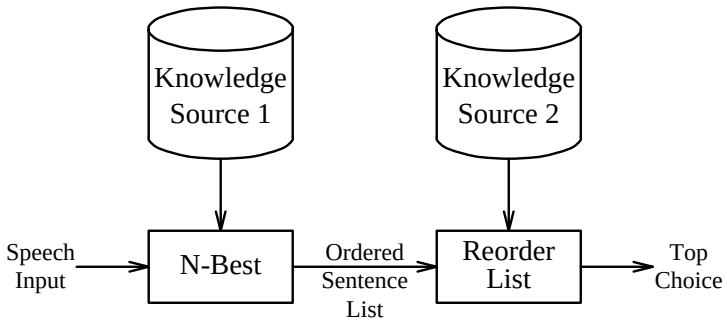
In this chapter, the N-best search paradigm is introduced and several different N-best algorithms are presented and compared. Then, an implementation of the tree-trellis N-best algorithm based on HTK is described. Finally, experimental results obtained with the tree-trellis algorithm are presented and discussed.

3.1 Introduction

A speech recognition system should take into account all available knowledge sources when recognising an utterance. Besides the speech signal and the models of the recognition units, also knowledge about syntax, semantic and other properties of the natural language might be used when searching for the most likely word sequence (hypothesis). One way to include these knowledge sources in the search process is to use them simultaneously to constrain a single search. Since many of the natural language knowledge sources contain "long-distance" effects (dependencies between words far apart in the input), the search can become quite complex. Furthermore, the common left-to-right search strategy requires that also all knowledges source are formulated in a predictive, left-to-right manner, which restricts the type of knowledge that can be used.

One way to solve these problems is to apply the knowledge sources not simultaneously but sequentially so that the search for the most likely hypothesis is constrained progressively [7]. Thus, the advantages provided by a knowledge source can be trade off against the costs of applying it. First, the most powerful and cheapest knowledge sources are applied to generate a list of the top N hypotheses. Then, these hypotheses are evaluated by the other more expensive knowledge source so that the list of hypotheses can be reordered according to a more advanced likelihood score. The whole N-best search paradigm is illustrated in Figure 3.1. As long as the correct hypothesis is among the N-best hypotheses, the N-best search paradigm finally will find the same correct hypothesis as a search that includes all knowledge sources simultaneously. The "correct hypothesis" is the most likely hypothesis according to the provided models and knowledge sources — whether this is really the utterance that was spoken, depends on the quality of the models and knowledge sources and *not* on the search algorithm.

Recently, different algorithms for finding the N-best sentence hypotheses have been proposed [7, 8, 9]. Some of these algorithms are exact while others use different approxima-



Knowledge Source 1	Knowledge Source 2
statistical grammar	full natural language model
syntax	semantics, etc.
bi-grammar	higher-order grammar
...	...

Figure 3.1: The N-best search paradigm: Combination of knowledge sources using the N-best algorithm (after [7]).

tions to reduce computation requirements. The different N-best algorithms are described and compared in Section 3.3.

Besides the N-best search paradigm, there are also other uses for these N-best algorithms:

- The N-best hypotheses lists generated during recognition tests can be used investigate new knowledge sources. Since it is not necessary to rerun the whole recognition process, experimental evaluation of the additional information provided by a new knowledge source can be done much easier [11].
- Methods for the discriminative training of HMMs usually require a list of errors and near-misses so that the correct answer can be made more likely and the errors and near-misses can be made less likely. Such a list can be easily provided by an N-best algorithm [11, 13]
- In a speech recognition system, some parameters like the weights of different knowledge sources (e.g. grammar-scalefactor, word-penalty, ...) can not be easily estimated. For the fine-tuning of these parameters, normally repeated recognition test are required. Using the N-best hypotheses lists generated during a single run of the recogniser, the parameter optimisation can then be done much easier [11].
- A modified version of the tree-trellis N-best algorithm can be used to generate the lexicon needed by sub-word based recogniser automatically [15]. Chapter 4 gives a detailed description of this technique.

3.2 The A* Tree Search Algorithm

Some of the N-best algorithms presented later make use of a special tree search algorithm, the A* algorithm. This algorithm is often applied to problems in the area of artificial intelligence. A detailed description of the A* algorithm can be found in [6, chapter 3]. It will be reviewed briefly now.

Many problems in artificial intelligence can be formulated in a state-space. This means that, starting from an initial state, operators are applied to state descriptions until a desired goal state is obtained. The 8-puzzle can be used as an example to illustrate this concept. Here, a configuration of the 8 numbered tiles on the 3×3 field is a *state* and moving the blank one step in one of the four possible directions is an *operation* (see also Figure 3.3).

The search for sequence of operations that will transform the start state into a goal state can be modeled by a graph. In this thesis, a tree as a special kind of graph is sufficient. Each node in this tree is associated to a state. By applying all possible operators to the state associated to a node, all successors of this node are generated and thus the node is *expanded*. First the start node having the initial state associated is expanded. Then the process of expanding successor nodes is continued until a goal node is generated. Pointers from the successor nodes back to their parent node are set up when a node is expanded. They allow to trace back the path to the start node when a goal node is finally found.

To minimise the number of expanded nodes and thus to optimise the tree search, a search method specifies the order in which the nodes are expanded. By using heuristic information about the global nature of the search tree and the general direction of the goal, it can be tried to "pull" the search towards the goal by expanding the most promising nodes first. This can be done by using an evaluation function $\hat{f}$ and selecting the node n with the smallest value $\hat{f}(n)$ to be expanded next. This ordered search method makes use of two lists called OPEN and CLOSED and is shown in Figure 3.2.

```
put start node s on OPEN
compute f̂(s) while OPEN is not empty
    remove from OPEN the node n whose f̂(n) value is smallest
    put n on CLOSED if n is a goal node
        obtain the solution path by tracing back through the pointers
        exit
    if node n has any successors
        expand node n by generating all of its successors n_i
        compute all f̂(n_i)
        put these successors on OPEN
        provide pointers back to n
exit with failure
```

Figure 3.2: The A* tree search algorithm.

The depth $d(n)$ of a node n is the length of the path from the start node s to the node n. Thus, the depth of the start node s is $d(s) = 0$, the depth of its successors s_i is $d(s_i) = 1$ etc.. With the depth limit $d_{\max}$, the evaluation function

$$\hat{f}(n) = -\min\{d(n), d_{\max}\} \tag{3.1}$$

results in a *depth-first* search where the deepest nodes are expanded first. On the other hand, the evaluation function

$$\hat{f}(n) = d(n) \tag{3.2}$$

results in a *breadth-first* search. It guarantees that the shortest path to a goal node will be found. The both search methods presented yet are *blind* methods since they make no use of heuristic information.

To obtain more powerful evaluation functions, costs can be associated to the arcs in the tree. Assuming a cost function $c(n_i, n_j)$ giving the cost of going from node n_i to its successor node n_j, the cost $g(n)$ of the path from the start node s to a node n can be calculated as

$$g(s) = 0 \tag{3.3}$$

$$g(n_i) = g(n) + c(n, n_i). \tag{3.4}$$

The evaluation function

$$\hat{f}(n) = g(n) \tag{3.5}$$

results in a *uniform-cost* search which finds the minimal cost path to a goal node. When using constant arc costs $c(n_i, n_j) = 1$, this method is identical to the breadth-first method.

The A* algorithm is an optimal search method that maximises search efficiency while still guaranteeing that a minimal cost path to a goal is found. These properties are obtained by using the evaluation function

$$\hat{f}(n) = g(n) + \hat{h}(n). \tag{3.6}$$

The function $\hat{f}(n)$ is an *estimate* of the cost of a minimal cost path from the start node s to a goal node which goes through node n. This path consists of to partial paths: The path from the start node to node n and the path from node n to a goal node. $g(n)$ is the cost of the first partial path, the minimal cost path from s to n. $\hat{h}(n)$ is an *estimate* of the cost $h(n)$ of the second partial path, the minimal cost path from node n to a goal node. The function $\hat{h}(n)$ is called the *heuristic function*. If $\hat{h}(n) \equiv 0$, no heuristic information is used and the A* algorithm is identical to the uniform-cost method.

The A* algorithm is *admissible* if $\hat{h}(n) \leq h(n)$ for all nodes n and if all arc costs are positive. This means that the search will always terminate in an optimal (i.e. minimal cost) path whenever such a path exists. If an admissible function $\hat{h}_1(n)$ is everywhere larger than an other admissible function $\hat{h}_2(n)$, $\hat{h}_1(n)$ can be called a *more informed* function giving a higher *heuristic power* than $\hat{h}_2(n)$. If $\hat{h}_{\text{opt}}(n)$ has a higher heuristic power than any other heuristic function $\hat{h}(n)$, $\hat{h}_{\text{opt}}(n)$ results in an *optimal* A* algorithm. The proofs and a detailed discussion of this theorems can be found in [6].

The A* algorithm can be illustrated by the 8-puzzle example shown in Figure 3.3. In this example, the cost of a path is defined as the number of tiles moved on this path (i.e. $g(n) = d(n)$). The number of misplaced tiles at node n is used as an admissible heuristic function $\hat{h}(n)$.

3.3 Different N-Best Algorithms

The Viterbi algorithm typically used in an HMM-based speech recogniser only finds the best word sequence (the hypothesis corresponding to the state sequence with the highest likelihood score). To obtain not only the first best hypothesis but the list of the best N hypotheses, several modifications of the Viterbi algorithm are necessary. Different algorithms that are able to find the N-best hypotheses are presented now.

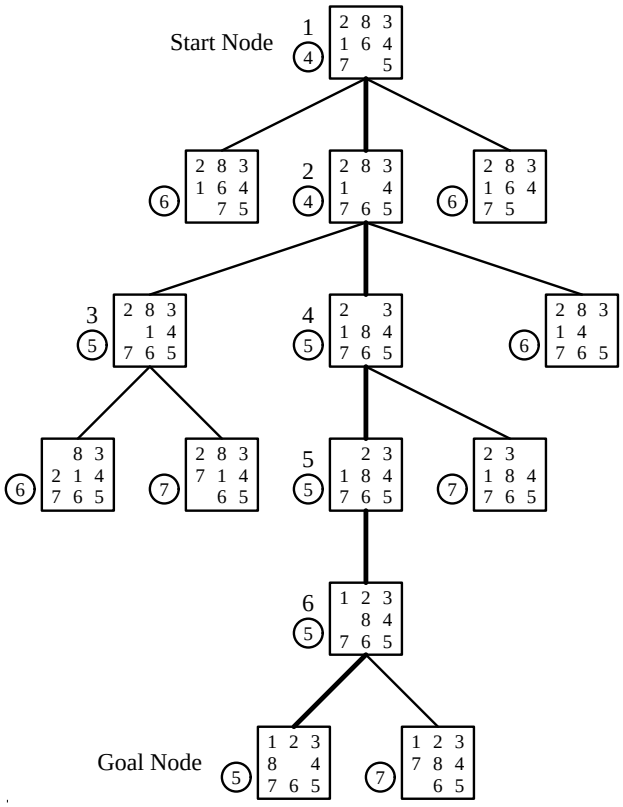


Figure 3.3: The tree produced by the A* algorithm while solving an 8-puzzle. The value of $\hat{f}$ for each node is circled, and the uncircled numbers show the order in which the nodes are expanded (after [6]).

3.3.1 The Exact N-Best Algorithm

The exact N-best algorithm was proposed in [7]. This algorithm is similar to the time-synchronous Viterbi algorithm, but instead of likelihood scores for state sequences, likelihood scores for word sequences are computed. To be able to find the N-best hypotheses, it is necessary to keep separate records for theories (paths) with different word sequence histories. When two or more paths come to the same state at the same time and also have the same history (word sequence), their probabilities are added. When all paths for a state have been calculated, only a specified number of these local theories are maintained. Their probabilities have to be within a threshold of the probability of the most like theory (word sequence) at that state. Therefore, any word sequence hypothesis that reaches the end of the utterance has an accurate likelihood score. This score is the conditional probability of the observed speech signal given the word sequence hypothesis. Thus, the list of the N-best hypotheses is generated. To reduce the exponentially growing number of possible word sequences, pruning is used. It can be shown that this algorithm will find all hypotheses that are within a search beam specified by the pruning thresholds [7].

Since the probabilities of paths with the same word sequence history are added, the *total likelihood score* is calculated, as opposed to the *maximum likelihood score* calculated by the Viterbi algorithm. This also leads to somewhat higher recognition rates [8].

To reduce the computation requirements connected with the exact N-best algorithm, it is possible to combine the N-best algorithm with the forward-backward search algorithm described in [10]. The forward-backward search algorithm takes place in two phases. In

the first phase, a fast time-synchronous search of the utterance in forward direction is performed. In the second phase, a more expensive search is performed, processing the utterance in reverse direction and using information gathered by the forward search. For example, a Viterbi search is performed in the forward phase and then the more expensive exact N-best search is performed in the backward phase. The information from the forward search is used to avoid expanding the backward search tree towards non-promising hypotheses and thus saves computation costs. This concept of combining a forward and a backward search is similar to the concept of the tree-trellis algorithm presented in Section 3.3.2.

3.3.2 The Tree-Trellis Algorithm

The tree-trellis algorithm for finding the N-best hypotheses was proposed in [9]. This algorithm combines a frame-synchronous forward trellis search with a frame-asynchronous backward tree search. In the forward trellis search, a modified Viterbi algorithm is used. In a normal Viterbi algorithm, only the backpointer arrays necessary to trace back the best hypothesis would be stored. The modified algorithm used here also stores rank-ordered predecessor lists for each grammar node and time frame. For a given grammar node and time frame, such a list has an entry for each predecessor of that grammar node. This entry contains the likelihood score of the best partial path comming via that predecessor to the grammar node. Before being stored, the entries in a predecessor list are rank-ordered according to their likelihood score. When the modified Viterbi search has reached the end of the utterance, the best hypothesis can already easily be obtained by backtracing.

In the backward search, an A* tree search algorithm is used to find the N-best hypotheses. This tree search starts from the end of the utterance at the final grammar node. In each step, the backward partial path is extended towards the beginning of the utterance by a time-reverse Viterbi search for the best single word extension. The best single word extension is found using the rank-ordered predecessor lists generated during the forward search. When the backward partial path reaches the beginning of the utterance, the best hypothesis is found (it is identical to the hypothesis already found in the forward Viterbi search). By continuing the A* tree search, the N-best hypotheses can be found sequentially. A block diagram of the whole tree-trellis algorithm is shown in Figure 3.4.

The backward tree search performed here is more complicated than a normal A* tree search due to the time varying aspects. This means that there are usually many paths for the same word sequence with different time trajectories and scores. Since only paths of different word content are considered here, paths with the same word content but different time trajectories are compared first. The score of the best possible path is then compared with other best paths of different word contents. Figure 3.5 illustrates the time varying aspect. Here, the word sequence up to grammar node $m(n)$ found in the backward tree search is first extended by the best word w' up to grammar node $m(n')$. This is done by a time-reverse Viterbi algorithm that extends the the best partial paths to grammar node $m(n)$ ending at the different time frames and thus finds the best paths to grammar node $m(n')$ for the different time frames. Then these extended backward partial paths to grammar node $m(n')$ are merged with the forward partial paths to grammar node $m(n')$ found in the modified Viterbi search. This means that for all time frames the backward partial path score is combined with the forward partial path scores of all predecessors of grammar node $m(n')$ that were stored during the modified Viterbi search. Finally, the

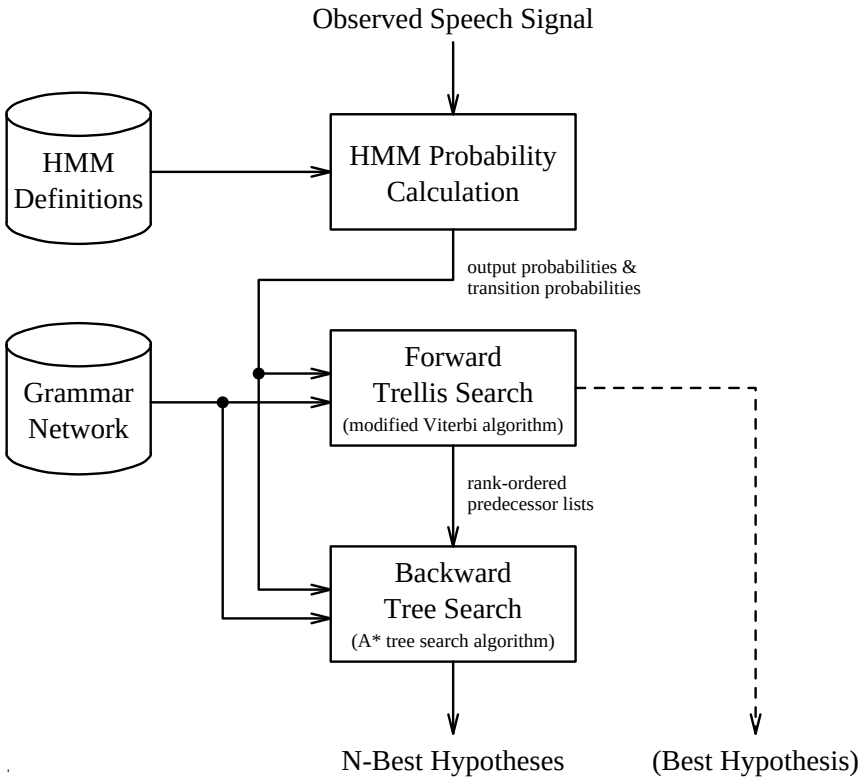


Figure 3.4: Block diagram of the tree-trellis algorithm.

highest combined likelihood score is found for each possible predecessor (i.e. each word that could be used to extend the backward path ending in grammar node $m(n')$) and thus the best word extension for the next step is found.

The full backward A* tree search algorithm is shown in Figure 3.6. A stack entry (node) n used by this A* search consists of a rank-ordered list of the $I(n)$ possible word extensions $w(n,i)$ and their total path scores $f(n,i)$ (with $f(n,i) \geq f(n,j)$ for $1 \leq i < j \leq I(n)$), the index $i_{\text{next}}(n)$ of the best word extension not yet performed, an array containing the likelihood scores $g(n,t)$ of the backward partial paths for each time frame t and a backpointer to the previous stack entry $n_{\text{pre}}(n)$. The rank-ordered lists of the $J(m(n))$ possible predecessors $p(m(n),j)$ of grammar node $m(n)$ and their forward partial path scores $h(m(n),t,j)$ at time frame t (with $h(m(n),t,i) \geq h(m(n),t,j)$ for $1 \leq i < j \leq J(m(n))$) have been generated in the forward search. $m(n)$ is the grammar node reached by the backward partial paths of n. The entries n on the OPEN stack are ordered according to $f(n, i_{\text{next}}(n))$. Since the modified Viterbi algorithm used in the forward search only differs from a normal Viterbi algorithm in the additional generation of the predecessor lists, it is not presented here again. The start node s for the backward tree search is the EXIT node $m(s)$ of the grammar network. The utterance consists of the time frames $t = 1, 2, \ldots, T$. In the time-reverse Viterbi search, $c(n, n', t, t')$ denotes the likelihood score of the best partial path for word w' from grammar node $m(n)$ to grammar node $m(n')$ starting at time frame t and ending at time frame t' (with $t > t'$ due to the *time-reversed* search). All likelihood scores used here are log likelihoods. Therefore they are *added* when partial paths are combined.

Different from a typical A* tree search, where the score for the incomplete portion of

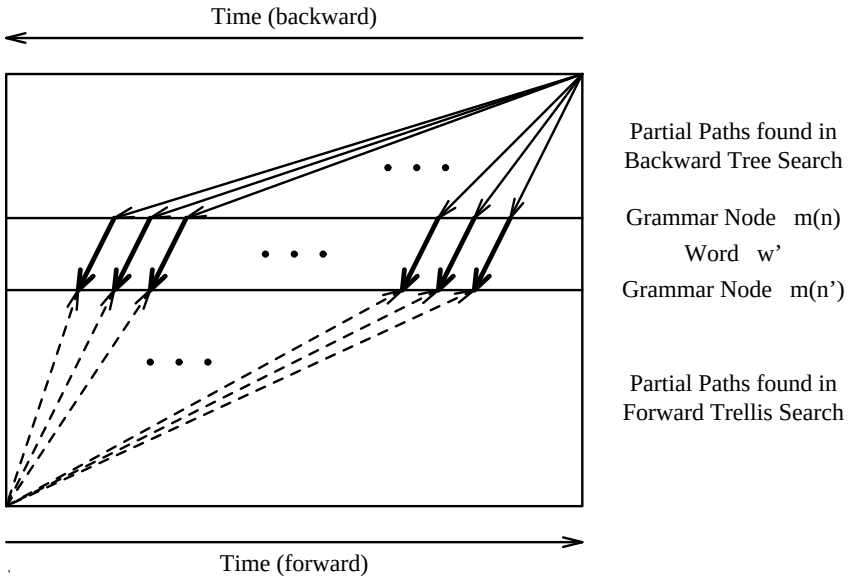


Figure 3.5: The tree-trellis algorithm: Extending the backward partial paths to grammar node $m(n)$ by the best word w' and merging these extended partial paths with the forward partial paths at grammar node $m(n')$.

a path is estimated by a heuristic function $\hat{h}(n)$, the tree search here uses the exact score $h(n)$ stored in the rank-ordered predecessor lists generated in the forward Viterbi search. Thus, maximum optimality of the A* tree search is obtained, which means that only the necessary path extensions are performed. Additionally, the evaluation function $f(n) = g(n) + h(n)$ gives the exact score of the complete path during the whole A* tree search for a hypothesis. Therefore, it is possible to limit the size of the OPEN stack to the number N of hypotheses to be found. The fact that *scores* instead of *costs* are used here, only means that not the node with the lowest costs but the one with the highest score is expanded first by the A* algorithm. Another difference to the normal A* tree search is that here a node is only expanded to its best ungrown successor node and not to all of its successor nodes.

A good summary of the theory behind the tree-trellis algorithm can be found in [1, pp. 236–238]. In [12], a modified version of the tree-trellis algorithm is presented, where a simple grammar is used in the forward Viterbi search and a more complex grammar is used in the backward tree search. This concept is similar to the forward-backward algorithm mentioned in Section 3.3.1 and can result in reduced computation requirements. More details about the implementation of the tree-trellis algorithm can be found in Section 3.4.

3.3.3 Approximate N-Best Algorithms

The exact N-best algorithm and the tree-trellis algorithm require both a significant amount of computation additional to that of a normal Viterbi search. Due to this reason, also faster N-best algorithms based on approximations have be suggested. These algorithms don't guarantee that the exact list of N-best hypotheses will be found. It can either happen that the likelihood score of an entry is underestimated or that an entry is missing totally. But the approximate list still might be sufficient for many applications. Two N-best algorithms using different approximations are described now.

/***** initialise start node s *****/
$w(s,i) = p(m(s),i), \quad i = 1,2,\ldots,J(m(s))$
$f(s,i) = h(m(s),T,i), \quad i = 1,2,\ldots,J(m(s))$
$I(s) = J(m(s))$
$i_{\text{next}}(s) = 1$
$g(s,t) = -\infty, \quad t = 0,1,\ldots,T-1$
$g(s,T) = 0$
$n_{\text{pre}}(s) = \text{NIL}$
put start node s on OPEN
$N_{\text{found}} = 0$
while OPEN is not empty
    remove from OPEN the top entry n (having maximum $f(n, i_{\text{next}}(n))$)
    take the single word extension $w' = w(n, i_{\text{next}}(n))$ to the best ungrown successor n'
    increment $i_{\text{next}}(n)$
    if $i_{\text{next}}(n) > I(n)$
        put n on CLOSED
    else
        reinsert n in OPEN according to $f(n, i_{\text{next}}(n))$
    if n' is a goal node (i.e. $m(n')$ is network ENTER node)
        increment N_{found}
        obtain the N_{found}'th hypothesis by tracing back through the pointers $n_{\text{pre}}(n)$
        if $N_{\text{found}} = N$
            exit
    else
        /***** grow successor node n' by expanding n by the single word w' *****/
        /* do time-reverse Viterbi search for word w' from $m(n)$ to $m(n')$ */
        $g(n',t') = max_{t=t'+1,t'+2,\ldots,T}(g(n,t) + c(n,n',t,t')), \quad t' = 0,1,\ldots,T-1$
        $g(n',T) = -\infty$
        /* merge backward partial paths to $m(n')$ with all predecessors */
        $w(n',i) = p(m(n'),i), \quad i = 1,2,\ldots,J(m(n'))$
        $f(n',i) = max_{t=0,1,\ldots,T}(g(n',t) + h(m(n'),t,i)), \quad i = 1,2,\ldots,J(m(n'))$
        $I(n') = J(m(n'))$
        sort $f(n',i)$ and $w(n',i)$ so that $f(n',i) \geq f(n',j)$ for $1 \leq i < j \leq I(n')$
        $i_{\text{next}}(n') = 1$
        provide backpointer $n_{\text{pre}}(n') = n$
        insert n' in OPEN according to $f(n', i_{\text{next}}(n'))$
exit with failure

Figure 3.6: The backward tree search algorithm to find the N-best hypotheses.

3.3.3.1 The Lattice Algorithm

The lattice N-best algorithm was proposed in [8]. It is based on a normal time-synchronous forward Viterbi search but differs in the backpointer information stored during the search. At each grammar node for each time frame, not only the best scoring word but all words that arrive at that node are stored in a traceback list together with their scores and the time when the word started. Instead of storing all arriving words, it is also possible to store only the best N_{local} words (local theories). Like in a normal Viterbi search, only the score of the best word is passed on as a basis for further scoring together with a pointer to the traceback list stored. At the end of the utterance, a simple tree search is used to step through the stored traceback lists and obtain the N-best complete sentence hypotheses sequentially. This tree search requires nearly no computation and can be performed very fast. The N-best theory traceback done by the tree search is illustrated in Figure 3.7, where each dot marks the beginning of a word and each dashed vertical line represents a stored traceback list.

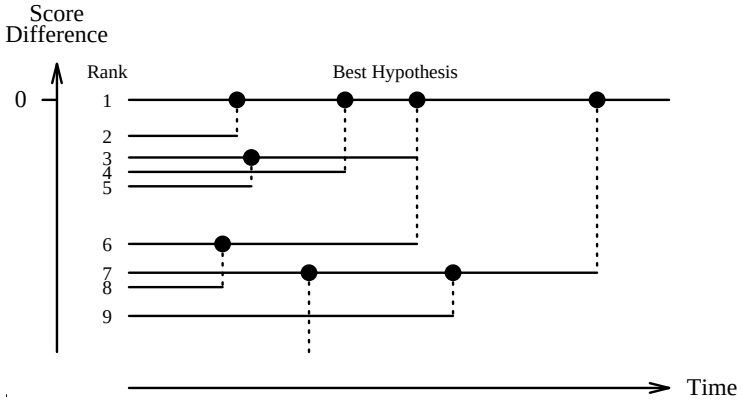


Figure 3.7: N-best theory traceback in the lattice algorithm (after [8]).

As described in [5], the lattice algorithm can also easily be implemented using a slightly extended token passing concept, where not only the single best but a list of best predecessors is stored in each phone link record. The lattice algorithm can be used either with *total likelihood scoring* [8] or like a normal Viterbi search with *maximum likelihood scoring* [5]. The difference between these two scoring principles is explained in Section 3.3.1.

A serious disadvantage of the lattice algorithm is that it underestimates or completely misses high scoring hypotheses due to the fact that all (expect the best) hypotheses are derived from segmentations found for other higher scoring hypotheses. This is caused by the assumption that the starting time of a word does not depend on the preceding word — an assumption that is inherent to the lattice algorithm. This problem can mostly be overcome by the word-dependent algorithm presented next.

3.3.3.2 The Word-Dependent Algorithm

Like the lattice N-best algorithm, also the word-dependent algorithm was proposed in [8]. It is a compromise between the exact N-best algorithm (with can be called a sentence-dependent algorithm) and the lattice algorithm. Here it is assumed that the starting time of a word *does* depend on the preceding word but *does not* depend on any word before that.

Therefore, theories are distinguished if they differ in the previous word. The advantages of this concept, compared with the lattice algorithm, are illustrated in Figure 3.8.

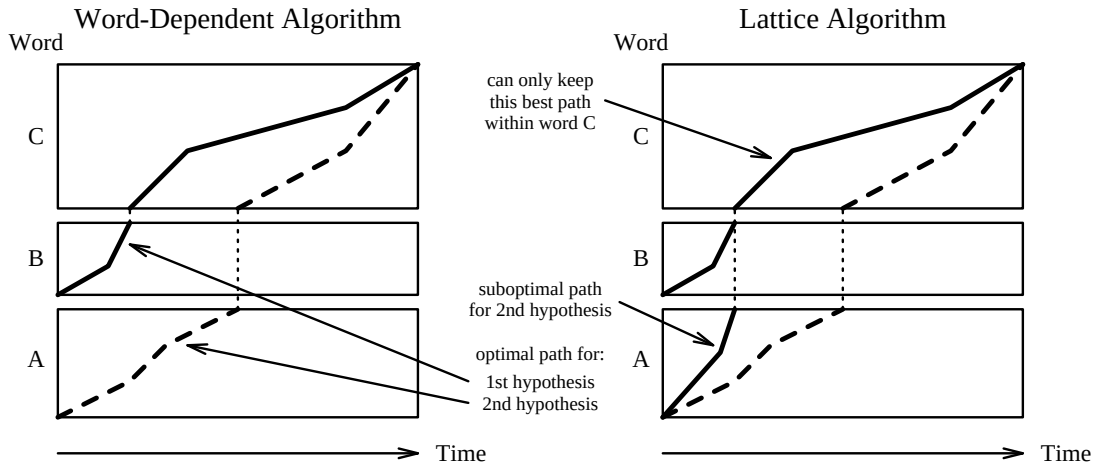


Figure 3.8: Comparison of the word-dependent algorithm and the lattice algorithm (after [8]).

Within a word, the likelihood scores for each of the different local theories (previous words) are preserved. At the end of each word, the likelihood score for each previous word is recorded along with the name of the previous word. Then a single theory with the name of the word that just ended is used to proceed. At the end of the utterance, a tree search (similar to the one used in the lattice algorithm) is used to obtain the list of the N most likely hypotheses. To reduce the computation requirements, the number N_{local} of theories kept locally (i.e. within a word) should be limited. Typical values for N_{local} range from 3 to 6. Like the lattice algorithm, also the word-dependent algorithm can be used either with *total likelihood scoring* or with *maximum likelihood scoring*.

3.3.4 Comparison of the Different N-Best Algorithms

All the N-best algorithms presented here have different features and disadvantages. Therefore, the optimal algorithm for a given task should be selected according to the requirements of that task. The main features of the different algorithms are summarised now:

- **Exact N-Best Algorithm:** This algorithm requires significantly more computation that a normal Viterbi search. It allows to use also *total likelihood scoring* instead of *maximum likelihood scoring*.
- **Tree-Trellis Algorithm:** This algorithm requires only somewhat more computation than a normal Viterbi search. It finds the exact list of N-best hypotheses and is recommended if N is not large, since the computation for the backward tree search is proportional to N.
- **Lattice Algorithm:** This algorithm is the fastest and requires only a little bit more computation than a normal Viterbi search. It might miss or underestimate several hypotheses but can be used easily with large values for N.
- **Word-Dependent Algorithm:** This algorithm is a compromise between the exact and the lattice algorithm. It requires less computation than the exact algorithm while

still generating a quite accurate list of N-best hypotheses [8]. Also this algorithm can be used easily with large values for N.

These N-best algorithms (except for the tree-trellis algorithm) can be used either with *maximum likelihood scoring* (like the Viterbi algorithm) or with *total likelihood scoring*. The latter technique leads to somewhat higher recognition rates [8].

3.4 Implementation of the Tree-Trellis Algorithm

Much of the effort that went into this thesis was spend on the implementation and optimisation of the new HTK tool HViteN. This program is based on HTK's Viterbi recogniser HVite and implements the tree-trellis N-best algorithm. All options offered by the original HVite program for the forward Viterbi search have also been implemented in the backward N-best search. Several new options are provided to control the operation of the tree-trellis algorithm. Section A.1.1 contains the user manual for HViteN.

The modifications of HVite's original forward Viterbi search, the implementation of the backward tree search and optimisations done to reduce memory and computation requirements are described in the next sections. Contrary to the tree-trellis algorithm described in Section 3.3.2, the grammar network used by HTK does not directly provide the necessary grammar nodes. A node in the HTK network corresponds to a PhoneInstance (i.e. an HMM or a word-begin or word-end node) and not to a grammar node between two words. Therefore, the predecessor LinkSets of the word-begin or word-external nodes are used to hold pointers to their rank-ordered predecessor lists generated during the forward search.

The basic backward tree search of the tree-trellis algorithm is presented in Figure 3.6. Figure 3.9 gives a more detailed illustration of the process of extending the backward partial paths and merging it with the forward partial paths, which is also shown in Figure 3.5. "TOS entry" refers here to the best single word extension of the backward path not yet performed.

3.4.1 Modifications in the Forward Trellis Search

The original implementation of the forward Viterbi algorithm based on the token passing paradigm is fully included in HViteN. The generation of the rank-ordered predecessor lists is added to that algorithm. This modification mainly concerns the function PropagateExitTokens which propagate tokens from word-end or word-external nodes and is executed once per time frame. At the end of PropagateExitTokens, all the predecessor LinkSets that belong to a word-begin or word-external nodes which got a new token passed are processed. For each of these LinkSets, the scores of the tokens of the active predecessors are collected in a preliminary list together with pointers to the predecessor nodes. "Active" means that the PhoneInstance which owns that token is within the beam width of the pruned Viterbi search. If no bigrammar transition probabilities are used, the transition log likelihoods are added to the scores in the preliminary list and the list is sorted according to these new scores. Then, the top N entries are stored as the rank-ordered predecessor list for that predecessor LinkSet and time frame (N is the number of hypotheses that should be generated later). In Figure 3.6, these predecessor lists were denoted by

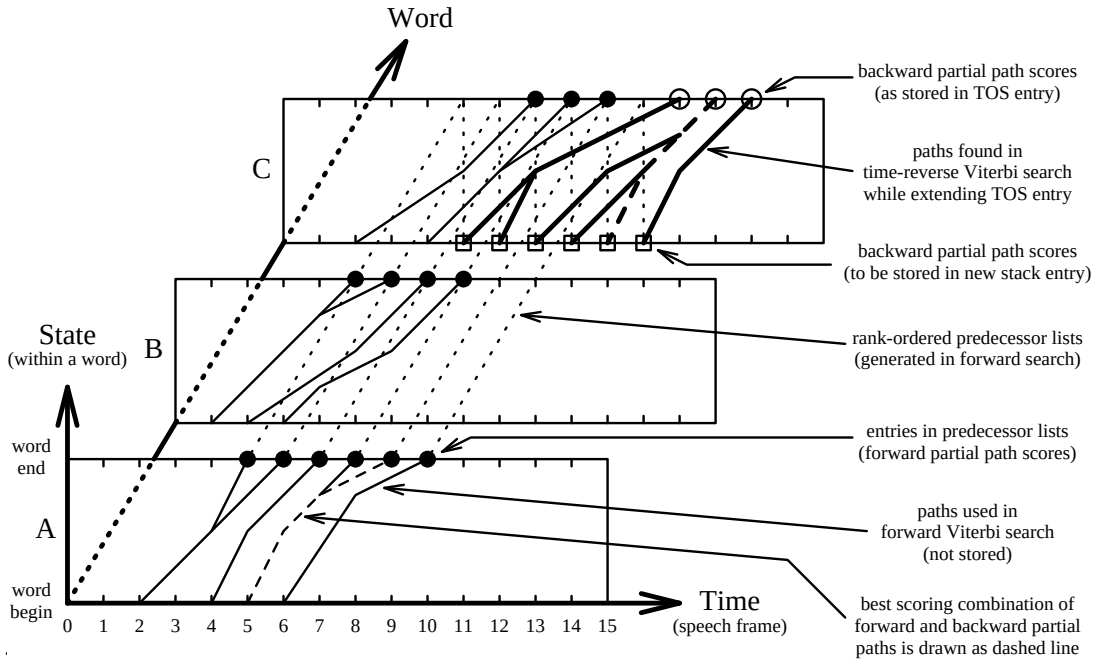


Figure 3.9: The tree-trellis algorithm: Extending the backward partial paths by the best word C and merging these extended partial paths with the forward partial paths.

$h(m(n), t, j)$ and $p(m(n), j)$. If bigrammar transition probabilities are used, the transition log likelihood can't be added since it depends on the succeeding node (i.e. word) which is not yet known. Therefore, the predecessor list can't be ordered and all its entries (not only the top N) have to be stored.

When a network file is loaded by HTK, an internal representation of the network is generated. To minimise memory requirements, HTK tries that those nodes having the same set of successors or predecessors also share their successor or predecessor LinkSets (see Figure 2.9). Since this concept minimises the number of predecessor LinkSets, also the number of rank-ordered predecessor lists (RankLists) is minimised.

The other important modification in the forward Viterbi search concerns the HMM state output probabilities $b_j(o_t)$. These probabilities are calculated for all HMMs that are active at the time frame being processed. Since the calculation of these probabilities requires a significant amount of computation, they are stored for each time frame. Thus, it is not necessary to recalculate them for the time-reverse Viterbi searches performed during the backward tree search.

Other minor changes to the forward Viterbi algorithm don't affect its basic functioning and are therefore not reported here. The main data structure used by HViteM is illustrated in Figure 3.10. nSamples is the number of time frames in the utterance being processed and numStates the number of states in an HMM. Further data structures that e.g. allow to step easily through all existing Nodes, HMMs, LinkSets, etc. are not included in this figure. It should be noted that memory is allocated only for that amount of data that actually will be stored in a data structure.

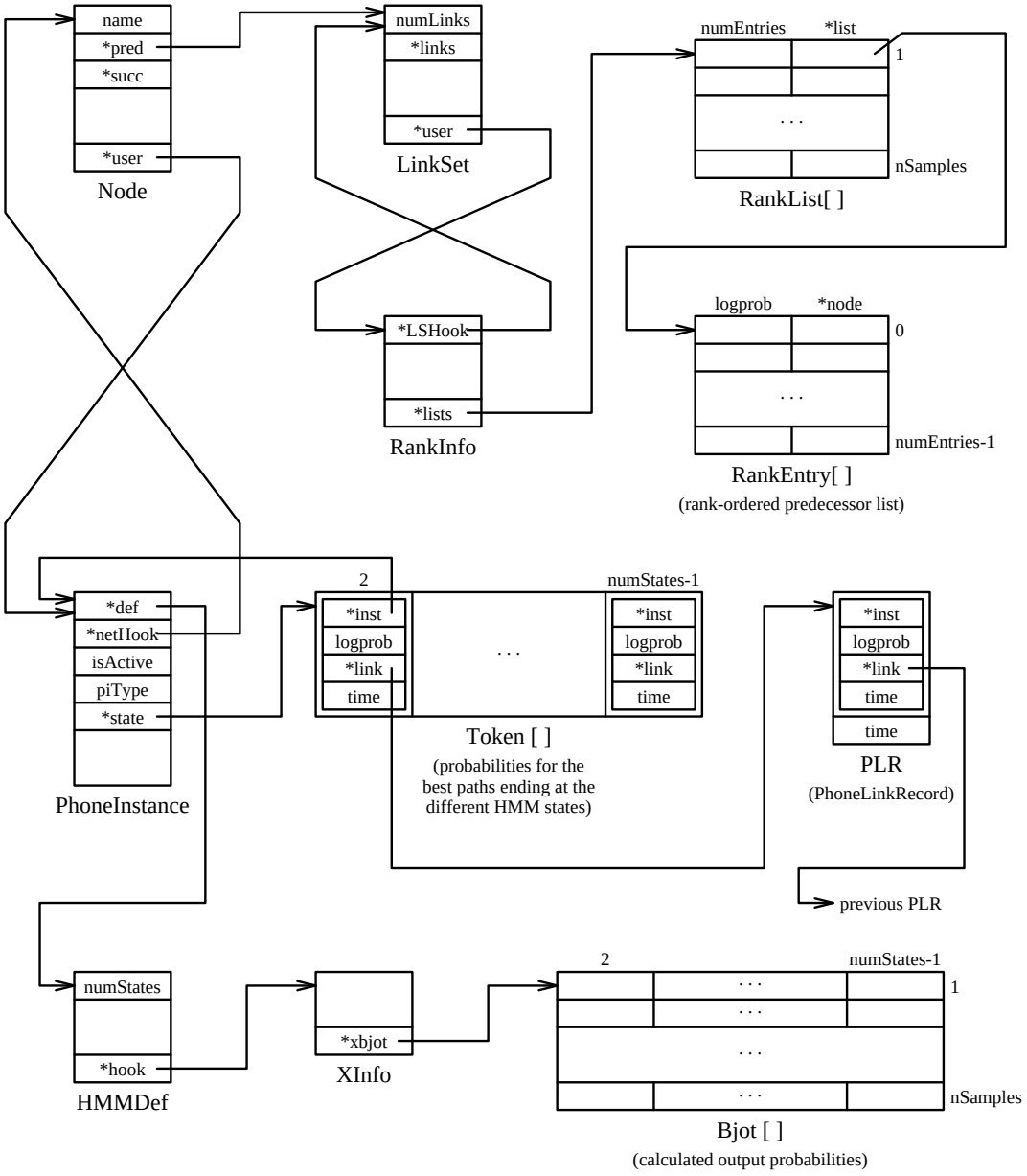


Figure 3.10: The main data structure in HViteN (simplified).

3.4.2 Implementation of the Backward Tree Search

The backward tree search is based on the algorithm shown in Figure 3.6. The OPEN and CLOSED stack are each implemented as a chained list of stack entries. The complete data structure of theses stacks is illustrated in Figure 3.11. Each stack entry represents a node n in the backward search tree and corresponds to the distinct word sequence (partial hypothesis) of the backward partial paths stored in the stack entry. Even though the grammar is represented by a network (and thus can include "loops"), the backward search really is based on a tree with the nodes corresponding to partial hypotheses (and not to words, like in the grammar network).

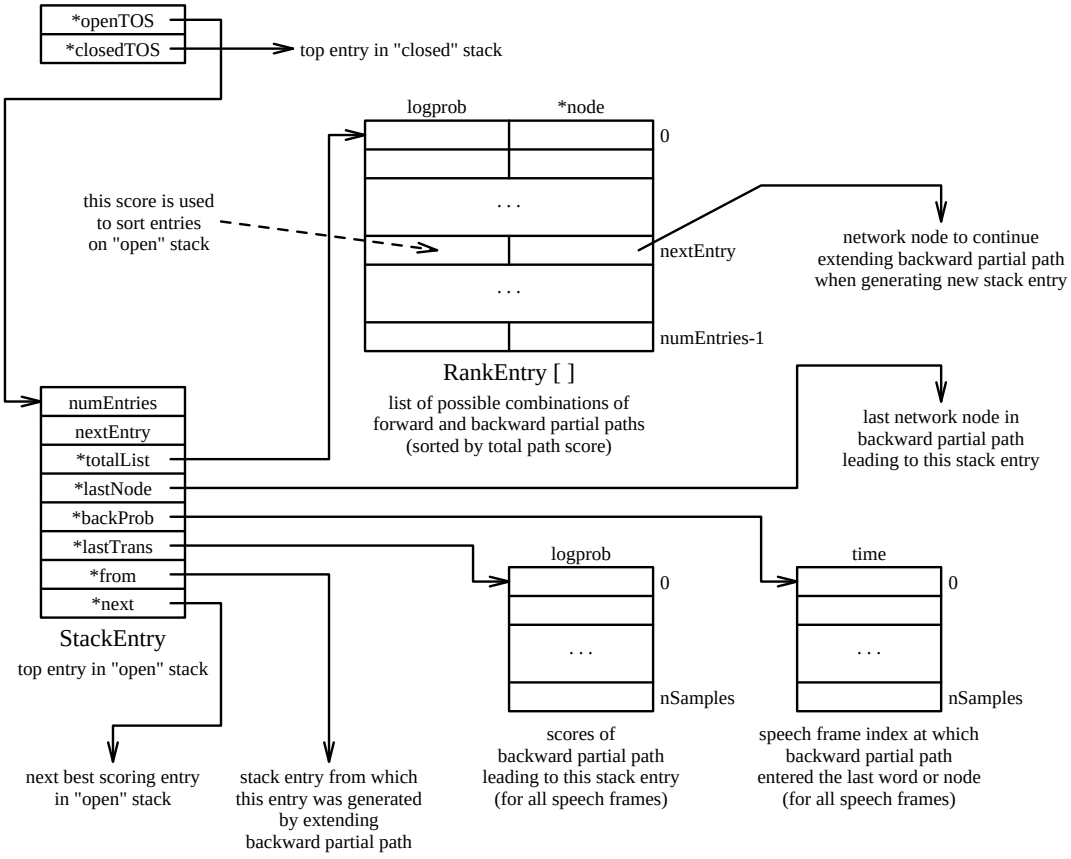


Figure 3.11: The complete data structure of the stack in HViteN.

Nearly all components of a StackEntry correspond directly to the different variables for a node n that were used in Figure 3.6. $I(n)$ is called numEntries, $i_{\text{next}}(n)$ is called nextEntry (to be exact, $i_{\text{next}}(n) - 1$ is stored here) and totalList points to the array containing $f(n,i)$ and $w(n,i)$. The best word extensions are stored in $w(n,i)$ as pointers to their network nodes. backProb points to the array containing $g(n,t)$, while the array pointed to by lastTrans is only needed to be able to obtain the time-alignment when a complete hypothesis was found. The backpointer to the previous stack entry is called from and the pointer next is used to chain the entries in a stack. The pointer lastNode points to the network node of the word that was added to the (backward) partial hypothesis when this stack entry was generated. T, the number of speech frames in the utterance, is called nSamples here.

The backward tree search is implemented as outlined in Figure 3.6. First, an initial entry is

generated and put on the OPEN stack. Then, the best stack entry n in OPEN is expanded by its best single word extension w' and thus the new stack entry n' is generated. This extension process is repeated iteratively until all N-best hypotheses have been found or until the OPEN stack is empty.

The time-reverse Viterbi search used to extend the backward partial paths by the best single word extension w' is a complex part in this implementation. Like the original forward search, also this backward Viterbi search is based on the token passing paradigm. It is implemented similar to the forward search and also includes an optional global pruning threshold to perform a beam search. For each time frame t, a token with the likelihood score $g(n,t)$ is injected to the end of the word w' and the token reaching the beginning of the word is stored in $g(n',t)$ (array backProb). The speech frame at which it was injected in the word is stored in the array lastTrans. Thus, an explicit calculation of the likelihood scores $c(n,n',t,t')$ is not required. The term "word" denotes here either a single word-external node in the network or the complete subnet of word-internal nodes including the word-end and word-begin nodes of that subnet. The same transition probabilities for inter-node transitions are used as in the original forward Viterbi search.

For each time frame t, directly after $g(n',t)$ was calculated, the backward partial path ending at t is merged with the forward partial paths stored in the rank-ordered predecessor list ($h(m(n),t,j)$ and $p(m(n),j)$) generated during the forward search. At this point, also the transition probabilities for the bigrammar case can be added since now also the successor of the last word in the forward partial path is known (the word w'). For each possible predecessor of w', the highest combined likelihood score for a complete path is determined after all time frames have been processed. This list of complete path scores for the different predecessors is sorted and thus $f(n',i)$ and $w(n',i)$ (array totalList) for the new stack entry n' are found.

3.4.3 Optimisation of the Implemented Algorithm

The new HTK tool HViteN, which implements the tree-trellis N-best algorithm, was otimised in different ways to increase performance and reduce memory requirements. These optimisations will be explained now.

HViteN works correct with arbitrary grammar networks. But the amount of memory required for the rank-ordered predecessor lists depends heavily on the structure and size of this network. The different basic structures of a grammar network are presented in Figure 3.12. They are named after different concepts for continuous speech recogniser that can't use an arbitrary grammar network [1, 5].

It is obvious, that the "one pass" network has the lowest total number of predecessors (4 and 3 from EXIT). If length constraints are required, "level building" like networks have to be used. In this example, legal sentences can have 1, 2 or 3 words. The "level building" networks have a significant higher total number of predecessors (1+3+3 and 9 from EXIT for Version A; 1+4+4 and 3 from EXIT for Version B). If EXIT is the global exit node of the grammar network, the corresponding rank-ordered predecessor list is only needed for the last time frame T while non of the other predecessor lists is needed for T. This effect is used in HViteN to store only the required predecessor lists and nearly halves the predecessor list memory requirements for a "one pass" network. Whether version A or version B of the "level building" networks requires less memory for the predecessor lists

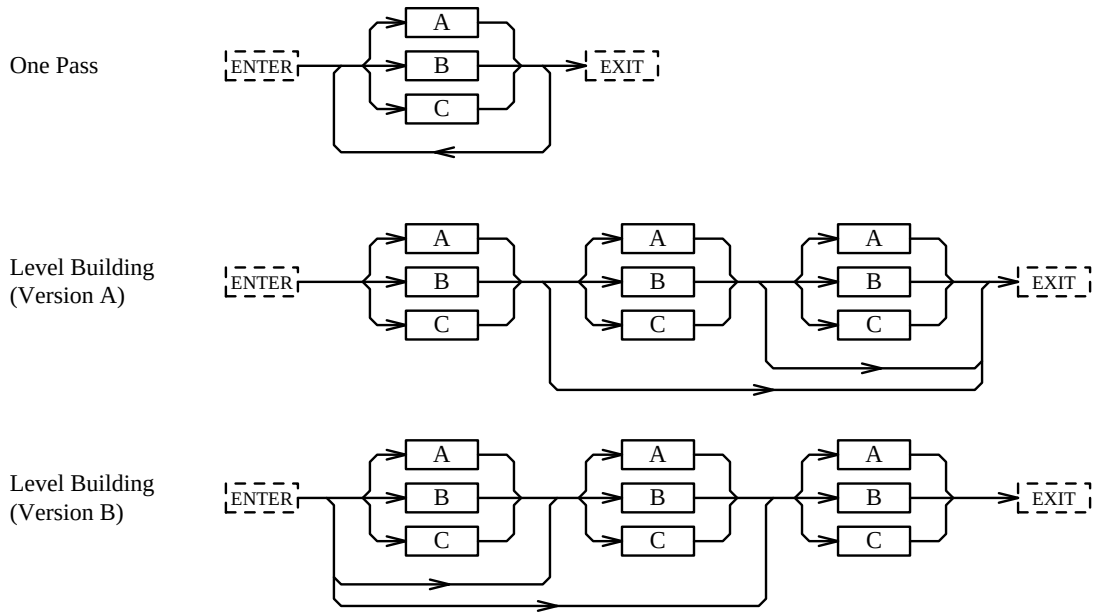


Figure 3.12: Basic structures of a grammar network.

depends on fact whether EXIT is the global network exit node or not.

HViteN offers also an option to specify the maximum number of entries in a rank-ordered predecessor list manually. The size of the predecessor lists also depends on the beam width in the forward search, since only active PhoneInstances lead to an entry in this list. The search beam width can be controlled by a global pruning threshold. To maintain an exact N-best search, all entries in a predecessor list have to be stored if bigrammar transition probabilities are used.

The combined path scores $f(n,i)$ used to order the stack entries are the exact scores of the full paths. Therefore, it is possible to limit the number of entries on the OPEN stack to the number of hypotheses that are still to be found. The entries, that are removed from OPEN in this process, are put on the CLOSED stack, since they might have backpointers pointing to them. The memory for the arrays totalList and backProb is freed when an entry is put on the CLOSED stack, since that information is not required any longer. If the time-alignment of the generated hypotheses is not required, it is not necessary to allocate memory for the lastTrans array in a stack entry. A special option in HViteN can be used to select this mode.

Since the combined path scores $f(n,i)$ are exact scores, a newly generated stack entry has to have the same score as the stack entry it was generated from using the best single word extension. If this is not the case, the beam width of a pruned search must have been to small and a warning is issued. If the new combined score is lower than the old one, the beam width of the time-reverse Viterbi search probably was too small. Otherwise, the forward Viterbi search beam width probably was too small. These warnings simplify the process of choosing reasonable values for the pruning thresholds. If such a pruning warning has be issued, it often happens that the hypotheses, which are found after the warning was issued, are not any longer ordered according to their likelihood score. Such heavy pruning causes the tree-trellis algorithm to loose its exactness.

Normally, the likelihood scores in the rank-ordered predecessor lists and in the arrays

totalList and backProb of a StackEntry are stored as double (8 byte) and the times in the lastTrans array of a StackEntry are stored as int (4 byte). A compile time switch in HViteN allows to compile a special version (HViteNm) that uses float (4 byte) and short (2 byte) instead of double and int for these arrays. The now slightly reduced accuracy does not make any difference in practice, while the memory requirements are significantly reduced (ca. 30% for the predecessor lists).

3.5 Experimental Results

Several recognition tests under different circumstances were conducted to investigate the performance of the new tree-trellis N-best recogniser HViteN.

In first tests, the output of the tree-trellis recogniser was compared with results from an earlier and much simpler implementation of this algorithm [21]. The same small set of 5 HMMs, the same fixed-length level building grammar and the same parameters as in the earlier tests were used. The first 20 hypotheses for 3 different utterances, which were given in [21], were exactly the same as those generated by HViteN.

The further recognition tests were based on the 1000 word DARPA resource management (RM) corpus and used the HMM sets and grammar networks described in Section 2.3.3. For all recognition tests reported in this chapter, the original RMTK lexicon of phoneme transcriptions for the words in the RM vocabulary was used to generate the grammar networks for the recogniser. The results of these tests are described in Section 3.5.1. In another thesis, a word-dependent N-best algorithm was implemented [14]. In Section 3.5.2, results of recognition tests with these both implementations are compared.

Also the computation and memory requirements were examined in recognition tests on the RM corpus. The following results were obtained using a *SUN SPARCstation IPX* with 32 Mbyte RAM. The word-pair grammar and the HMM set "base" were used with a pruning threshold of 200 in the forward as well as in the backward search. 10 utterances with an average length of 2.85 sec were used here. They were randomly selected from the "feb89" test set. The measured average durations per utterance for the recognition process are shown in Table 3.1.

Recogniser Mode	CPU Time
forward Viterbi search only	32.7 sec (100%)
full tree-trellis search ($N = 1$)	48.7 sec (148%)
full tree-trellis search ($N = 10$)	58.4 sec (178%)

Table 3.1: Average CPU time per utterance for different recogniser modes.

Based on this data, the average durations of the different phases of the recognition process were calculated. They are shown in Table 3.2.

Table 3.2 shows, that using HViteN to find the best 15 hypotheses with the tree-trellis algorithm takes less than twice the time the original HVite needs to find the first hypotheses in the forward Viterbi search. It also illustrates, that a significant amount of computation is needed for the generation of the rank-ordered predecessor lists.

Additionally, about 125 sec were needed by HViteN to load the grammar network and the HMM definitions. This data is only loaded once and thereafter, an arbitrary number of

Recogniser Phase	CPU Time
forward Viterbi search	32.7 sec (100%)
rank-ordered predecessor list generation	14.9 sec (45%)
backward tree search (per hypothesis)	1.1 sec (3.3%)

Table 3.2: Average CPU time per utterance for the different phases in the recognition process.

utterances can be processed by HViteN.

The average memory requirements for the different data structures in HViteN are listed in Table 3.3. It shows, that a large amount of memory is required to store the rank-ordered predecessor lists generated in the forward search. The memory requirement for these predecessor lists is reduced by ca. 30% if HViteNm is used instead of HViteN.

Data Structure	Size
Network	690 Kbyte
PhoneInstances	1100 Kbyte
Bjot (output probabilities)	210 Kbyte
RankLists ($N = 1$)	2340 Kbyte
RankLists ($N = 10$)	5380 Kbyte
OPEN stack ($N = 10$)	60 Kbyte
CLOSED stack ($N = 10$)	70 Kbyte

Table 3.3: Average memory requirements for the different data structures in HViteN.

3.5.1 Recognition Tests with the Tree-Trellis Algorithm

To examine the performance of the new tree-trellis N-best recogniser HViteN, several recognition tests were performed. In Figure 3.13, Figure 3.14 and Figure 3.15, the time-aligned best 10 hypotheses are shown together with original speech signal for 3 different utterances from the test set "feb91". The HMM set "demo" and the word-pair grammar were used for this recognition test.

The example in Figure 3.13 shows, that the correct hypotheses on rank 8 is the first hypotheses in the list that is a correct and meaningful sentence. This illustrates the basic idea behind the N-best search paradigm. To implement this paradigm, a full grammar for the RM task could be used to find the first "legal" message automatically. Also the example in Figure 3.14 shows, that the correct hypotheses on rank 2 is the first correct sentence in this list. In the example in Figure 3.15, the correct hypotheses is on the first rank and thus was already found in the forward viterbi search.

All the three examples given here show some typical properties of an N-best hypotheses list. One typical property is, that the different N-best hypotheses for an utterance normally are quite similar. Most of the longer words are correctly recognised in all hypotheses and differences occur only at some points of the utterance and mainly concern short words. Another property is, that the average log likelihood scores (per time frame) are quite similar for the different hypotheses in an N-best list. For example in Figure 3.13, the score difference between the 1st and the 8th (correct) hypothesis is 0.233, which is only about 0.3% of the best score.

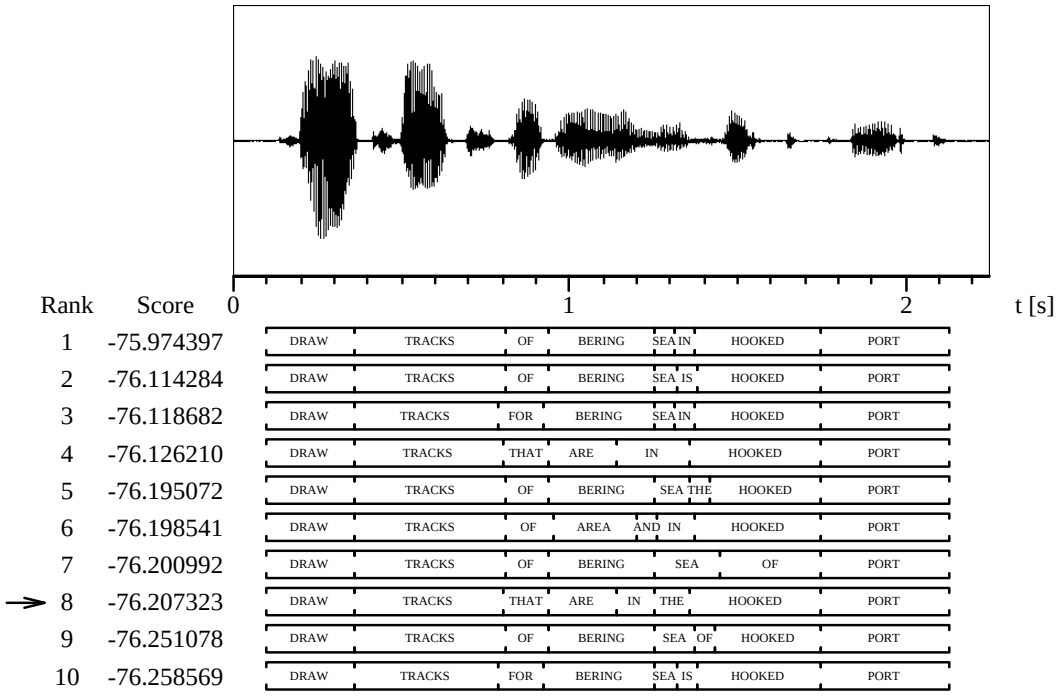


Figure 3.13: First 10 hypotheses for the utterance jwg0_5_st0580 from test set "feb91". The correct hypothesis (rank 8) is marked by an arrow (HMM set: "demo", grammar: word-pair).

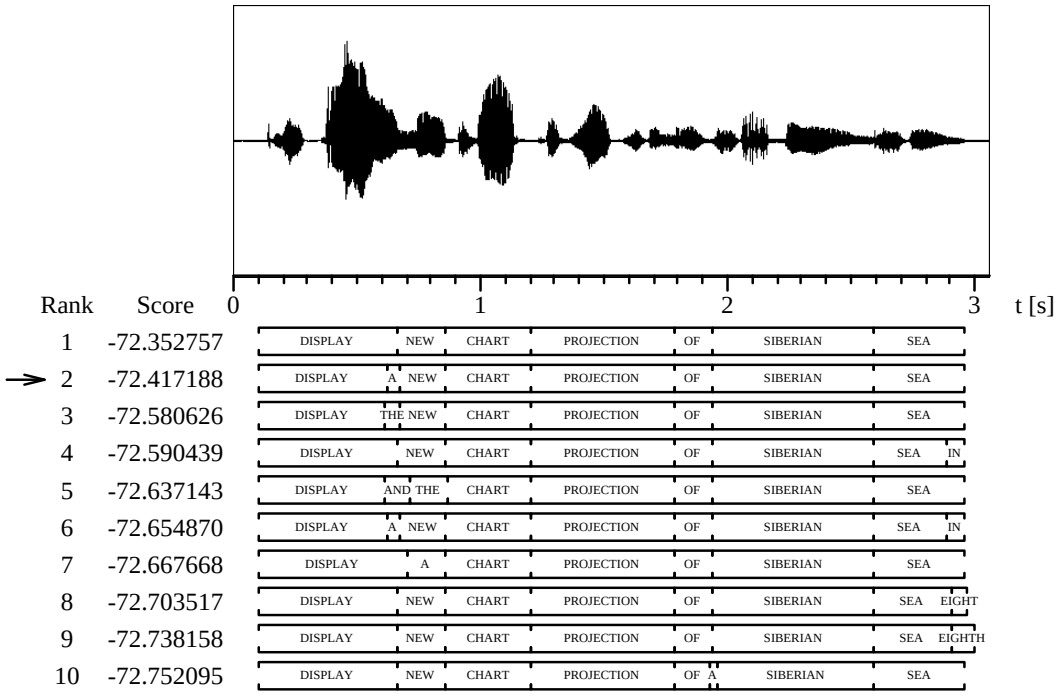


Figure 3.14: First 10 hypotheses for the utterance eac0_2_st1261 from test set "feb91". The correct hypothesis (rank 2) is marked by an arrow (HMM set: "demo", grammar: word-pair).

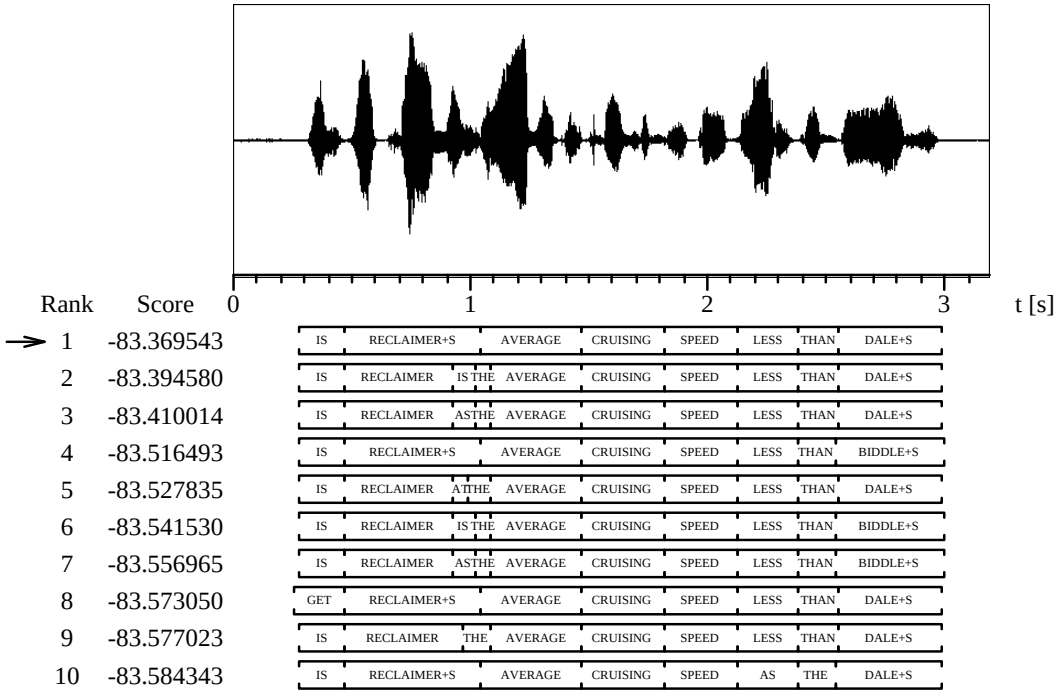


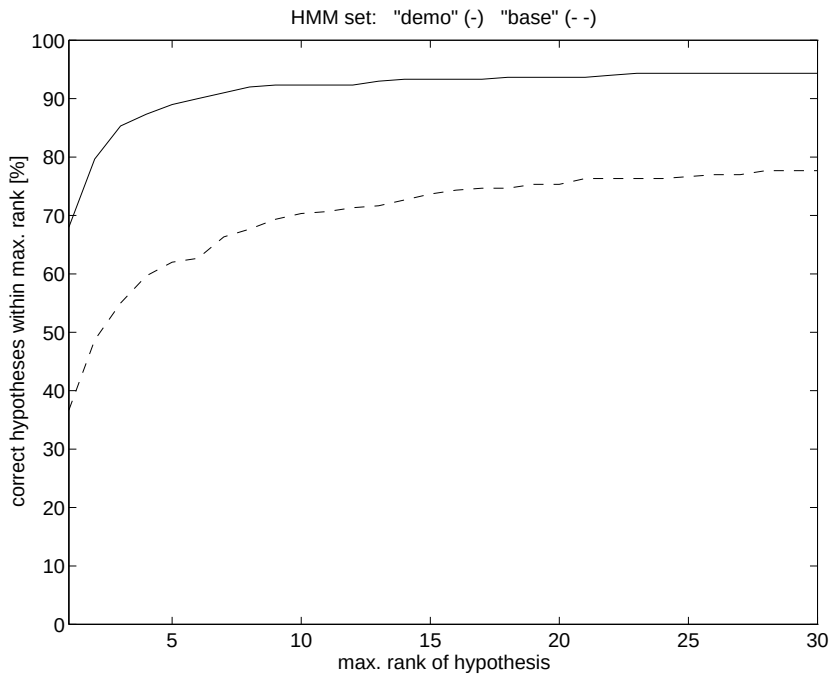
Figure 3.15: First 10 hypotheses for the utterance alk0_3_st0428 from test set "feb91". The correct hypothesis (rank 1) is marked by an arrow (HMM set: "demo", grammar: word-pair).

To compare the results of recognition tests with different HMM sets and grammars, the cumulative distribution of the rank of the correct hypothesis was investigated. In Figure 3.16, the two available HMM sets are compared. In both cases, the word-pair grammar was used and the 300 utterances in the test set "feb91" were tested. The figure shows, that the HMM set "demo" (about 2300 context dependent HMMs for the 47 phonemes plus 2 silence models) as expected offers a significantly higher recognition rate than the HMM set "base" (49 context independent HMMs for the 47 phonemes plus 2 silence models).

In Figure 3.17, results of recognition tests ar shown for the two HMM sets being used together with a simple network that does not include grammar constraints. As a reference, the cumulative distributions from Figure 3.16 are included here. This figure shows clearly the advantages of a search constrained by a grammar. The distributions shown in this graph also indicate, that a low initial recognition error rate (at the first hypothesis) results in a faster reduction of this error rate for a increasing number N of hypotheses that a higher initial error rate.

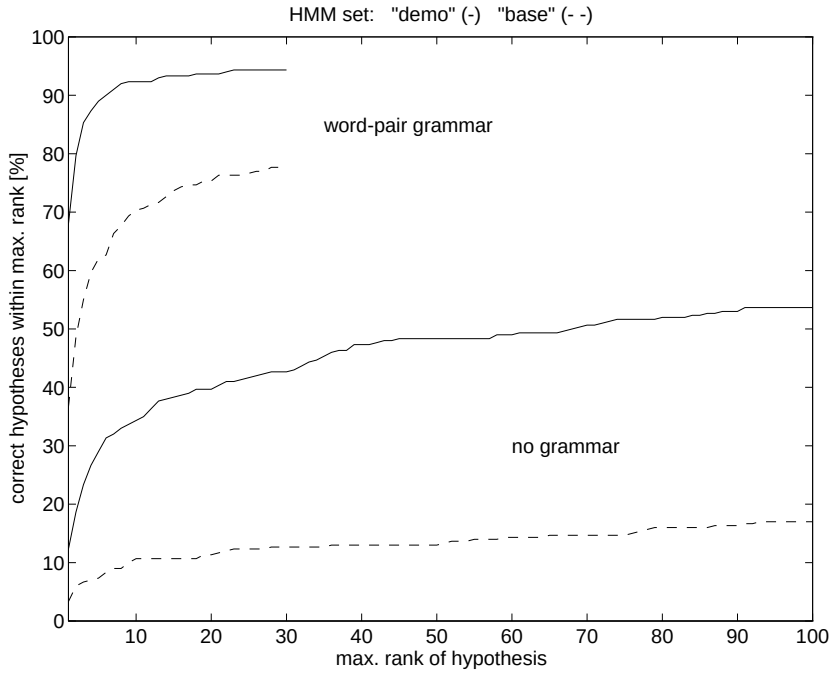
In Figure 3.18, the cumulative distribution of the rank of the correct hypotheses is compared for the four available test sets. The word-pair grammar and the HMM set "demo" was used for this tests. Also the average distribution for all test sets is included in this graph. The cumulative distributions for the four test sets are, as expected, quite similar.

The pruning thresholds for these recognition tests were chosen to make the beam width as wide as possible on the available hardware. Only for a few percent of the utterances, pruning warning messages were issued during the generation of the best 30 hypotheses. And except some very few exceptions (in Figure 3.18), all utterances in the test sets could be processed with the chosen pruning levels on the available hardware.



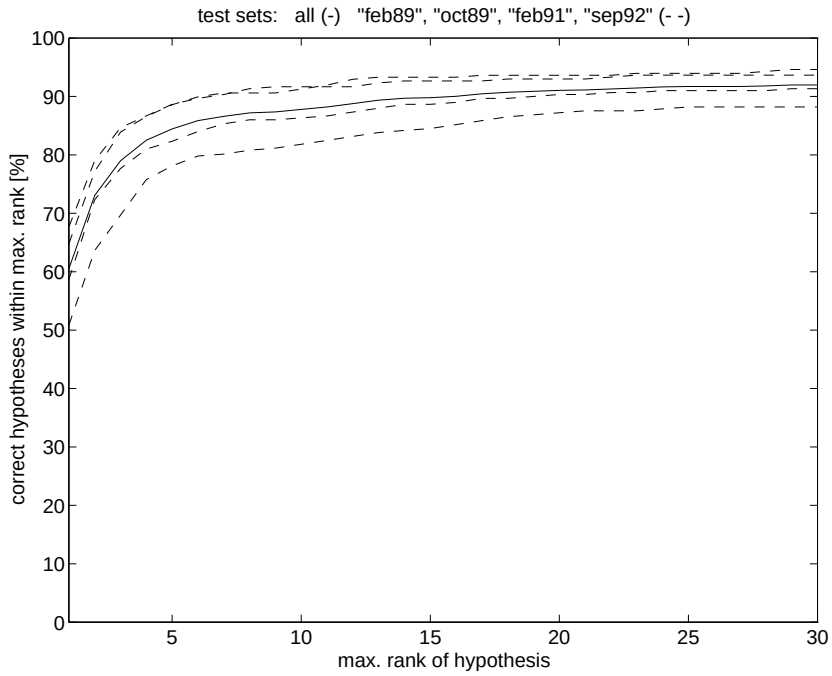
HMM set "demo": 68.0% (top 1), 94.3% (top 30) HMM set "base": 36.7% (top 1), 77.7% (top 30)

Figure 3.16: Recognition test results for the two different HMM sets "demo" and "base" (test set: "feb91" (300 utterances), grammar: word-pair).



HMM set "demo": 12.3% (top 1), 53.7% (top 100) HMM set "base": 3.3% (top 1), 17.0% (top 100)
(The results using the word-pair grammar are the same as in Figure 3.16)

Figure 3.17: Recognition test results using the word-pair grammar and no grammar for both HMM sets "demo" and "base" (test set: "feb91" (300 utterances)).



all test sets (total 1195 utterances used): 60.5% (top 1), 92.0% (top 30)
"feb89" (298 utt.): 64.8% (top 1), 94.6% (top 30) "oct89" (300 utt.): 58.7% (top 1), 91.3% (top 30)
"feb91" (300 utt.): 67.7% (top 1), 93.7% (top 30) "sep92" (297 utt.): 50.8% (top 1), 88.2% (top 30)

Figure 3.18: Recognition test results for the four different test sets (HMM set:"demo", grammar: word-pair, stronger pruning than in Figure 3.16).

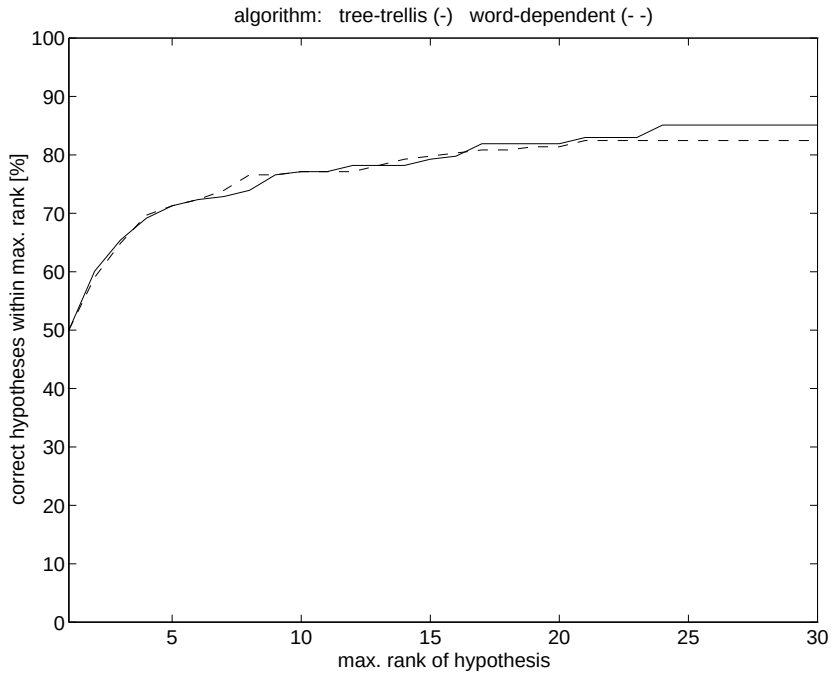
3.5.2 Comparison of the Tree-Trellis and the Word-Dependent Algorithm

In another thesis, a word-dependent N-best algorithm was implemented [14]. Also this implementation was based on HTK. Recognition test results for these both implementations are shown in Figure 3.19. In both cases, the word-pair grammar together with the HMM set "demo" was used. Test results for the word-dependent algorithm were only available for 188 of the 300 utterances in test set "sep92". To maintain comparability, the same set of 188 utterances were tested with the tree-trellis algorithm here. Although total likelihood scoring was used with the word-dependent algorithm, both algorithms give very similar results. The slightly lower performance of the word-dependent algorithm for large values of N (e.g. $N = 30$) is probably caused by the fact, that only 3 local theories in the word-dependent algorithm were used in this test.

3.6 Discussion

In this chapter, different N-best algorithms were described and compared. The implementation of the tree-trellis N-best algorithm based on HTK's original Viterbi recogniser was explained and recognition test results were presented.

This new N-best recogniser HViteN has shown to be a versatile and efficient new tool for the HTK system. Due to the large memory requirements of the tree-trellis algorithm, the implementation was optimised in different ways. Recognition tests on the RM corpus using



"tree-trellis": 50.0% (top 1), 85.1% (top 30) "word-dependent": 50.0% (top 1), 82.4% (top 30)

Figure 3.19: Comparison of the tree-trellis algorithm and the word-dependent algorithm (test set: "sep92" (188 utterances used), HMM set: "demo", grammar: word-pair).

the word-pair grammar and the advanced HMM set "demo" were possible and gave the expected results. But the generation of $N = 30$ hypotheses required nearly all memory resources of the available hardware when typical values for the pruning threshold were used. The largest part of the memory was required for the rank-ordered predecessor lists generated during the forward search. This shows, that it is worth to spend the significant amount of computation needed to generate and sorting these lists, since the rank-ordering allows to limit the size of a list to N entries.

An option is provided to limit the size of these lists even further, but then the correctness of this algorithm is not any longer guaranteed. Nevertheless, the probability, that this approximation causes errors in the generated N-best list, can be very low. The decision, to which degree such errors can be tolerated, depends very much on the specific application. Therefore, the effects of such approximations were not examined in detail in this theses.

Chapter 4

Automatic Lexicon Generation

In this chapter, concepts for the automatic generation of the lexicon required by a sub-word based speech recogniser are presented. A modified version of the tree-trellis algorithm is used as the base for the automatic lexicon generation techniques studied here. Its implementation is described and different extensions to the lexicon generation process are explained. Finally, the performance of different automatically generated lexica is examined.

4.1 Introduction

In a sub-word based speech recogniser, a lexicon containing the transcriptions in terms of basic recognition units for each word in the vocabulary is required. This lexicon is normally generated using a pronunciation dictionary or by an experienced phonetician. Thus, it is an important exception from the concept, that the parameters specifying an HMM based recogniser are optimised in training procedures.

Different methods for the automatic generation of lexicon entries have been proposed [15, 16, 17, 18, 19]. These methods try to finding the optimal lexicon entry for a word. The spelling of the word as well as training utterances of the word can be utilised in this process. Commonly, the most likely transcription $\hat{W}$ of a word is assumed to be the optimal entry for that word. The process of finding this transcription can be summarised as

$$\hat{W} = \arg\max_{W} P[W|O,S], \tag{4.1}$$

where S is the spelling of the word and O denotes observed training utterances of the word. Using Bayes' rule, $P[W|O,S]$ can be rewritten as

$$P[W|O,S] = \frac{P[W,O,S]}{P[O,S]} = \frac{P[O|W,S]\,P[W|S]\,P[S]}{P[O,S]}. \tag{4.2}$$

It is common to assume, that the spelling S and the observed utterances O independently contribute information about the most likely transcription $\hat{W}$. With S and O being statistically independent now, Equation 4.2 can be written as

$$P[W|O,S] = \frac{P[O|W]\,P[W|S]\,P[S]}{P[O]\,P[S]} = \frac{P[O|W]\,P[W|S]}{P[O]}. \tag{4.3}$$

Since $P[O]$ does not depend on W, Equation 4.1 can now be written as

$$\hat{W} = \arg\max_{W} P[O|W]\, P[W|S]. \tag{4.4}$$

Different methods can be used to find the probability $P[W|S]$ of a transcription given the spelling of the word. In [16, 18], probablistic spelling-to-sound rules were automatically derived using an available lexicon of pronunciations. In [19], the text-to-sound rules of the DECtalk speech synthesiser were used together with phoneme confusion matrix. In this thesis, the spelling of a word is not taken into account in the automatic generation of a new lexicon entry. Thus, $P[W|S]$ does not depend on W and can therefore be ignored in the maximisation in Equation 4.4.

A varying number K of training utterances of the word can be taken into account in the lexicon generation process. If no utterances ($K = 0$) are utilised, $P[O|W]$ is 1 and can be ignored in Equation 4.4. Then, the new transcription depends only on the word's spelling. If a set of K training utterances O_k ($k = 1, 2, \ldots, K$) is utilised, the probability $P[O|W]$ can be calculated as

$$P[O|W] = \prod_{k=1}^{K} P[O_k|W] \tag{4.5}$$

The probabilities $P[O_k|W]$ can be calculated using an HMM based speech recogniser. Since the log likelihood $\log P[O_k]$ of an observation O_k is in average proportional to the length T_k of that observation, also another way to combine the probabilities $P[O_k|W]$ was investigated in this thesis. The combined log likelihood score

$$\log P_r[O|W] = \left(\sum_{k=1}^{K} T_k\right) \sum_{k=1}^{K} \frac{1}{T_k} \log P[O_k|W] \tag{4.6}$$

was used here to compensates this length dependency. This causes a short utterance to have the same influence on the lexicon entry being generated as a long utterance. The combined likelihood score in Equation 4.5 gives longer utterances a higher influence than shorter ones.

To implement the maximisation in Equation 4.4, the following method can be used: First, the spelling of a given word is used to generate a grammar network that represents the likelihood $P[W|S]$ of the different possible transcriptions of that word. Then, a search algorithm is used to find the most likely transcription based on this network and the training utterances O. This search is quite similar to the search in a normal speech recogniser. The main difference is, that now not only a single utterance O but a set of K utterances O_k has to be taken into account. In Section 4.2, an efficient implementation of this search is described. It is based on a modified version of the tree-trellis algorithm presented in Chapter 3.

Until now, a deterministic word lexicon was assumed. Such a lexicon contains a single transcription for each word in the vocabulary. But also a statistical word lexicon can be used in a speech recogniser. An entry for a word in such a lexicon contains a representation of the probabilities if the different possible pronunciations (transcriptions) of that word. The simplest form of such an entry is a list of possible transcriptions of the word. A network with specified transition probabilities can be used as a more advanced representation. It can be seen as an HMM of the whole word with the models for the basic recognition units being the states of this HMM. To estimate the transition probabilities in the whole-word

HMM, given $P[W|S]$ and a set of K training utterances O_k, techniques similar to the common Baum-Welch reestimation or simpler approximations could be used [17]. The use of a statistical lexicon can lead to significantly increased computation and memory requirements.

If the sub-word units used in a speech recogniser are not based on linguistic units like phonemes or syllables, the lexicon can not be generated manually and automatic lexicon generation techniques are indispensable. Different such techniques are investigated in [17], where acoustic sub-word units are employed.

4.2 The Modified Tree-Trellis Algorithm

Commonly, a tree search algorithm is used to find the best transcription $\hat{W}$ according to Equation 4.4. This search might become quite complex, especially if a large number K of training utterances is used. In [15], the use of a modified tree-trellis algorithm was proposed to optimise this search process.

The modified tree-trellis algorithm is very similar to the original algorithm described in Section 3.3.2. A hypothesis is now seen as transcription consisting of several phones and not as a sentence consisting of several words, but this is *only* a difference in terminology and not in the algorithm itself. The modified algorithm differs in two places from the original algorithm. The first difference concerns the different arrays used by the original tree-trellis algorithm. To accommodate the K utterances, they are extended by one dimension with the index k. The rank-ordered predecessor lists $p(m(n), j, k)$ and $h(m(n), t, j, k)$ (having $J(m(n), k)$ entries) are generated independently for each of the K utterances by K runs of the forward Viterbi search. Also in a stack entry n, the rank-ordered lists of possible phone extensions $w(n, i, k)$ and $f(n, i, k)$ (having $I(n, k)$ entries) as well as the arrays $g(n, t, k)$ for the likelihood scores of the backward partial paths are kept separate for the K utterances.

The second difference concerns the array $\hat{f}(n, i)$, which is used to sort the entries on the OPEN stack exactly in the same way as $f(n, i)$ was used in the original algorithm. $\hat{f}(n, i)$ contains the estimated combined likelihood scores for the complete paths for the $I(n)$ possible phone extensions $w(n, i)$. It is calculated at the end of the process of growing a new stack entry n' by expanding the current top entry n in the OPEN stack by the single phone $w' = w(n, i_{\text{next}}(n))$. In this process, first time-reverse Viterbi searches for the phone w' are performed independently for the K utterances and thus $g(n', t, k)$ is calculated for $k = 1, 2, \ldots, K$ and $t = 0, 1, \ldots, T_k$. Then, the forward and backward partial paths are merged independently for the K utterances and thus the arrays $f(n', i, k)$ and $w(n', i, k)$ are generated as in the original algorithm. Now, the phone extensions $w(n', i, k)$ that are possible for *all* K utterances are collected in the new list $w(n', i)$. The corresponding likelihood scores $\hat{f}(n', i)$ are found by combining the scores $f(n', i, k)$ of that phone extension. As mentioned in Section 4.1, two alternative formulas (Equation 4.5 and Equation 4.6) can be used to find this combined score.

As denoted by the "hat" on $\hat{f}(n, i)$, the scores used to order the stack entries on OPEN are now estimates and not exact values as in the original algorithm. This is caused by the fact, that the best forward partial paths used in the merging process might be based on different partial hypotheses for the different utterances. The final hypothesis with the highest combined likelihood score can have a forward partial hypothesis that for an

utterance k can be different from the forward partial hypothesis that lead to the likelihood score stored in the rank-ordered predecessor lists. Since always the highest likelihood score is stored in the predecessor lists, the score, which actually has to be used later, can not be higher. Therefore, the estimate $\hat{f}(n,i)$ is an upper bound for the actual score $f(n,i)$ of the complete hypothesis consisting of the backward partial hypothesis represented by the stack entry n combined with the best forward partial hypothesis. Thus the *admissibility* of the backward A* tree search is given. As long as no method to calculate a better estimate $\hat{f}$ is found, this A* search can also be called *optimal*.

Although it is no problem for this modified tree-trellis algorithm to generate the list of N-best hypotheses, normally only the first best hypothesis generated in the backward search is required in the lexicon generation process.

4.2.1 Implementation of the Modified Tree-Trellis Algorithm

The new HTK tool HViteM is an implementation of the modified tree-trellis algorithm described in Section 4.2. It is based on the tree-trellis N-best recogniser HViteN described in Section 3.4 and can take into account multiple utterances in the search for the best hypothesis. Section A.1.2 contains the user manual for HViteM.

Most of the modifications in the forward Viterbi search of HViteN are necessary to accommodate the K different utterances used now. When HViteM is invoked, a list of speech files has to be specified. These speech files are regarded as separate utterances and loaded into an internal buffer. It is also possible to specify time-aligned label files for the speech files together with the name of a label. In this case, all the speech file segments labeled with the given name are extracted and now these segments are regarded as separate utterances. Thus, all tokens of a word can be extracted as separate utterances from a set of speech files if time-aligned word label files are available and the name of the word is specified.

After all utterances are loaded, a forward Viterbi search is performed individually for each of these K utterances. During such a forward search, also the rank-ordered predecessor lists $p(m(n),j,k)$ and $h(m(n),t,j,k)$ are generated for the current utterance. This search and the predecessor list generation is implemented exactly in the same way as in HViteN.

When all the individual forward trellis searches are finished, a common backward tree search is initiated. This search is an extended version of the backward tree search implemented in HViteN. The process of generating a new stack entry n' by expanding the top entry n in the OPEN stack by its best single phone extension w' now has to take into account all K utterances. First, the backward partial paths are independently extended by the phone w' for each of the K utterances using the same implementation of the time-reverse Viterbi search as in HViteN. Thus, $g(n',t,k)$ is calculated for $k = 1,2,\ldots,K$ and $t = 0,1,\ldots,T[]_k$. Then, the lists of the possible predecessors $w(n',i,k)$ together with their complete path scores $f(n',i,k)$ are generated individually for each utterance k by the same merging technique as in HViteN. But now, all (and not only the top N) entries are maintained.

At the end of the process of generating a new stack entry n', the complete path scores $f(n',i,k)$ for the K utterances are combined to calculate the estimated score $\hat{f}(n',i)$ of the complete hypotheses for the possible predecessors $w(n',i)$. Normally, Equation 4.5 is used to calculated the combined score $\hat{f}(n',i)$. HViteM provides an option to use Equation 4.6

instead of Equation 4.5 to calculate $\hat{f}(n', i)$. The technique of sorting the entries on the OPEN stack and marking the already extended predecessors in a stack entry n using $i_{\text{next}}(n)$ is implemented as in HViteN.

The memory for the different data structures is allocated individually for each utterance, thus minimising memory requirements. Besides the data structures directly needed in the modified tree-trellis algorithm, also the HMM state output probabilities $b_j(o_t)$ are stored individually for the K utterances. The complete data structure of HViteM is not shown here, since it is very similar to the data structure of HViteN shown in Figure 3.10 and Figure 3.11. The pointers lists in RankInfo and xbjot in XInfo (Figure 3.10) now point to arrays containing K pointers to the actual data structures RankList[ ] and Bjot[ ] for the different utterances. The same is done with the pointers backProb and lastTrans in StackEntry (Figure 3.10). In StackEntry, also a pointer to an array of K pointers to RankEntry[ ] data structures is added. They are used to store $f(n, i, k)$ and $w(n, i, k)$. The array RankEntry[ ] pointed to by the pointer totalList in a StackEntry is now used to store $\hat{f}(n, i)$ and $w(n, i)$.

This implementation of the modified tree-trellis algorithm can also be used to find the N hypotheses with the highest combined likelihood score, but normally N is set to 1 and only the best hypothesis is generated.

HViteN offers nearly all the options that are provided by HViteN. Only the demo mode and the generation of an output Master Label File (MLF) are not supported any longer. On the other hand, several new options have been added to control the operation of the modified tree-trellis algorithm. Different from HViteN, where the tree-trellis algorithm was executed for each of the speech files specified, HViteM only executes the modified tree-trellis algorithm once for all the utterances loaded.

4.2.2 Optimisation of the Implemented Algorithm

All the optimisations of HViteN described in Section 3.4.3 are also included in the implementation of the modified tree-trellis algorithm. The number of entries stored in the rank-ordered predecessor lists is not any longer limited to N, but an option to specify the maximum number of entries manually is provided. Since the scores used to sort the stack entries on OPEN are not any longer exact values but estimates, more than N entries might be required on the OPEN stack. Therefore, also the maximum number of entries on OPEN can be specified manually. The entries removed from OPEN because of this size limit are moved to the CLOSED stack. A variable

$$f_{\text{CLOSED}} = \max_{n \text{ on CLOSED}} \hat{f}(n, i_{\text{next}}(n)), \tag{4.7}$$

containing the highest score of a not yet performed expansion for the entries put on CLOSED, is maintained. If the score $\hat{f}(n, i_{\text{next}}(n))$ of the top entry on OPEN becomes smaller than f_{CLOSED}, the size of the OPEN stack is to small. In this case, an entry which is now already put on CLOSED would normally have been expanded next. Because of this fact, it is not any longer guaranteed that the backward A* tree search will find the correct hypothesis and therefore, a warning message is issued.

To minimise the size of the CLOSED stack, a garbage collection for the CLOSED stack is executed periodically every 10th stack entry expansion. During this garbage collection, all entries that are used in backpointer chains starting in any of the entries on the

OPEN stack are marked. All the entries that remained unmarked are completely removed from CLOSED since they will never be used in a backpointer chain when finally the best hypothesis is traced back through the stack entries.

4.3 The Lexicon Generation Process

The whole process of generating a new optimal lexicon for a sub-word based recogniser includes some additional problems besides the search for the most likely transcription of a word as defined by Equation 4.4. In this section, these additional problems as well as different approximations to reduce to complexity of the search for the most likely transcription will be addressed.

4.3.1 Overview of the Lexicon Generation Process

To give an overview of the automatic lexicon generation process, its dataflow is shown in Figure 4.1. This diagram illustrates the interdependencies between the different data bases for the processes of HMM training and lexicon generation. It also shows the need for a word level segmentation (a time-aligned word label file) of the training utterances. This segmentation is needed to be able to extract the different tokens of a word from the training utterances, since it is common to use e.g. whole sentences as training utterances in a continuous speech recogniser.

For the training utterances used here, only orthographic transcription were available. The time-aligned word label files were obtained automatically using HAlignW, a special version of HTK's recogniser HVite. The network used by HAlignW is generated automatically by concatenating the lexicon entries of the words in utterance and inserting optional silence models between the words. HAlignW is a slightly modified version of HTK's original HAlign tool. The modifications were necessary since HAlign only generates a phone level label file and not a word level label file. The optional silence models were not concerned as a part of a word here, although they are normally included in the subnet for a word (see Figure 2.7 and Figure 2.8). Section A.1.3 contains the user manual for HAlignW.

Since the optimal HMM parameters depend on the lexicon (it is needed during the embedded HMM reestimation) as well as the optimal lexicon depends on the HMM parameters, a joint optimisation of the HMMs and the lexicon could be advantageous. It could be implemented by alternately reestimating the HMM parameters and generating a new lexicon.

In this thesis, the spelling S was not used as a an information source in the search for the most likely transcription $\hat{W}$. Therefore, a simple network allowing arbitrary combinations of the 47 different phonemes was used:

```
$phn = (ax|ey| ... |dh);
(<$phn>)
```

This "one pass" like network results in nearly equal a priori probabilities $P[W]$ for the different transcriptions W if the default transition probability in Equation 2.40 is used.

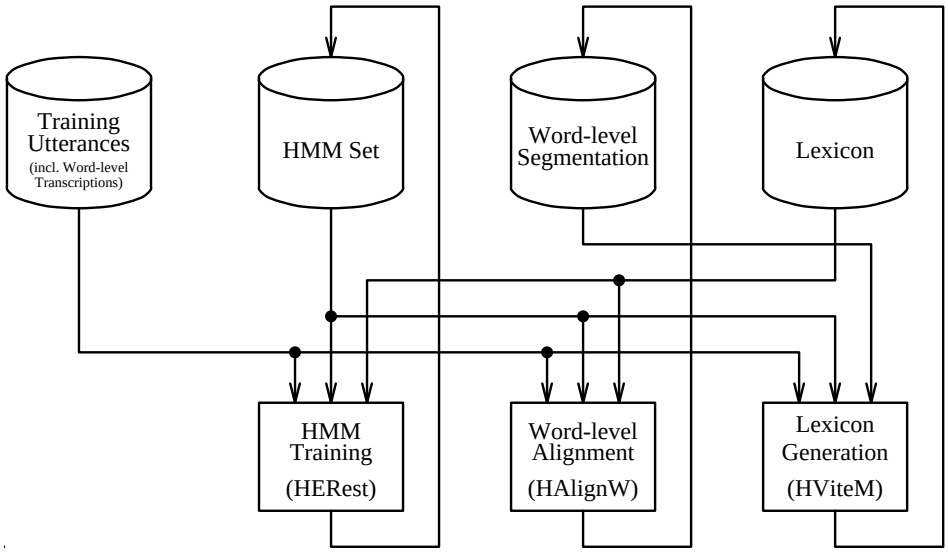


Figure 4.1: Dataflow in the automatic lexicon generation process.

To be precise, this probability is

$$P[W] = \left(\frac{M}{M+1}\right)^{L(W)-1} \frac{1}{M+1}, \tag{4.8}$$

where M is the number of different phonemes (here $M = 47$) and $L(W)$ is the number of phonemes is transcription W.

An important problem in the automatic lexicon generation process is the selection of training tokens. The number of training tokens as well as their selection both have a significant influence on the search for the most likely transcription. It is obvious that an increased number of training tokens results in a more reliable new transcription — simply because of the increased amount of information being taken into account. But on the other hand, this increases also the complexity of the search for the most likely transcription. In first experiments, where simply all available training tokens were used, the this search was in several cases (i.e. for several words) too complex for the available hardware. Therefore, different techniques for the preselection of training tokens were used and also approximations in the search process were investigated.

4.3.2 Extensions to the Modified Tree-Trellis Algorithm

The implementation HViteM of the modified tree-trellis algorithm described in Section 4.2 was extended in different ways to allow techniques for the preselection of training tokens to be used.

The first extension is a new option which allows to specify the maximum number of tokens being loaded. If no word label is specified, a whole speech file is regarded as a token (see Section 4.2.1). The tokens are loaded from the speech files in the order these files appear in the speech file list. When the specified number of tokens is loaded or all listed speech files are processed, the loading process is terminated and then the modified tree-trellis search is performed as usual.

Another option allows to specify a file containing a list of token indices. The tokens are simply indexed by the order in which they normally would have been loaded. This option allows to use an arbitrary subset of the available training tokens for a word. It is needed when a preselection of training tokens is done. This option can be used together with the option specifying the maximum number of tokens to load.

As a base for the preselection of training tokens or for the estimation of the most likely transcription, a list containing the N-best transcriptions and their average log likelihood scores per time frame for each of the training tokens of a word is used in this thesis. To simplify the generation of this list, HViteM was extended in such a way, that the backward tree search is not performed commonly for all tokens but that a separate backward tree search is performed for each of the loaded training tokens. Thus, the normal (unmodified) tree-trellis search is performed individually for each token. This special operation mode of HViteM is controlled by an option. The main remaining difference to HViteN is the method in which the tokens are loaded.

It is also possible to disable the backward tree search completely. If this option is used, only the separate forward Viterbi searches are performed for the loaded tokens. It is mainly intended to provide an easy means of calculating the likelihood scores of the loaded tokens of a word and is normally used with a network that represents a specific transcription of that word.

4.3.3 Preselection of Training Tokens

The need for a preselection of training tokens arises mainly from problems with the complexity of the search for the most likely transcription. The simplest way to preselect training tokens is to randomly choose a fixed number of tokens. This is implemented here by randomly scrambling the speech file list (the list of all training utterances) and then limiting the number of tokens loaded by HViteM. But even if a set of only 10 tokens was used, the resulting search was in several cases (about 10% of the words, see Section 4.4.1.2) still too complex for the available hardware. This can e.g. happen if the tokens represent quite different pronunciations of the same word.

To avoid this problem, different token clustering algorithms were employed to find the biggest cluster containing the tokens that represent the most common pronunciation of a word. These clustering algorithms are based on the individually most likely transcriptions of the different tokens of a word. Different possible clustering methods will be described in this section.

In the lexicon generation process discussed here, it is assumed that an initial (manually generated) lexicon is available. This lexicon is e.g. needed to train the HMMs by embedded reestimation. It can also be used to select words which might be excluded from the lexicon generation. Words with long transcriptions (e.g. more than 10 phonemes) in this original lexicon might be excluded, since they are more unlikely to be misrecognised than shorter words. Excluding them can also be advantageous, since the complexity of the backward tree search normally increase exponentially with the length of a word and its transcription.

It is also reasonable to specify a minimum number of training tokens for words to be included in the lexicon generation process. If the automatically generated lexicon entry for a word is based only on a few training tokens of that word, the new transcription might not be very reliable and can e.g. represent an untypical pronunciation. The turnover point

at which the automatically generated lexicon starts to perform better than original lexicon seems to be at slightly under 10 tokens (assuming that they are randomly chosen) [15, p. 16]. The results obtained in this thesis confirm this observation. Thus, it is reasonable to include only the tokens with e.g. at least 10 available training tokens in the lexicon generation process.

Since the pronunciation of a word might depend on its adjacent word in the sentence, it could also be advantageous to ensure that a set of training tokens is used which represent the typical variety of adjacent words. But this effect was not taken into account in this thesis.

In the following, different methods for the preselection of training tokens based on clustering techniques will be discussed.

4.3.3.1 The String Distance Measure

The clustering techniques used here are based on the N-best transcriptions for each individual token of a word. These transcriptions are regarded as a string of phonemes here. To be able to cluster these transcriptions, a string distance measure is required. Here, the Levenshtein distance is used which sets the costs of deletions, substitutions and insertions uniformly to 1. This distance can be calculated by a string matching process based on dynamic programming [15, 20].

The distance $d(A, B)$ between the two strings $A = \{a_1, a_2, \ldots, a_I\}$ and $B = \{b_1, b_2, \ldots, b_J\}$ is the minimum cost of a transformation that converts A into B. Using the sub-strings $A_i = \{a_1, a_2, \ldots, a_i\}$ and $B_j = \{b_1, b_2, \ldots, b_j\}$, it can be computed iteratively. Starting from the empty sub-strings A_0 and B_0 with

$$d(A_0, B_0) = 0, \tag{4.9}$$

the following three basic transformation steps are possible to convert A into B:

Deletion: The cost of deleting a is $D(a)$

$$d_D(A_i, B_j) = d(A_{i-1}, B_j) + D(a_i), \quad 1 \leq i \leq I, \quad 0 \leq j \leq J. \tag{4.10}$$

Substitution: The cost of substituting a by b is $S(a, b)$

$$d_S(A_i, B_j) = d(A_{i-1}, B_{j-1}) + S(a_i, b_j), \quad 1 \leq i \leq I, \quad 1 \leq j \leq J. \tag{4.11}$$

Deletion: The cost of inserting b is $I(b)$

$$d_I(A_i, B_j) = d(A_i, B_{j-1}) + I(b_j), \quad 0 \leq i \leq I, \quad 1 \leq j \leq J. \tag{4.12}$$

The basic iteration step is now

$$d(A_i, B_j) = \min\{d_D(A_i, B_j), d_S(A_i, B_j), d_I(A_i, B_j)\}, \quad 1 \leq i \leq I, \quad 1 \leq j \leq J \tag{4.13}$$

with the two special cases

$$d(A_i, B_0) = d_D(A_i, B_0), \quad 1 \leq i \leq I \tag{4.14}$$

and

$$d(A_0, B_j) = d_I(A_0, B_j), \quad 1 \leq j \leq J. \tag{4.15}$$

At the end of this iteration, the string distance

$$d(A, B) = d(A_I, B_J) \tag{4.16}$$

is found.

As mentioned, the different costs are set uniformly to 1 here:

$$D(a) = 1 \tag{4.17}$$

$$S(a, b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases} \tag{4.18}$$

$$I(b) = 1 \tag{4.19}$$

This also causes the distance measure to be symmetric:

$$d(A, B) = d(B, A) \tag{4.20}$$

4.3.3.2 The Nearest Neighbour Clustering Algorithm

In [20], a string clustering algorithm based on the nearest neighbour decision rule was presented. A slightly modified version of this algorithm was also employed in [15]. It is similar to the modified K-means clustering algorithm described in [1, pp. 271–274].

The nearest neighbour clustering algorithm divides a set $T = \{t_1, t_2, \ldots, t_S\}$ of S strings t_s into M clusters $U_m = \{u_{m1}, u_{m2}, \ldots, u_{mL_m}\}$ each containing the L_m strings u_{ml}. The centroid of a cluster U_m is defined as that string c_m in cluster U_m that minimises the intra-cluster distortion

$$\sum_{j=1}^{L_m} d(c_m, u_{mj}) \leq \sum_{j=1}^{L_m} d(u_{ml}, u_{mj}), \quad 1 \leq l \leq L_m. \tag{4.21}$$

In the nearest neighbour clustering algorithm, the string clustering is performed iteratively. Before the first iteration $i = 1$, the number of clusters is initially set to $M = 0$. In the core procedure of this iteration, each of the S strings t_s in T are processed sequentially. For each string t_s, first the cluster $U_{m_{\text{opt}}}$ is found by a nearest neighbour search

$$m_{\text{opt}} = \arg\min_m d(t_s, c_m). \tag{4.22}$$

If this minimum distance $d(t_s, c_{m_{\text{opt}}})$ is not greater than a intra-cluster string distance threshold $d_{\max}$, the string t_s is assigned to cluster $U_{m_{\text{opt}}}$. Otherwise, a new cluster with the centroid t_s is created and t_s is assigned to this new cluster. If all K strings have been processed, the empty clusters are removed, the new cluster centroids c_m are computed and the total intra-cluster distortion

$$D_i = \sum_{m=1}^{M} \sum_{j=1}^{L_m} d(c_m, u_{mj}) \tag{4.23}$$

for this i'th iteration is calculated. The clustering process is terminated now if convergence can be assumed. This is the case if all new centroids are equal to the centroids found in the previous iteration $i-1$, if the maximum number of iterations $i_{\max}$ is reached or if the relative reduction of the total distortion D_i is not above a threshold a, i.e.

$$\frac{D_{i-1} - D_i}{D_i} \leq a. \tag{4.24}$$

Otherwise, all clusters are cleared and only their centroids are maintained before the core procedure is started again for the next iteration $i+1$.

After each iteration, the clusters are sorted according to their size L_m. When this nearest neighbour clustering algorithm has terminated, the members of the first cluster (i.e. the biggest cluster) are returned. The final clustering obtained by this algorithm depends very much on the maximum intra-cluster string distance $d_{\max}$, on the termination threshold a and on $i_{\max}$.

When this clustering algorithm is used for the preselection of training tokens, the list of the N-best individual transcriptions of the K training tokens is the set T of $S = NK$ strings to be clustered. After the clustering algorithm has terminated, each of the K tokens is assigned to the biggest cluster that contains a transcription belonging to that token. The tokens assigned to the cluster that has the highest number of tokens assigned are returned. Thus, a set of preselected training tokens is obtained.

In [15], a modified version of the nearest neighbour clustering algorithm was used. In this version, two transcriptions of the same token are not allowed to occupy the same cluster. Thus, the N different transcriptions of a training token had to belong to N different clusters. It is also possible to use the M_{entry} biggest clusters to generate multiple entries for a word.

4.3.3.3 Other Clustering Algorithms

Also some alternative clustering algorithms were implemented and briefly investigated in this thesis. The first alternative algorithm is very similar to the nearest neighbour clustering algorithm described above. It clusters the tokens and not the transcription strings themselves. The distance between a token and the centroid string of a cluster is computed as the sum of the distances between the centroid string and the N transcription strings of that token.

The other alternative clustering algorithm is based on the "unsupervised clustering without averaging" described in [1, pp. 268–270]. It iteratively splits the string set R_i into a new cluster of strings within a distance threshold of the centroid of R_i and a new set of remaining strings R_{i+1}. This iteration is initialised by $R_0 = T$.

Additionally, different minor modifications of these algorithms were tried. In Section A.1.5, the usage of the program wordclust that implements all these clustering methods is described. The outcome of the different clustering methods presented here could only be studied briefly in this thesis due to the complexity of a detailed investigation. Also the clustering parameters (e.g. the maximum intra-cluster string distance $d_{\max}$) were not optimised by objective criteria but only by manual examination of the algorithms' outcome.

4.3.4 Estimation of the Most Likely Transcription

Based on a set of lists that contain the likelihood scores of all different possible transcriptions for each of the K tokens, the transcription $\hat{W}$ with the highest combined likelihood score could be found by computing the combined likelihood scores of all possible transcriptions W based on this set of lists (Equation 4.5 or Equation 4.6) and finally performing the maximisation in Equation 4.4. This technique is of course impossible due to the prohibitive size of these lists.

Nevertheless, an approximative technique based on this concept is feasible. It requires the lists of the individual N-best transcriptions of each of the K tokens together with the likelihood scores of these transcriptions. For each of the different transcriptions W that occur in these lists, the combined likelihood score is computed by combining the scores this transcription has for each of the K tokens. If a transcriptions W is not within the N-best list for a token k, the score of that transcription for that token has to be estimated. The estimation technique used here is based on that transcription $W_{\min}$ of the token k in the N-best list that has the minimum string distance $d_{\min} = d(W, W_{\min})$ to transcription W. The log likelihood score $\log P[O_k|W]$ is now estimated by

$$\frac{1}{T_k} \log \hat{P}[O_k|W] = \min \left\{ \frac{1}{T_k} \log P[O_k|W_N], \frac{1}{T_k} \log P[O_k|W_{\min}] - p - x(d_{\min})^y \right\}, \quad (4.25)$$

where W_N, the N'th (i.e. worst scoring) transcription in the N-best list for token k, obviously is an upper limit for the estimated score. The parameters p (fixed penalty), x (distance score penalty factor) and y (distance score penalty exponent) allow to optimise this estimation method. T_k is the length of the k'th token. It is not explicitly required here, since the average log likelihood scores per time frame is directly given in the N-best lists. Because of this fact, Equation 4.6 is used here to compute the combined likelihood score.

When all the combined likelihood scores for the different transcriptions W in the N-bests lists have be computed, the highest scoring transcription $\hat{W}$ is returned according to Equation 4.4. In this thesis, the parameters p, x and y were optimised only by manual examinations of the algorithm's outcome. This algorithm was implemented as a special option in the string clustering program wordclust. In Section A.1.5, the usage of this program is described.

4.4 Experimental Results

To investigate the different automatic lexicon generation techniques described above, these techniques were used to generate several new lexica. Then, the performance of these lexica and their effect on the results of recognition tests were examined.

The lexicon generation experiments conducted in this thesis were based on the 1000 word DARPA resource management (RM) corpus described in Section 2.3.3. In all experiments, the HMM set "base" consisting of 47 context independent phoneme models (the 2 silence models are not required here) were used together with the simplest possible grammar network. This network was described in Section 4.3.1 and allows all possible phoneme transcriptions W with a nearly equal a priori probabilities $P[W]$. Thus, the spelling of a word was not taken into account in these experiments.

The complete speaker independent training set consisting of 3990 utterances with a total of 34722 words was used for the automatic lexicon generation. The performance of the new lexica was investigated using the test set "feb89". This test set consists of 300 utterances with a total of 2561 words.

Different subsets of the complete set of 991 words in the RM corpus were used here. 988 of the 991 words had at least one token in the training set. The set of words having at least 10 tokens in the training set and an original transcription of not more that 10 phonemes contains 592 words. It is called "set10" here. In the test set "feb89", 577 of the 991 words were represented with at least one token. Of the 592 words in "set10", 409 were represented with at least one token in "feb89".

4.4.1 Generation of the New Lexica

In the following, the details of the generation of the different new lexica are described. The search for a new transcription was aborted unsuccessfully for several words in these lexica due to the limited memory resources of the available hardware. The fact, that therefore new transcriptions are missing for several words in the new lexica, has to be taken into account when the performance of these lexica is compared.

The actual generation of a new lexicon was done using several UNIX shell scripts. These scripts were used to call the different programs like HAlignW, HViteM, wordclust, ... with the necessary parameters. Additionally, several small utility programs were required to handle and convert the different data file formats used in the lexicon generation process. These utility programs and script files are described in Section A.2 and Section A.3.1. As an example, the process of generating and testing the new lexicon "clust100" is described in detail in Section A.3.2.

4.4.1.1 The Lexicon "max100"

The lexicon "max100" was generated quite straightforward. HViteM was used to find the most likely transcription of all 988 words having at least one token in the training set. If there were not more than 100 tokens of a word in the training set, all these tokens were taken into account. Otherwise, a subset containing exactly 100 randomly chosen tokens of that word were used. This was necessary for 67 words. Equation 4.5 was used to compute the combined likelihood scores.

The search for a new transcription was aborted if further required memory could not be allocated (i.e. the available memory resources were exhausted) or if a stack size warning was issued. Such a warning is issued by HViteM if it is detected that the limitation of the OPEN stack size could lead to a search error.

Different OPEN stack sizes on the order of 200 to 500 entries were used here. Different stack sizes were tried since the required stack size for word a with a given set of training tokens depends on the number of tokens, the duration of these tokens and on the variation of pronunciation encountered in the tokens. If the stack size was too small, a stack size warning could be issued. Otherwise, a memory allocation problem could be encountered. During the lexicon generation process, it was possible to allocate maximum of about 20 Mbyte memory for HViteM.

Finally, new lexicon entries were found for 801 of the 988 words tried. The performance of this lexicon is discussed in Section 4.4.2.

4.4.1.2 The Lexica "rand10" and "rand10_r"

The lexica "rand10" and "rand10_r" were also generated quite straightforward. The first main difference to the process of generating "max100" is, that here only the 592 words in "set10" were included in the lexicon generation process. Thus, it was not tried to find a new entry for a word with less that 10 training tokens available or with an original transcription longer than 10 phonemes. The other main difference is, that for each word only 10 randomly chosen tokens were used. Like for "max100", also here different OPEN stack sizes were tried.

The two lexica "rand10" and "rand10_r" differ only in the equation that was used to compute the combined likelihood score. Equation 4.5 was used for lexicon "rand10" while the token length compensated score in Equation 4.6 was used for lexicon "rand10_r". In lexicon "rand10", new lexicon entries were found for 526 of the 592 words tried. In "rand10_r", 541 of 592 word entries were found. The performance of these lexica is discussed in Section 4.4.2.

4.4.1.3 The Lexicon "clust100"

The generation of the lexicon "clust100" was based on the nearest neighbour clustering algorithm described in Section 4.3.3.2. First, lists with the individual 10-best transcriptions of all tokens of the 988 words that occur in the training set were generated using a special option in HViteM. For each of these words, the listed transcriptions were clustered and the tokens assigned to the biggest cluster were then used as the set of preselected training tokens. To obtain reasonable results from the clustering algorithm for words with short transcriptions as well as for words with long transcriptions, the maximum intra-cluster string distance $d_{\max}$ was chosen depending on the average length $\bar{L}$ of the listed transcriptions of a word. Here, the distance threshold

$$d_{\max} = d_0 + l\bar{L} \tag{4.26}$$

was used. The actual parameter values used here for the clustering were $d_0 = 1$, $l = 0.3$ and a threshold $a = 0$ for the relative reduction of the total distortion D_i. Different from [15], the number of transcriptions of the same token within one cluster was not limited. In brief experiments with these both versions of the clustering algorithm, no significant advantages of the limitation used in [15] were observed.

Based on these preselected sets of training tokens, the most likely transcriptions for the 988 words were found using HViteM. If a cluster contained more than 100 token, exactly 100 of these tokens were selected randomly and used by HViteM. Otherwise, all tokens in the cluster were used. Due to the preselection process, the tokens taken into account in the search for the new transcription represented similar pronunciations of a word and therefore, no problems with the complexity of this search were encountered. To compute the combined likelihood score, Equation 4.5 was used.

Finally, new lexicon entries for all 988 words were found. Of the new transcriptions in this lexicon, 413 (41.8%) were equal to the centroid of the biggest cluster — the cluster

determining the set of training tokens used. The performance of this lexicon is discussed in Section 4.4.2.

4.4.1.4 The Lexicon "estimate"

The generation of the lexicon "estimate" was based on the same lists of the individual 10-best transcriptions of all tokens that were also used for the generation of "clust100". To estimate the most likely transcriptions for the 988 words having at least one training token, the method described in Section 4.3.4 was used. The parameters of the score estimation (Equation 4.25) were set to $p = 0$, $x = 1$ and $y = 2$. Since only the average log likelihood scores per time frame are available in the transcription lists, Equation 4.6 was used to compute the combined likelihood scores.

The lexicon estimation algorithm requires about the same amount of memory as the string clustering algorithm, which is a fraction of the memory that could be required by HViteM in the exact search for the most likely transcription. Thus, it was no problem to generated new entries for all the 988 words. The performance of this lexicon is discussed in Section 4.4.2.

4.4.2 Comparison of the New Lexica

To evaluate the performance of the different new lexica, the test set "feb89" consisting of 300 utterances with a total of 2561 words was used. Of the 988 words having tokens in the training set, 576 words have at least on token in this test set. And of the 592 words in "set10", 409 have a token in the test set "feb89". Table 4.1 contains an overview of the different new lexica. It lists number of lexicon entries for which a transcription was found in the automatic lexicon generation processes. Also the number of new lexicon entries that were equal to the corresponding entry in the original lexicon is given.

To measure the performance of the new lexica, the average likelihood scores for the transcriptions in the different lexica were computed. First, the transcription in the lexicon entry for a word was converted into a network. Then, this network was used by HViteM to find the average log likelihood score per time frame for each token of that word in the test set. Only the forward Viterbi searches for the tokens were required to find these average scores. Therefore, the backward tree search in HViteM was disabled. The required time-aligned word label files for the utterance in the test set were generated using HAlignW in exactly the same was as for the training set. This means, that the time-alignment for the words in the training set as well as in the test set was computed using the transcriptions in the original lexicon.

After the average log likelihood scores per time frame were computed for all tokens of the different words in a lexicon, the mean value of these scores for all tokens was calculated. Also the average likelihood scores for the words were computed and the mean value of these score for all words was calculated. Since the new lexica contain a different numbers of entries, only those words that have an entry in all new lexica were used for the comparison. There are totally 451 of these word and 301 of them have at least on token in the test set "feb89". This set of words is called "common301". To avoid that words with only a few (maybe untypical) tokens in the test set have a big influence on the average word score, another set of words was defined. This set, called "common47", contains all those words

in "common301" that have at least 10 tokens in the test set "feb89".

The average word log likelihood scores per time frame for the words in "common47" are listed in Table 4.1. For all lexica except "rand10_r", Equation 4.5 was used to calculate the average score for a word. For the lexicon "rand10_r", Equation 4.6 was used. The name "original_r" refers to the same original lexicon as "original". The appended "_r" indicates only, that Equation 4.6 instead of Equation 4.5 was used to compute the average word scores. Besides the average word score, also the average score gain relative to the original lexicon is given for each lexicon.

The average token log likelihood scores per time frame for the tokens of the words in "common301" are also listed in Table 4.1. This mean score was calculated for all tokens. Thus, the more frequent tokens have a higher influence on the mean score. Because of this fact, these token mean scores can be seen as a more realistic measure of the lexicon performance than the word mean score. Besides the average token score and the average score gain relative to the original lexicon, also the standard deviation of the difference between the token scores for the original and for the new transcription is given.

Lexicon	**Lexicon Entries**			**Performance** (test set: "feb89")				
			equal	47 words ("common47")		1881 tokens ("common301")		
	tried	found	"orig."	mean score	score gain	mean score	score gain	std.dev.
"original"		(991)		-78.3485	0.0000	-78.4206	0.0000	0.0000
"original_r"		(991)		(-78.6970)	(0.0000)			
"max100"	988	801	144	-78.0569	0.2916	-78.1962	0.2244	0.8169
"rand10"	592	526	92	-78.1595	0.1890	-78.3372	0.0834	1.1910
"rand10_r"	592	541	95	(-78.5444)	(0.1526)	-78.3168	0.1038	1.1790
"clust100"	988	988	120	-78.1293	0.2192	-78.2345	0.1861	1.0840
"estimate"	988	988	96	-78.2459	0.1026	-78.2925	0.1281	1.2010

Table 4.1: Size and performance of the different lexica.

The likelihood score gains listed in Table 4.1 show, that the lexicon "max100" offers the highest performance gain compared to the original lexicon. The lexicon "clust100" offers a slightly lower performance gain due to the approximations inherent in concept of token preselection. The increased standard deviation of the score difference, compared with "max100", indicates also a decreased reliability of the new transcriptions. The lexica "rand10" and "rand10_r" offer a significant lower performance gain than "max100" or "clust100". This is caused by the low number of training tokens taken into account in the search for the new transcription. Also the higher standard deviation of the score difference, compared with "max100" and "clust100", indicates, that a low number of training tokens results in a less reliable new transcription. No clear performance difference between the two methods for calculating the combined likelihood score (Equation 4.5 and Equation 4.6) can be observed. The performance gain of the lexicon "estimate" is comparable to the gains obtained by "rand10" and "rand10_r". Also the standard deviation of the score difference of these three lexica is comparable.

In Figures 4.2 to 4.6, the distribution of the difference between the word scores and token scores of the new and the original transcription is shown for all the five different lexica generated in this thesis. As in Table 4.1, also here the average log likelihood score per time frame is used. In the distributions of the word score differences, all words having an entry in the new lexicon and at least one token in the test set "feb89" were included. The distributions of the token score differences only include the tokens of the words in "common301". Therefore, these token score difference distributions can easily be compared

for the different lexica.

Some characteristic parameters of these distributions are given in the diagrams. The mean value and the standard deviation of the word or token score difference is shown. Also the percentage of words or tokens with a lower, equal or higher score is given.

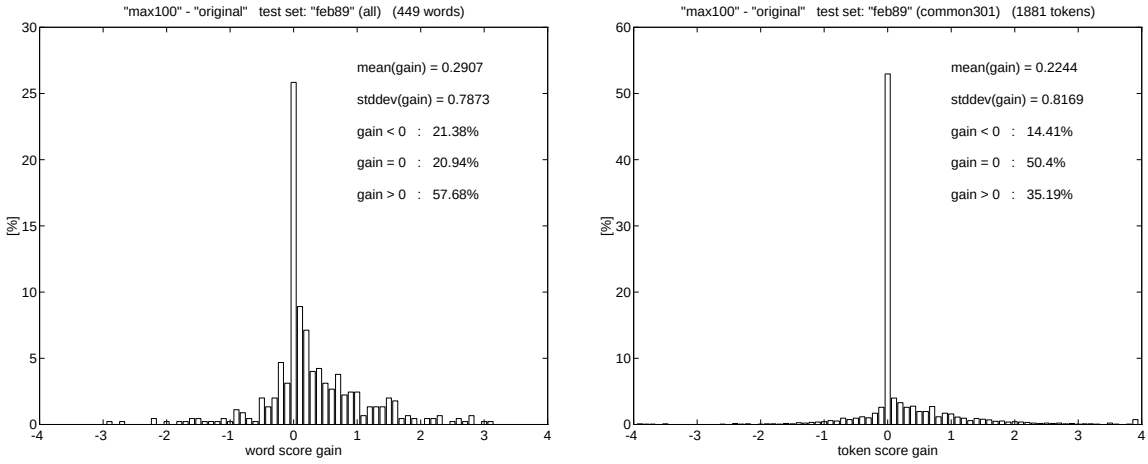


Figure 4.2: Distribution of the word score gain and token score gain for lexicon "max100".

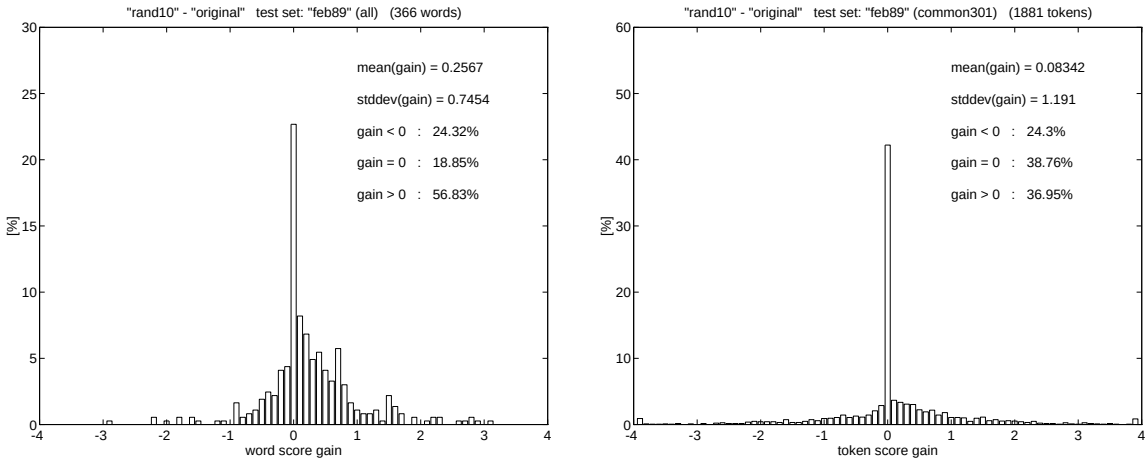


Figure 4.3: Distribution of the word score gain and token score gain for lexicon "rand10".

Finally, full recongition test were performed to investigate the new lexica under real conditions. For each of the five new lexica, two recognition tests were performed. In the first test, all transcriptions in a new lexicon were used and only for those words that had no transcription in the new lexicon, the corresponding entries from the original lexicon were copied. Due to the different number of entries in the new lexica, the results of these recognition test can not be easily compared for the different lexica. Therefore, a second test was performed for all lexica. In this test, only the new transcriptions for the words in "common301" were used and for all other words, the corresponding original entries were copied. The results of these tests now can be directly compared for the different lexica.

The results of these recognition tests are summarised in Table 4.2. The number of automatically generated entries in the different lexica is given as well as the recognition rate for whole utterances. Also the recognition accuracy on word level is given. The number

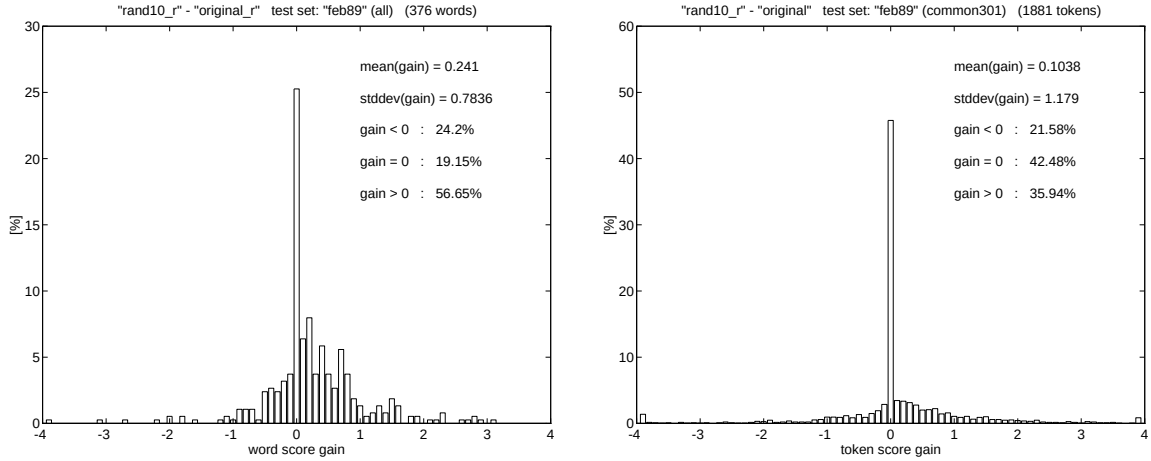


Figure 4.4: Distribution of the word score gain and token score gain for lexicon "rand10_r".

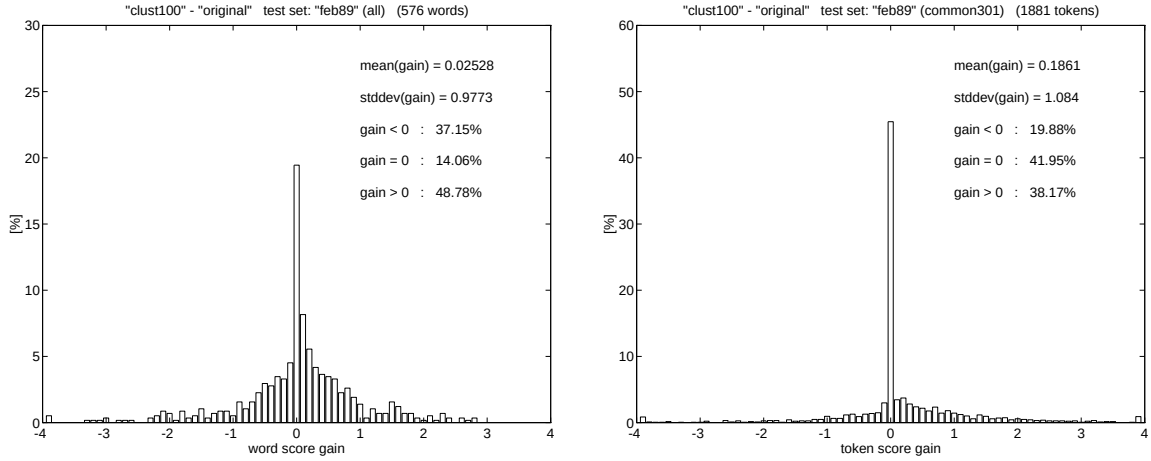


Figure 4.5: Distribution of the word score gain and token score gain for lexicon "clust".

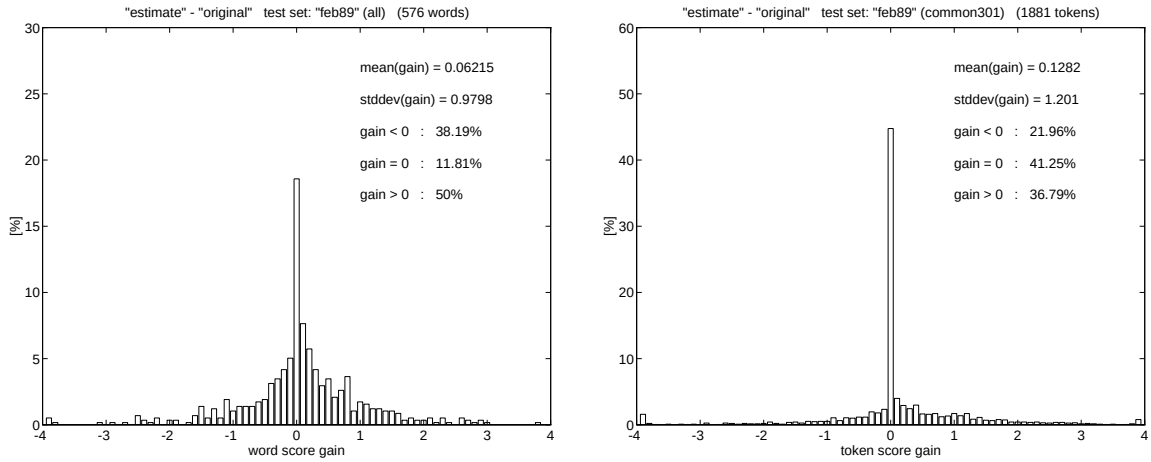


Figure 4.6: Distribution of the word score gain and token score gain for lexicon "estimate".

of word deletions, substitutions and insertions were found by the HTK tool HResults and are based on dynamic programming matches between the recognised and the correct sentences. In these tests, the same HMM set "base" as for the lexicon generation processes was used. The grammar networks for the recognition tests were based on the word-pair grammar and were generated by the RMTK tool HSnor2Net using the new lexica.

Lexicon	**New Entries**		**Utterances**			**Words**					
		in	Correct	$N = 300$		Correct	Accuracy	$N = 2561$			
	total	"feb89"	H/N	H	S	H/N	$(H-I)/N$	H	D	S	I
"original"	(991)	(577)	29.00%	87	213	78.33%	76.10%	2006	124	431	57
all new lexicon entries used:											
"max100"	801	448	33.67%	101	199	81.30%	79.70%	2082	119	360	41
"rand10"	526	366	30.67%	92	208	79.85%	77.70%	2045	112	404	55
"rand10_r"	541	376	29.67%	89	211	79.77%	77.39%	2043	118	400	61
"clust100"	988	576	28.67%	86	214	78.60%	75.75%	2013	111	437	73
"estimate"	988	576	30.33%	91	209	79.85%	76.92%	2045	119	397	75
only common new lexicon entries used ("common301"):											
"max100"	451	301	32.00%	96	204	80.59%	78.33%	2064	111	386	58
"rand10"	451	301	29.67%	89	211	79.19%	76.53%	2028	113	420	68
"rand10_r"	451	301	29.00%	87	213	78.95%	76.45%	2022	115	424	64
"clust100"	451	301	32.67%	98	202	79.77%	77.43%	2043	105	413	60
"estimate"	451	301	32.00%	96	204	78.95%	75.91%	2022	114	425	78

(H = correct items, D = deletions, S = substitutions, I = insertions, N = items in reference)

Table 4.2: Recognition test results for the different lexica (test set: "feb89", HMM set: "base", grammar: word-pair).

The word recognition accuracies listed in Table 4.2 confirm the observations concerning the performance of the different lexica that were made for the average token and word scores in Table 4.1. Since only a total of 300 utterances were tested, the differences in the number of correct recognised utterances are are quite low. These differences seem to be only slightly above the statistical uncertainty inherent in such recognition tests. Nevertheless, also the utterance recongition rates indicate the same performance differences between the lexica that were observed otherwise.

If the two different tests for a lexicon are compared, the influence of the number of training tokens on the generated transcription can be observed for the different lexicon generation techniques. This is due to the fact, that in the second test ("common301"), for each word at least 10 training tokens were available for the lexicon generation. For the lexicon "max100", the word recognition accuracy increased if all new lexicon entries were used. The interesting observation in this comparison is, that word recognition accuracy for "estimate" increased while the accuracy for "clust100" significantly decreased if all new lexicon entries were used in the tests. This is caused by the fact, that even if only a few training tokens are available, an even smaller subset of these tokens will be selected by the clustering algorithm and then taken into account in the search for the new transcription. Thus, the reliability of the new transcriptions is decreased significantly in these cases.

In Table 4.3, the entries in the different lexica are given for five words. These words were chosen to illustrate typical properties and problems of the different automatic lexicon generation techniques studied here. For word CHART, all lexica except "estimate" contain the original transcription. For word WHICH, all new lexica contain the same transcription. It differs in the vowel "ih" and the following stop closure "td" from the original transcription and represents a more sloppy pronunciation. For the word TWENTY, several different

transcriptions were found. They show problems with the recognition of the initial stop "t" and also indicate a sloppy pronunciation. The phoneme "r" in the entries in "rand10" and "rand10_r" illustrates the kind of errors that can be caused by only using a few training tokens. The word INDIAN mainly illustrates the effect of a sloppy pronunciation on the generated lexicon entry. The word NOVEMBER shows problems with the recongition of the initial nasal "n". The stop closures "td" or "kd" at the end of the new transcriptions might be caused by problems with the word time-alignment of the training utterance and possibly only represent silence.

During the generation of "max100", the search for a new transcription of the words INDIAN and NOVEMBER was aborted due to complexity problems. These problems might have been caused either by the lack of a common pronunciation in the training tokens or by long duration of the tokens. The difference between the transcriptions in "clust100" and the corresponding centroids illustrate that the string distance measure used in the clustering does not take into account possible similarities between phonemes.

Lexicon	**Transcriptions**				
word ⇒	CHART	WHICH	TWENTY	INDIAN	NOVEMBER
tokens ⇒	124	307	152	38	35
"original"	td ch aa r td	w ih td ch	t w ah n iy	ih n d iy ax n	n ow v eh m b er
"max100"	td ch aa r td	w ax dd ch	k w dx iy	(not found)	(not found)
"rand10"	td ch aa r td	w ax dd ch	t w r iy	ih n y n	m ow v eh m b er td
"rand10_r"	td ch aa r td	w ax dd ch	t w r iy	ih n y n	m ow v eh m b er td
"clust100"	td ch aa r td	w ax dd ch	p w dx iy	ih n y n	m ow v ah m b er kd
centroid ⇒	dd ch aa r	w ax ts ch	p w dx iy	ih n y m	m ow v ah m b er kd
cluster size ⇒	141	67	76	14	12
"estimate"	dd ch aa r	w ax dd ch	t w dx iy	ih n y ax n	m ow v eh m b er kd

Table 4.3: Examples of transcriptions from the different lexica.

The performance of the different lexica can be summarised as follows:

- **"max100"** This lexicon gives the highest increase of performance compared to the original lexicon. It still has the disadvantage that for several words, the search for a new entry was aborted due to complexity problems.
- **"rand10"** This lexicon gives an increase in performance but also shows the problems that arise of only a low number of training tokens is taken into account in the lexicon generation process.
- **"rand10_r"** This lexicon is quite similar to "rand10" and mainly indicates that there is no clear difference in performance between the two different methods to combine the token likelihood scores (Equation 4.5 and Equation 4.6).
- **"clust100"** This lexicon gives nearly the same performance as "max100" and contains entries for all words since no search complexity problems were encountered. But this lexicon shows also, that no preselection of training tokens should be used if only a few tokens are available.
- **"estimate"** This lexicon gave an increase in performance of about the same order as "rand10" and "rand10_r". It contains entries for all words, but these entries should only be used if none of the other lexicon generation techniques can be used.

4.5 Discussion

In this chapter, different lexicon generation techniques were described. New lexica for a continuous speech recogniser for the DARPA resource management corpus were generated and their performance were compared in recognition tests. During this work, several problems of the automatic lexicon generation were observed. Different suggestions to solve some of these problems and thus further optimise the automatic lexicon generation will be described now.

To make maximum use out of the available training tokens and the available hardware, the different lexicon generation techniques described here should be combined. In a first step, a lexicon like "max100" should be generated taking into account as much training tokens as possible. For those words where no new entry was found due to search complexity, a clustering based token preselection like in "clust100" should be used. A threshold for the minimum number of tokens in the preselected set is suggested. This threshold could have a value on the order of 10 tokens. If the preselected token set is smaller than this threshold, an estimation technique like in "estimate", based on *all* available tokens, should be used instead of a full search for the most likely transcription for the small preselected token set.

To improve the string distance measure that represents the base for the clustering and transcription estimating algorithms, the cost $S(a, b)$ of a substitution should depend on the similarity between the two substituted phonemes. A phoneme confusion matrix generated by the HTK tool HResults could be uses as a base for defining $S(a, b)$.

The parameters of the clustering and transcription estimating algorithms were found but manual examination of the algorithms' results. Using objective criteria to optimise these parameters will likely lead to better performance of these algorithms. But this parameter optimisation might be a very complex task.

To solve some of the occasionally occurring problems with initial and final silence in the training tokens, the network

```
$phn = (ax|ey| ... |dh);
([sp] < $phn > [sp])
```

could be used instead of the network described in Section 4.3.1. But the included optional silence models "sp" would lead to increased memory requirements because of the increased complexity of the network. If, as described in [15], the use of initial and final silence models should be possible independently for each of the different training tokens, large modifications of HViteM are needed since this would require to partly give up the versatile recognition network concept of HTK.

In this thesis, only the quite simple HMM set "base" was used in the lexicon generation experiments. It would be interesting to investigated the automatic lexicon generation when more advanced HMMs are used. If context dependent models instead of context independent models should be used, a much more complex network would be required.

Starting from a more advanced HMM set than "base" to avoid some obvious errors in the automatically obtained transcriptions, it would be interesting to study the joint HMM and lexicon optimisation described in Section 4.3.1.

The lexicon generation techniques described here try to find the transcription with the

maximum likelihood for the training tokens. This might lead to similar or even equal transcriptions for different words in the lexica. It would be interesting to study techniques that also try to increase the discrimination between different words.

Besides for the lexicon generation, the modified tree-trellis algorithm implemented in HViteM could also be used for totally other applications. If e.g. problems in the recognition of a spoken utterance are encountered, the speaker could be prompted to repeat the utterance and then both utterances could be included in the search for the most likely hypothesis for that utterance.

Chapter 5

Conclusions

In this thesis, an HMM based system for speaker independent recognition of continuous speech was studied. Different modifications of the recognition system were implemented and their effect on the system's performance was investigated.

First, the fundamentals of speech recognition were briefly reviewed and the HMM Toolkit (HTK) as well as its Viterbi recogniser were introduced. Then, a sub-word based recognition system for the 1000 word DARPA resource management task was described. It uses phonemes as basic recognition units and is based on HTK. This system represents the base for the work done in this thesis.

In the first part of this thesis, the N-best search paradigm was presented. This concept simplifies the incorporation of additional knowledge sources in the recognition process. It requires an N-best algorithm to produce a list of the best N hypotheses for a spoken utterance. These hypotheses can then be rescored by the additional knowledge sources and thus these sources can be included in the recognition process. Different N-best algorithms have been proposed recently and were described here. Some of these algorithms can generate an exact N-best list while others use different approximations to reduce computation. These different algorithms were compared with respect to their optimality and efficiency.

Much effort went into the implementation of the tree-trellis algorithm, an exact and efficient N-best algorithm. This implementation is based on HTK's Viterbi recogniser for continuous speech. Several adaptations of the original tree-trellis algorithm were required to include all the options of HTK's original recogniser in this new recogniser. The implementation was optimised in different ways in order to obtain maximum performance. Finally, this N-best recogniser was tested on the 1000 word DARPA resource management task. Different HMMs and grammars were used for these tests and the obtained results correspond to those known from literature. Also the computation and memory requirements were examined.

The tree-trellis recogniser developed as a part of this thesis work represents a new and powerful tool for the HTK system. It is fully compatible with HTK's original Viterbi recogniser and provides a good base for further work on N-best based recognition systems.

In the second part of this thesis, different techniques for the automatic generation of the lexicon required in a sub-word based recogniser were presented. This lexicon contains transcriptions in terms of basic recognition units for each word in the recogniser's vocabulary. To be able to generate a new lexicon entry for a word, a set of training utterances of that

word is required. Although possible, other information like the spelling of the word was not taken into account here. A modified version of the tree-trellis N-best algorithm was used to find the lexicon entry (transcription) that is optimal with respect to its likelihood for the training utterances of the word.

The implementation of the tree-trellis N-best algorithm was modified and extended to allow the search for the most likely hypotheses given not only a single utterance but a set of utterances. This new program was optimised in order to increase performance and minimise the memory requirements. Despite of this optimisation, the search for a new lexicon entry for a word was in several cases too complex for the available hardware. Therefore, different approximative techniques reducing the complexity of this search were developed and examined. If the number of training utterances is limited by using only a randomly selected subset of the available utterances, the reliability and performance of the new lexicon entry is reduced. More advanced techniques are based on the lists of N-best transcriptions for all individual training utterances of a word. With these lists, it is possible to estimate the most likely transcription for all utterances. Another technique is based on a string clustering algorithm. The transcriptions in the N-best lists are clustered and the biggest cluster is expected to contain the transcriptions of those training utterances that represent the most common pronunciation of the word. This subset of training utterances is then used to find the most likely transcriptions using the modified tree-trellis algorithm.

Finally, several new lexica for the phoneme based recogniser for the 1000 word DARPA resource management task were generated. They were used to compare the performance of the different automatic lexicon generation techniques mentioned above. The average likelihood scores for the different transcriptions of a word and the results of full recognition tests using the new lexica were used in this comparison. It was found that the lexicon generation technique taking into account all available training utterances offered the highest increase in performance compared to the original lexicon. The lexicon based on the string clustering technique offered a somewhat lower increase in performance. The other techniques were less promising, but nevertheless all automatically generated lexica offered a higher performance than the original lexicon.

Different problems of these automatic lexicon generation techniques were discussed and several suggestions for further improvements were presented. Much effort went into the development and optimisation of the various programs that were required to implement the different techniques mentioned above. These programs provide a good base for further work and refinement of techniques for automatic lexicon generation.

References

[1] L.R. Rabiner, B.-H. Juang: *Fundamentals of Speech Recognition.* Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, 1993.

[2] A. Krokstad, J. Tro: *Speech and Music Technology* (in Norwegian). Scripts for a Lecture, Department of Telecommunications, Norwegian Institute of Technology, 1994.

[3] L.R. Rabiner: *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition.* Proceedings of the IEEE, Vol. 77, No. 2, February 1989, pp. 257–286.

[4] S.J. Young: *HTK: Hidden Markov Model Toolkit V1.5* (User, Reference & Programmer Manual). Cambridge University Engineering Department Speech Group and Entropic Research Labs Inc., 1993.

[5] S.J. Young, N.H. Russell, J.H.S. Thornton: *Token Passing: a Simple Conceptual Model for Connected Speech Recognition Systems.* CUED Technical Report F-INFENG/TR38, Cambridge University, 1989.

[6] N.J. Nilsson: *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[7] R. Schwartz, Y.-L. Chow: *The N-Best Algorithm: An Efficient and Exact Procedure for Finding the N Most Likely Sentence Hypotheses.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1990, pp. 81–84, Albuquerque, April 1990.

[8] R. Schwartz, S. Austin: *A Comparison of Several Approximate Algorithms For Finding Multiple (N-BEST) Sentence Hypotheses.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1991, pp. 701–704, Toronto, May 1991.

[9] F.K. Soong, E.-F. Huang: *A Tree-Trellis Based Fast Search for Finding the N Best Sentence Hypothesis in Continuous Speech Recognition.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1991, pp. 705–708, Toronto, May 1991.

[10] S. Austin, R. Schwartz, P. Placeway: *The Forward-Backward Search Algorithm.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1991, pp. 697–700, Toronto, May 1991.

[11] R. Schwartz et al.: *New Uses for the N-Best Sentence Hypotheses within the BYBLOS Recognition System.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1992, pp. I-1–I-4, San Fransisco, March 1992.

[12] E.-F. Huang, F.K. Soong, H.-C. Wang: *The use of tree-trellis search for large-vocabulary Mandarin polysyllabic word speech recognition.* Computer Speech and Language, Vol. 8, No. 1, pp. 39–50, January 1994.

[13] J.-K. Chen, F.K. Soong: *An N-Best Candidates-Based Discriminative Training for Speech Recognition Applications.* IEEE Transactions on Speech and Audio Processing, Vol. 2, No. 1, Part II, pp. 206–216, January 1994.

[14] N.H. Madsen: *"N-best" speech recognition* (in Norwegian). Thesis (Hovedoppgave), Department of Telecommunications, Norwegian Institute of Technology, 1994.

[15] T. Svendsen: *Optimal acoustic baseforms.* Internal Technical Memo, Department of Telecommunications, Norwegian Institute of Technology, 1991.

[16] J.M. Lucassen, R.L. Mercer: *An Information Theoretic Approach to the Automatic Determination of Phonemic Baseforms.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1984, pp. 42.5.1–42.5.4, San Diego, March 1984.

[17] K.K. Paliwal: *Lexicon-Building Methods for an Acoustic Sub-Word Based Speech Recognizer.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1990, pp. 729–732, Albuquerque, April 1990.

[18] L.R. Bahl et al.: *Automatic Phonetic Baseform Determination.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1991, pp. 173–176, Toronto, May 1991.

[19] A. Asadi, R. Schwartz, J. Makhoul: *Automatic Modeling for Adding New Words to a Large-Vocabulary Continuous Speech Recognition System.* Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing 1991, pp. 305–308, Toronto, May 1991.

[20] S.Y. Lu, K.S. Fu: *A Sentence-to-Sentence Clustering Procedure for Pattern Analysis.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-8, No. 5, pp. 381–389, May 1978.

[21] L. Maugat: *Automatic Lexicon Generation in Speech Recognition using Hidden Markov Models.* Thesis, Department of Telecommunications, Norwegian Institute of Technology, 1992.

[22] B.W. Kerninghan, D.M. Ritchie: *The C Programming Language, Second Edition, ANSI C* (German edition). Hanser, München & Prentice-Hall International Inc., London, 1990.

Appendix A

Program Descriptions

This appendix contains user manuals for the main tools and utility programs that were written as a part of this thesis work. It is assumed that the user is familiar with the main HTK tools and their usage.

At the end of the program descriptions, the UNIX script files used for the automatic lexicon generation are described. As an example, the process of generating and testing a new lexicon is illustrated for the lexicon "clust100".

Because of their size, it is not possible to include program listings in this report. To give a rough impression of the size of the main tools written during this thesis, the lengths of their source codes are listed in Table A.1. These tools (except for wordclust) make use of the HTK library which contains totally about 9300 lines of source code and about 1700 lines in the header files.

Program	Source Code Lines
(HVite)	(2217)
HViteN	4012
HViteM	4547
(HAlign)	(1870)
HAlignW	1988
HBst2Lab	388
wordclust	1758

Table A.1: Source code lengths for the different main tools.

A.1 Main Tools

In this section, the main tools written during this thesis are described. These tools form an extension to the HMM Toolkit (HTK). Their data handling and user interface were designed such that they comply to the other HTK tools as far as possible.

The programs HViteN and HViteM originate from the HTK tool HVite, and HAlignW is a slightly modified version of HTK's HAlign. Therefore, only the differences in their usage and functioning are described here.

Both programs HViteN and HViteM are based on the version V1.5 of HVite and the HTK libraries. The first versions of these two programs were based on the eralier HTK version 1.4A, but they lack some the options that are available in HViteN and HViteM version V1.5. HAlignW is only available for version V1.5. HBst2Lab exists only for version V1.4A, but there is no need for upgrading it to version V1.5.

A.1.1 HViteN

The program HViteN implements the tree-trellis algorithm for the normal single utterance case. It is used like HTK's HVite recogniser and has the ability to find not only the first best hypothesis but the list of the first N hypotheses. The forward trellis search is performed exactly in the same way as in HVite. If the option `-N` is used, also the backward tree search is performed and the list of best hypotheses is written into an additional N-best label file. If more than one speech file is used, a full tree-trellis search is performed and a separate N-best label file is generated for each of the speech files. An N-best label file contains the hypotheses in the normal HTK label format. The beginning of each hypothesis is indicated by a line like

```
>> hypothesis 1 (-12.345678)
```

were the average likelihood score per time frame is given in parentheses. A full N-best label file might look like

```
>> hypothesis 1 (-12.345678)
         0     1200000     a
   1200000     2300000     b
   2300000     4500000     c
>> hypothesis 2 (-23.456789)
         0     1200000     a
   1200000     3400000     b
   3400000     4500000     d
```

were the first two columns give the start and end times in 100ns units and the third column gives the labels. The program HBst2Lab can be used to convert N-best label files into other formats.

Actually, there exist two versions of the program: HViteN and HViteNm. The only difference is that the version HViteNm was compiled with a special option and hence uses `float` instead of `double` and `short` instead of `int` for RankLists and partial path scores and times allocated during the forward and backward search. This leads to reduced memory requirements while the lower accuracy only gives rise to very small differences in scores and does not cause practical problems.

HViteN is used in the following way:

```
USAGE: HViteN[V1.5] [options] hmmList netFile speechFiles...

 Option                                        Default

 -a      enable demo mode                      off
 -b f    load bigram from file f               none
 -c f    tied mixture pruning threshold        10.0
```

```
-d s    dir to find hmm definitions          current
-e      exit if pruning WARNING              continue
-f      use full grammar (ENTER & EXIT)      off
-i s    Output transcriptions to MLF s       off
-k      no label times (N-best mode)         label times
-l      output labels at frame centres       frame ends
-m      add label scores in output           off
-n      disable bigram triphone-stripping    on
-o      disable overwriting                  on
-p f    inter model trans penalty (log)      0.0
-q f    set pruning threshold (backward)     off
-s f    grammar scale factor                 1.0
-t f    set pruning threshold                0.0
-u i    set pruning max active               0
-v f    set word end pruning threshold       0.0
-y s    external program used in demo mode   none
-x s    extension for hmm files              none
-B s    extension for N-best label files     bst
-F fmt  Set data file format to fmt          HTK
-H s    Set master input model file s        none
-L s    dir to store label file(s)           current
-N N    find N best hypotheses               off
-P fmt  Set output label file format to fmt  HTK
-R N    max N entries in fwd. rank list      N best
-S f    set script file to f                 none
-T N    set trace flags to N                 0
-X s    extension for label files            rec
```

HViteN offers the same options as HVite and additionally has several new options to control the generation of the N-best hypotheses in the backward tree search. Here, only those options are listed which are new or different from HVite.

`-e` Abort the program if a pruning warning is issued.

`-f` Use also the transition probabilities from the network ENTER node and to the network EXIT node when calculating the score of a full path. This option can also be used together with a bigram file.

`-i s` This option does *not* affect the N-best label files. Only the first best hypothesis found in the forward search can be written to an MLF.

`-k` The N-best label file will not contain start and end times (-1 is written to file as "dummy" value). This leads to reduced memory requirements for the stack.

`-m` The label scores are also added in the N-best label file. Due to a hardly fixable bug in HVite, the label scores for the best hypothesis in the forward and backward search might differ slightly if a bigram file is used.

`-q f` Enable backward beam search with a pruning threshold `f`. This option has the same effect as `-t f` has on the forward search.

`-B s` This sets the extension for the N-best label file(s) (default: "bst").

`-N N` This option enables the full tree-trellis search and specifies the number `N` of hypotheses to generate in the backward search.

`-P fmt` This option does *not* affect the format of the N-best label file(s).

`-R N` The number of entries in the predecessor RankLists generated during the forward search is limited to the best `N` entries. The default value is the number N of best hypotheses to generate. If no bigram file is used, this option can lead to reduced memory requirements.

`-T N` Set the trace level to `N`. `N` is treated as a vector of binary flags. Besides the flags used in HVite

```
0000002   recognition network
0000004   garbage collection
0000010   beam width
0000020   phone instances
0000040   phone link records
0000100   output probs
0000200   recognition output
0000400   network memory stats
```

several new flags are added:

```
0001000   rank lists gen'ed in forward search
0002000   TOS entry
0004000   full stack contents
0010000   backward Viterbi
0020000   backward Viterbi PIs
0040000   backward Viterbi scores & times
0100000   rank lists generation in backward search
0200000   recognition memory statistics
0400000   stack entries removed or deleted
```

A.1.2 HViteM

The program HViteM implements the tree-trellis algorithm for the multiple utterance case and is the main tool used for automatic lexicon generation. The program is based on HViteN and modified in such a way that it searches for the N-best hypotheses that have the highest combined likelihood score averaged for all given utterances. Normally, all given speech files are regarded as single utterances. But if the option `-w` is used to specify a word label, all tokens of this word are extracted from the given speech files and then these tokens are regarded as separate utterances. In this case, also word label files including label start and end times must be available. They can be specified using the options `-I`, `-V` and `-Y`. Such time-aligned word label files can e.g. be generated by HAlignW.

Like HViteN, HViteM first performs a forward trellis search independently for all utterances. The first best hypotheses for each of the utterances are written to a single multi label file having the result file name with the label file extension. The beginning of each utterance is indicated by a line like

```
-- sequence 1 (-12.345678)
```

and the hypotheses are in the normal HTK label format. The average likelihood score per time frame is given in parentheses. At the end of the file, a line like

```
++ WORD -12.345678
```

is added. It gives the average likelihood score per time frame for all utterances. `WORD` is the word label specified by `-w`.

After the the separate forward trellis searches are performed for all utterances, a single common backward tree search is performed to find the N-best hypotheses that have the highest combined score for all utterances. The hypotheses are then written to the result file. The format of this result file is similar to the N-best label file format used by HViteN. The beginning of each hypotheses is indicated by a line like

```
>> hypothesis 1 (-12.345678)
```

were the average likelihood score per time frame for all utterances is given in parentheses. Then the labels for the utterances follow, only differing in their time-alignment. The beginning of each utterances is indicated by a line like

```
-- sequence 1 (-12.345678)
```

containing also the average likelihood score for that utterance.

Both label files generated by HViteM can be converted to other formats by HBst2Lab. There are different options (`-g` and `-Z`) to specify that only a subset of the given utterances is used by HViteM. If the option `-z` is used, only the forward searches are performed and only the multi label file is generated. The option `-h` selects a special mode were the N-best hypotheses for each utterance are found by independent backward searches and no common backward search is performed. These hypotheses are written to the file specified with the option. The file format of this multi one-line file is similar to the one-line format use by HBst2Lab. For each utterance, the file contains a set of $N + 1$ lines like

```
## 1
-12.345678 a b c
-23.456789 a b d
```

The first line indicates the index of the utterance (sequence) and the following lines give the N-best hypotheses, starting with the best one. Each hypothesis line starts with the average likelihood score per time frame and continues with the label sequence of that hypothesis.

Like for HViteN, there also exist two versions of this program: HViteM and HViteMm. They differ in exactly the same way as described for HViteN.

HViteM is used in the following way:

```
USAGE: HViteM[V1.5] [options] hmmList netFile resultFile speechFiles...

 Option                                        Default

 -b f    load bigram from file f               none
 -c f    tied mixture pruning threshold        10.0
 -d s    dir to find hmm definitions           current
 -e      exit if pruning WARNING               continue
 -f      use full grammar (ENTER & EXIT)       off
 -g s    load sequences indexed in file s      all
 -h s    only each sequence N-best to file s   off
```

```
-k      no label times (N-best mode)        label times
-l      output labels at frame centres      frame ends
-m      add label scores in output          off
-n      disable bigram triphone-stripping   on
-p f    inter model trans penalty (log)     0.0
-q f    set pruning threshold (backward)    off
-r      use rel. word score (prob/frame)    abs. score
-s f    grammar scale factor                1.0
-t f    set pruning threshold               0.0
-u i    set pruning max active              0
-v f    set word end pruning threshold      0.0
-w s    set word label to s                 none
-x s    extension for hmm files             none
-z      calc. average word score only       off
-F fmt  Set data file format to fmt         HTK
-H s    Set master input model file s       none
-I s    Set master label file s             none
-K N    max N entries in open stack         N best
-L s    dir to store label file(s)          current
-N N    find N best hypotheses              only 1 best
-R N    max N entries in fwd. rank list     all
-S f    set script file to f                none
-T N    set trace flags to N                0
-V s    dir to find word label file(s)      current
-X s    extension for label files           mre
-Y s    extension for word label files      wrd
-Z N    load max N sequences                all
```

HViteM offers most of the options of HViteN. To handle the multiple utterance case, several options have been added. But is was also necessary to remove some of the original HVite options. Here, only those options are listed which are new or different from HVite.

`-a` (can *not* be used)

`-e` Abort the program if a pruning warning is issued.

`-f` Use also the transition probabilities from the network ENTER node and to the network EXIT node when calculating the score of a full path. This option can also be used together with a bigram file.

`-g s` Only utterances are loaded and used whos index can be found in the file `s`. This file must contain one index number per line and has to be sorted. The utterances (whole speech files or, if `-w` is used, tokens) are indexed in the order they appear in the input, starting with index 1. The indices appearing in any HViteM output always refer only to the actually loaded utterances.

`-h s` This option disables the common backward search. Instead of the common search, independent backward searches are performed for all utterances. The N-best hypotheses for each utterance are written to the file `s` in the format described above.

`-i` (can *not* be used)

`-k` The result file will not contain start and end times (-1 is written to file as "dummy" value). This leads to reduced memory requirements for the stack. The label sequence for each hypothesis is only written once for the last utterance and not for all utterances.

`-m` The label scores are added in the multi label file and in the result file. Due to a hardly fixable bug in HVite, the label scores for the best hypothesis in the forward and backward search might differ slightly if a bigram file is used.

`-o` (can *not* be used)

`-q f` Enable backward beam search with a pruning threshold `f`. This option has the same effect as `-t f` has on the forward search.

`-r` The average likelihood score for all utterances can be calculated in two ways. Normally, the scores for all utterances are added and the divided by the total length of all utterances. If the option `-r` is used, first the average score per time frame is calculated separately for each word and finally the mean of these average scores is calculated. This option affects both the average score written to the multi label file and the score used during the backward search.

`-w s` If this option is used, a speech file is not regarded as a single utterance. Instead of a whole speech file, now only those tokens labeled `s` are extracted from a speech file and then treated as independent utterances. This option requires a corresponding label file for every speech file in the input. These label files can be specified with the options `-I`, `-V` and `-Y`.

`-y` (can *not* be used)

`-z` This option disables the common backward search. Hence, only the multi label file is written. This option can not used together with `-h`.

`-I s` This option load the master label file (MLF) for the word label files.

`-K N` This option specifies the maximum number of entries in the OPEN stack. If this number is exceeded, the worst entries are moved to the CLOSED stack. An entry in the CLOSED stack requires significantly less memory than an entry in the OPEN stack. Thus, memory requirements are reduced.

`-N N` This option specifies the number `N` of hypotheses to be generated in the backward search(es). As default, only the first best hypothesis is generated in the backward search.

`-P fmt` (can *not* be used)

`-R N` The number of entries in the predecessor RankLists generated during the forward search is limited to the best `N` entries. If no bigram file is used, this option can lead to reduced memory requirements.

`-T N` Set the trace level to `N`. `N` is treated as a vector of binary flags. Besides the flags used in HVite

```
00000002   recognition network
00000004   garbage collection
00000010   beam width
00000020   phone instances
00000040   phone link records
00000100   output probs
00000200   recognition output
00000400   network memory stats
```

several new flags are added:

```
00001000   rank lists gen'ed in forward search
00002000   TOS entry
00004000   full stack contents
00010000   backward Viterbi
00020000   backward Viterbi PIs
00040000   backward Viterbi scores & times
00100000   rank lists generation in backward search
00200000   recognition memory statistics
00400000   stack entries removed or deleted
01000000   loaded speech data sequences
02000000   closed stack garbage collection
```

`-V s` This option specifies the directory that is searched for the word label files. If this option is not used, the word label files are expected in the same directory as the speech files.

`-X s` Set the extension for the multi label file (default: "mre").

`-Y s` Set the extension for the word label files (default: "wrd").

`-Z N` This option allows to specify a limit for the number of utterances actually used by HViteM. It can be combined also with the options `-g` and `-w`.

A.1.3 HAlignW

HTK's original tool HAlign is a modified version of HVite that finds a phone or state alignment for a speech file with a given non-aligned label file. A network file can be specified so that word pronunciation subnets can be used if the input label file contains only a word level transcription. But also in this case, the output label file will still give an aligned phone level transcription. To obtain also aligned word level transcriptions as output, a modified version, HAlignW, was written.

HAlignW is used in the following way:

```
USAGE: HAlignW[V1.5] [options] hmmList speechFiles...

 Option                                       Default

 -a s    insert initial/final silence s        none
 -c f    tied mixture pruning threshold        10.0
 -d s    dir to find hmm definitions          current
 -e s    dir to store output labels           current
 -i s    output transcriptions to MLF s        off
 -f      do full state alignment               off
 -l      output labels at frame centres       frame ends
 -m      append log probs for each state       none
 -n s    load network from s                   none
 -o s    extension for output label files       seg
 -s s    insert interword/model silence s      none
 -t f    set pruning threshold                  0.0
 -w      output word transcription only        off
 -x s    extension for hmm files               none
 -y s    external program used in demo mode    none
 -F fmt  set data file format to fmt           HTK
```

```
-G fmt  Set label file format to fmt       HTK
-H s    Set master input model file s      none
-I s    set master label file s            none
-L s    dir to find label file(s)         current
-P fmt  Set output label file format to fmt HTK
-S s    set script file to s               none
-T N    set trace flags to N                0
-X s    extension for input label files    lab
```

Nearly all options in HAlignW work in the same way as in HAlign. Here, only those options are listed which are new or different from HAlign.

`-s s` The inter-word silence model `s` is inserted between each source label as in HAlign. Additionally, the model `s` is removed from all word pronunciation subnets. Thus, the true word end times are written to the aligned label file even if the word subnets contain optional final inter-word silence models.

`-w` If this option is used, only an aligned word level label file is generated.

A.1.4 HBst2Lab

The program HBst2Lab is used to convert N-best label files, multi label files or result file generated by HViteN or HViteM into other file formats. Single hypotheses or sequence (i.e. utterance) transcriptions can be extracted and written to separate files which obey to the HTK label file format. Additionally, one-line files can be generated. They contain a full hypothesis (label transcription) per line. Such a line looks like

```
-12.345678 a b c
```

and starts with the average likelihood per time frame followed by the label sequence of the hypothesis.

HBst2Lab is used in the following way:

```
USAGE: HBst2Lab[V1.4] [options] bstFiles...

 Option                                Default

 -a      extract all sequences         one sequence
 -s N    extract sequence N            1st sequence
 -z s    dir to store one-line files   current
 -L s    dir to store label files      current
 -N N    write N best label files      off
 -S f    set script file to f          none
 -T N    set trace flags to N          0
 -X s    extension for label files     rec
 -Z s    extension for one-line files  lin
```

The operation of HBst2Lab can be controlled by the following options:

`-a` This option causes all transcriptions found in the input files to be written in the one-line files. It should not be used together with options like `-s` and `-N` and is mainly intended to be used with multi label files.

`-s N` This options specifies which sequence (i.e. utterance) is extracted from a HViteM result file or a multi label file.

`-z s` Set the directory to store the one-line files. The directory of the input file is used as default.

`-L s` Set the directory to store the label files. The current directory is used as default.

`-N N` If this option is used, the `N` best hypotheses are extracted and written as separate HTK label files. Their file names are build by appending `_1`, `_2`, ... to the base file name of the input file. The one-line file *always* contains all hypotheses found, not only the first `N`. If the input file is a multi label file, the number `N` is irrelevant and nothing is appended to the base file name.

`-S f` This enables the script HTK file `f`. An HTK script file contains one file name per line. These file names are appended to the command line.

`-T N` Set the trace level to `N`. Trace level 0 gives no trace output, 1 gives basic progress information and 2 gives maximum tracing.

`-X s` Set the extension for the label files (default: "rec").

`-Z s` Set the extension for the one-line files (default: "lin").

A.1.5 wordclust

The program wordclust implements several different string cluster algorithms. It is mainly intended to be used for the selection of training tokens in the automatic lexicon generation process. The input file has to be a list of strings. A line in the input file is assumed to consist of several field delimited by blank space. A string is then a sequence of fields in a single input line. Special options (`-f` and `-r`) are provided to allow multi one-line files generated by HViteM directly to be used as input files.

The clustering algorithms implemented here are based on a string distance measure which sets the costs of insertions, deletions and substitutions uniformly to 1. The distance between two strings is found using dynamic programming. The strings / words are assigned to the cluster with nearest centroid if this distance is not greater than the maximum intra-cluster distance. After the clustering, the generated clusters are sorted according to their size (number of members) so that the biggest cluster becomes the top cluster.

Finally, the generated string clusters and their centroids are written to the output file. The program can also handle sets of strings that make up separate words (e.g. N-best transcriptions of a token). A special custering method is provided to estimate the most likely transcription of a word based upon the token scores available in HViteM's multi one-line file. The program employs dynamic data structure to obtain reasonable short execution times.

wordclust is used in the following way:

```
Usage: wordclust [options] infile outfile
Options:
-f S  first field in lines between words     (1 word per line)
```

```
-r    read score before string              (off)
-h N  skip N fields before score/string     (0)
-d F  max. intra-cluster distance           (0.0)
-l F  aver. length factor for max. dist.    (0.0)
-p F  score penalty                         (0.0)
-x F  score penalty dist. factor            (0.0)
-y F  score penalty dist. exp.              (1.0)
-a F  min. rel. reduct. of tot. distortion  (-1.0 = off)
-i N  max. number of iterations             (unlimited)
-s N  max. num. of strings per word loaded  (all)
-w N  max. num. of strings per word written (all)
-c N  max. num. of clusters written         (all)
-n S  write top cluster indices to file S   (off)
-z S  write top cluster centroid to file S  (off)
-t N  trace output (0=off .. 3=full)        (off)
-m N  clustering method                     (0)
-o N  clustering options (bitvector)        (0)

method
     options
  0        word clustering
       +1    update centroid immediately
       +2    resort clusters after iteration
  1        string clustering
       +1    update centroid immediately
       +2    resort clusters after iteration
       +4    word output (word once in biggest cluster)
  2        string clustering (word max. once in cluster)
       +1    update centroid immediately
       +2    resort clusters after iteration
       +4    word output (word once in biggest cluster)
  3        opt. score sum
  4        sequential string clustering (-i sets max # clusts)
       +2    resort clusters
       +4    word output (word once in biggest cluster)
```

The operation of wordclust can be controlled by the following options:

`-f S` If this option is used, several strings can be loaded for each word. `S` is the first field in the lines separating the different words in the input file. For a multi one-line file, `'##'` should be used as parameter of `-f`.

`-r` If this option is used, the first (unskipped) field in a line is assumed to be the likelihood score of the string starting in the next field. The score information is only used by clustering method 3.

`-h N` Set the number of fields `N` in a line to be skipped before the string actually starts.

`-d F` Set the maximum intra-cluster string distance $d_{\max}$ to `F`.

`-l F` Increase the maximum intra-cluster string distance $d_{\max}$ by the average length of the loaded strings multiplied with the factor `F`.

`-p F` Set the score penalty p to `F`.

`-x F` Set the distance score penalty factor x to `F`.

`-y F` Set the distance score penalty exponent y to `F`.

`-a F` Set the minimum required relative reduction a of the total distortion to `F`. The clustering using method 0, 1 or 2 is ended if the relative reduction of the total distortion between two iterations is less than a.

`-i N` Set the maximum number of iterations i to `N`. If clustering method 4 is used, `N` specifies the maximum number of clusters to be generated.

`-s N` Load maximum `N` strings for each word.

`-w N` Write maximum `N` strings for each word in the output file.

`-c N` Write maximum `N` clusters to the output file.

`-n S` Write the indices of the strings / words in the top cluster to the file `S`. This file can be used with HViteM's option `-g`.

`-z S` Write top cluster centroid to the file `S`.

`-t N` Set the trace level to `N`. Trace level 0 gives no trace output, 1 gives basic progress information and 2 and 3 give more detailed tracing.

`-m N` Set clustering method to `N`.

`-o N` Set clustering options to `N`. The parameter `N` is treated as a vector of binary flags.

The following clustering methods are available:

`0` The loaded words are clustered using a nearest neighbour clustering algorithm. The distance between a word and a cluster centroid string is calculated as the sum of the string distances of the word's strings and the centroid.

`1` The loaded strings are clustered using a nearest neighbour clustering algorithm.

`2` This method is similar to method 1. The only difference is that here only one string per word is allowed to belong to a given cluster.

`3` This method tries to find the string with the maximum likelihood score when averaged for all words. For all loaded strings, this average score is calculated. If a given string is not within the set of strings belonging to a word, the score for that word is estimated. The estimate used is the minimum of the worst score in that word and the score of the most similar string reduced by $p + x(d_{\min}^{y})$, where $d_{\min}$ is the distance to the most similar string.

`4` The loaded string are clustered using an unsupervised sequential clustering algorithm.

The following binary flags in the option vector can be used:

`1` If this flag is set, the cluster centroid is updated immediately after a new string is added to the cluster. Normally, the centroids are updated the end of each iteration.

`2` The clusters are resorted after each iteration so that the biggest cluster comes to the top of the cluster list.

`4` After the clustering, the generated clusters are pruned in such a way that for each word only the string belonging to the biggest cluster is retained.

A.2 Utility Programs

Several smaller utility programs were written during this thesis work. They are described briefly now.

A.2.1 NResults

The program NResults is used to give recognition statistics for a set of N-best label files generated by HViteN. First, these files have to be converted to one-line file by HBst2Lab. NResults needs a file containing a list of one-line file names as input. Also an MLF (Master Label File) containing the correct transcriptions of the recognised utterances must be specified. Then, all listed one-line files are processed and it is counted how many correct hypotheses were found in the 1st, 2nd, ..., `maxN`'th rank. After all files have been processed, the values of the `maxN` counters are written to the result file. As a `maxN`+1'th line, the number of utterances with no correct hypotheses among the top `maxN` is appended.

NResults is used in the following way:

```
Usage: NResults maxN linfilelist MLFfile resultfile
```

A.2.2 bst2fig

The program bst2fig is used to convert an N-best label file generated by HViteN into a ".fig" data file for the graphic editor program xfig. Thus, a graphically presentation of the different N-best time-aligned hypotheses of an utterance can be obtained. `maxN` is the number of hypotheses shown, `correct` is the rank of the correct hypothesis and `maxt` is the length of the utterance in seconds.

bst2fig is used in the following way:

```
Usage: bst2fig maxN correct maxt bstfile figfile
```

A.2.3 lin2dct

The program lin2dct is used to convert a set of result files from HViteM into a lexicon (dictionary) in the RMTK format. First, the result files have to be converted to one-line files by HBst2Lab. When starting lin2dct, these files are expected in the current directory, having the name of the words in the lexicon as base names and `extension` specified in the command line. It is also necessary to specify a list of words for the lexicon to be generated. The `mode` specifies what to do if no new transcription for a word is found. In mode `c`, the word's entry in the word list is written to the new lexicon, assuming that the word list

file itself is e.g. another lexicon. Otherwise, `(no file)` or `(no entry)` is written. The resulting lexicon might look like

```
ABERDEEN        ae b er d iy n
ABOARD          ax b ao r dd
ABOVE           ax b ah v
...
```

lin2dct is used in the following way:

```
Usage: lin2dct mode extension wordlist outdict
mode 'c': copy wordlist entry if no new entry found
others  : (no file)/(no entry) if no new entry found
```

A.2.4 extractdct

The program extractdct extracts the entries for those words from the new lexicon (dictionary) that also have an entry in the reference lexicon and writes the resulting lexicon to stdout. If the option `-r` is used, the reference lexicon entry is written instead of the new lexicon entry. The order of the entries written is still determined by their order in the new lexicon. If the option `-j` is used, both lexica are joined. This means that, if a word in the new lexicon also has an entry in the reference lexicon, the reference entry is written, while otherwise the new entry is written.

extractdct is used in the following way:

```
Usage: extractdct [-r|-j] newdict refdect
```

A.2.5 cmpdct

The program cmpdct compares a new lexicon (dictionary) with a reference lexicon. The number of equal and different entries is found. Is is also counted how many of the new entries are `(no entry)` or `(no file)` and for how many new entries no reference entry was available.

cmpdct is used in the following way:

```
Usage: cmpdct newdict refdect
```

A.2.6 dct2net

The program dct2net takes the transcription of `word` in the lexicon `dict` and converts it into a network file in the HTK format. If the option `-s` is used, `sil` is added as an optional silence model to the beginning and end of the network. The generated network file might look like

```
([sil] a b c [sil])
```

dct2net is used in the following way:

```
Usage: dct2net [-s sil] word dict outnet
```

A.2.7 mksrc

The program mksrc generates an output file that, for each line of the input file, contains a line concatenating `progname` and the first field of the input file line. It is mainly used to generate shell script files that process all entries in a lexicon.

mksrc is used in the following way:

```
Usage: mksrc progname infile outfile
```

A.2.8 randhead

The program randhead is mainly used to scramble the lines of a file randomly. In a first step, randhead (in mode `+`) adds random numbers (5 digits) are as a first field to all input lines. Then, the new file is sorted using the UNIX tool "sort". Finally, randhead is used to remove the first field (the random number) from all input lines. If no input file is specified, stdin is used. The output is written to stdout.

randhead is used in the following way:

```
Usage: randhead mode [infile]
mode '+': add random numbers & blank as line header
others  : remove numbers & blanks as line header
```

A.2.9 wgrep

The program wgrep behaves similar to the UNIX tool "grep". It writes all those input lines to stdout the contain the field `string`. A field is delimited by blank space or the beginning or end of a line. If no input file is specified, stdin is used. If the option `-c` is used, only the number of lines found is written.

wgrep is used in the following way:

```
Usage: wgrep [-c] string [infile]
```

A.3 Script Files for the Automatic Lexicon Generation

In this section, the UNIX script files that were written to simplify the automatic generation of lexica are briefly described. Then, as an example, the process of generating and testing a new lexicon is illustrated for the lexicon "clust100".

A.3.1 Script File Descriptions

The UNIX script files described here are mainly used to set all the parameters needed by the different programs and thus avoid typing very long command lines manually. Most of the script file can handle only a single word (lexicon entry) at the time. Therefore they must be called for all words in the lexicon. The utility program mksrc simplifies this

process by generating a script file that calls another script file for each word in a lexicon (sending the word as a parameter to the called script file). cntmono is an example of a script file generated in this way.

Most of the script files here require a specific directory structure for the different data files. They normally also must be started from the correct current directory. But it should not be difficult to modify them so that they can be used with other directory structures. Some of the script files expect file names to be specified in shell variables. The script files listed here require access to several of the files that are generated when the basic HMM recogniser for the RM task is build according to the RMTK recipe. It also should be noted that most of the script files produce a log file.

- **clust_m1:** This script file must be called with a WORD as parameter. It requires the multi one-line file for that WORD, runs wordclust to find the biggest cluster for that word and writes the indices of the tokens in that cluster to a file called WORD.idx.
- **clust_m3:** This script file must be called with a WORD as parameter. It requires the multi one-line file for that WORD, runs wordclust to estimate the most likely transcription and append this transcription to a lexicon file called dct.
- **cntwrd:** This script file must be called with a WORD as parameter. It calls wgrep to count the number of tokens of a given WORD in the MLFs (master label files) containing the word-level transcriptions of the 4 different test sets and appends the counts found to the word count files for these test sets.
- **cntwrd_trn:** Same as cntwrd, but instead of the tokens in the test sets, the tokens in the training set are counted.
- **hlexr100m1:** This script file must be called with a WORD as parameter. It runs HViteMm to find the most likely transcription for the tokens of the given WORD specified in the file WORD.idx. Then HBst2Lab is called to generate the file WORD.lin containing only a single line with the transcription found. Several parameters and files needed by HViteMm are specified in this script file. Similar script files are used to generate the other lexica with different parameters for HViteMm.
- **hwalign:** This script file calls HAlignW to obtain a time-aligned word-level MLF (master label files) for the whole training set. The not time-aligned word-level MLF provided by the RMTK is needed here.
- **hwalign_test:** Same as hwalign, but generates the time-aligned word-level MLFs for the 4 test sets.
- **rtest:** This script file runs the RMTK tool HSnor2Net to generate an HTK network file based on the lexicon in the file dct and using the word-pair grammar available in RMTK. Hereafter, HVite and HResults are executed to obtain the recognition statistics for the lexicon dct. This is done similar to recognition tests in the RMTK recipe. The test set to be used must be specified as a parameter of this script file.
- **tscore:** This script file must be called with a WORD as parameter. It first calls dct2net to generate a network file according to the transcription of WORD in the lexicon specified in the shell variable WDCT. Then HViteM is called to find the

scores of the tokens of WORD in the test set specified by the shell variable WTEST for the given transcription in the lexicon. The token scores (lines starting with --) and the average score for WORD (line starting with ++) are copied to the log file.

- **wrlist:** This script file must be called with a WORD as parameter. It runs HViteM to generate the multi one-line file for the WORD. This file is called WORD.n10 and contains the 10-best transcriptions and their scores for all tokens of the WORD in the training set.
- **wscore:** Same as tscore, but only the average word score is copied to the log file.

A.3.2 Generating and Testing the Lexicon "clust100"

To give an example of how the different programs and script files are used for the automatic lexicon generation, the process of generating and testing the lexicon "clust100" is describe here. This description does *not* care about the details of the directory structure for the different data files. To indicate the computation requirements, the CPU time needed to run the programs on a *SUN SPARCstation IPX* is given in "hours:minutes" here.

First, a randomly scrambled version of the HTK script file ind_trn109.scp containing the file names of all the 3990 speech files in the training set is generated using the utility program randhead. This file causes that a *random* subset of the training tokens for a word is used when the number of tokens is limited to a given value.

```
randhead + ind_trn109.scp | sort | randhead - > rand_ind_trn109.scp
```

It is necessary to have time-aligned word-level transcriptions of the training utterances to be able to cut out the different tokens of a word from the training set. They are obtained using the script file hwalign (00:50 CPU time). Also hwalign_test should be called to get time-aligned word-level transcriptions of the test utterances.

```
source hwalign
source hwalign_test
```

Now the 10-best transcriptions (hypotheses) for all tokens in the training set are generated using the script file wrlist (15:15 CPU time). The original lexicon mono.dct is only used as a list of all 991 possible words.

```
mksrc wrlist mono.dct rankall
source rankall
```

The number of tokens for the different words in the training set is found using the script file cntwrd_trn. The the subset of the word list containing all words that have at least one token in the training set is generated.

```
mksrc cntwrd_trn mono.dct cntall
source cntall
grep -v ' 0' cntwrd.out > cntwrd.out_ge1
```

The script file clust_m1 is used to perform for each word the clustering based on the tokens' 10-best transcriptions and to generate the list of the tokens belonging to the biggest cluster (00:50 CPU time).

```
mksrc clust_m1 cntwrd.out_ge1 clustall
source clustall
```

Based on the output of the clustering, the most likely transcriptions for all words are found using the script file hlexr100m1 (10:50 CPU time).

```
mksrc hlexr100m1 cntwrd.out_ge1 genall
source genall
```

The ".lin" files generated by hlexr100m1 for every word are used to generate the new lexicon max100m1.dct. The list of those words where the automatic lexicon generation failed is written to the file FAILED_WORDS (which should be empty). For those words where no lexicon entry could be generated, the original lexicon entry is copied from mono.dct and finally the complete lexicon is written to the file clust100.dct.

```
lin2dct a lin cntwrd.out_ge1 max100m1.dct
grep '(no' max100m1.dct > FAILED_WORDS
grep -v '(no' max100m1.dct > max100m1.dct_found
extractdct -j mono.dct max100m1.dct_found | sort > clust100.dct
```

To measure the performance of the new lexicon, the likelihood scores of the new transcriptions are calculated for all tokens in a test set (here: feb89) using the script file tscore (01:30 CPU time). At this point, the time-aligned word-level MLF for the test set (as generated by hwalign_test) is required. The likelihood scores for all tokens in the test set and the average likelihood scores for all words in the test set are extracted from the log file. The files feb89_score.token and feb89_score.word can be quite easily converted to a file format that can be loaded e.g. by MATLAB for further processing.

```
mksrc tscore mono.dct scoreall
set WDCT = clust100.dct
set WTEST = feb89
set WLOG = feb89_score.log
source scoreall
grep -e '--' feb89_score.log | randhead - > feb89_score.token
grep -e '++' feb89_score.log | randhead - > feb89_score.word
```

Finally, a recognition test using the new lexicon and a word-pair grammar is performed by the script file rtest (02:30 CPU time). The recognition statistics obtained are copied to a file called results.

```
cp clust100.dct dct
source rtest feb89
tail -7 feb89.log > results
```