Continuous speech recognition in the WAXHOLM dialogue system

Ström, N.

journal: TMH-QPSR
volume: 37
number: 4
year: 1996
pages: 067-096

http://www.speech.kth.se/qpsr
Continuous speech recognition in the WAXHOLM dialogue system

Nikko Ström

Abstract
This paper presents the status of the continuous speech recognition engine of the WAXHOLM project. The engine is a software only system written in portable C code. The design is flexible and different modes for phonetic pattern matching are available. In particular, artificial neural networks and standard multiple Gaussian mixtures are implemented for phone probability estimation, and for research purposes, a general model where the input consists of a phone-graph also exists. A lexicon with multiple pronunciations for many words and a class bigram-grammar is used. The lexicon and grammar constraints are represented by a lexical graph, optimized for efficient lexical decoding. The decoding is performed in a two-pass search. The first pass is a Viterbi beam-search and the second is an A* stack-decoding search. Pruning-strategies and memory management in the two passes are discussed in the report. Several different output formats are available. Results can be reported either on the word or phoneme level with or without the time alignment information. Multiple hypotheses can be output either as standard N-best lists or in a more compact word-graph format. Continuous speech recognition can be performed on a standard UNIX workstation in real-time with a lexicon of about 1000 words.

Introduction
The WAXHOLM man-machine dialogue demonstrator is built on a generic framework for man-machine spoken dialogue under continuous development at the speech group at our department of KTH. The demonstrator is currently mature enough to be displayed and tested outside the laboratory by completely novice users. A successful such attempt was made at “Tekniska Mässan” (the technology fair) in Älvsjö in October ’96 where visitors with no prior experience with the system were invited to try the demonstrator in a rather noisy environment. The domain of the WAXHOLM application is boat traffic and tourist information about hotels, camping grounds and restaurants in the Stockholm archipelago. It references timetables for a fleet of some twenty boats from the Waxholm company, which connects about two hundred ports. The user input to the system is spoken language exclusively, but the responses from the system include synthetic speech as well as pictures, maps, charts and timetables (Figure 1). The application has similarities to the ATIS domain within the ARPA community, the Voyager system from MIT (Glass et al., 1995) and European systems such as SUNDIAL (Peckham, 1993), Philips’s train timetable information system (Aust et al., 1994; Oerder & Aust, 1994) and the Danish dialogue project (Dalsgaard & Baekgaard, 1994). Overviews of the WAXHOLM dialogue system and the WAXHOLM project database can be found in (Bertenstam et al. 1995a; Berstenstam et al. 1995b) and an early reference is Blomberg et al. (1993).

One of the key technologies of the application is automatic speech recognition (ASR). This report describes the speech recognition currently in use in the system. It is structured in the four main building blocks identified in Figure 2. The blocks are arranged in a linear chain of processing and the representations of an
utterance passed from each block to the next changes gradually from describing the characteristics of the acoustic signal in the early blocks to a symbolic description in the last block. The report will describe in detail the processing performed in each block. The discussion is focused on the solutions of the WAXHOLM system that we believe are original and innovative, but detailed information is provided about all parts of the ASR system.

Timing is very important in human-human dialogue and certainly also in a man-machine system. It is important that the machine response is not delayed too much after the user has stopped talking. Therefore, the chain of modules of Figure 2 works as a pipeline; each block passes processed information on to the next block as soon as it is ready. This process operates in real time when the user speaks. Thus, most of the processing is already finished at the end of the utterance and the system can respond almost instantly. Real time considerations are discussed in several sections, in particular in the section “Beam pruning”.

The block structure of the system allows rapid evaluation of new ideas. In Figure 3 it is seen how different implementations of the blocks can be combined in a flexible manner. This is important as the system is a research tool under constant evolution. In particular, the second block modelling phonetic classification, has three modes for competing implementations:

1) A hidden Markov model (HMM) mode, implementing the currently dominating ASR technology for medium to large dictionary tasks (Woodland et al., 1990; Murweit et al., 1993, 1994).
2) An artificial neural network (ANN) mode, implementing a hybrid HMM/ANN recogniser (Bourlard & Morgan, 1993; Cohen et al., 1992) where the output activities of the ANN are treated as phoneme probabilities.

3) A very general mode where the input is a phonetic graph with phoneme probabilities on the arcs.

These three modes are described in the section “Feature extraction”.

The different versions of the last block correspond to different types of output, varying from a simple output of the most likely word-string, via N-best lists of hypotheses, to the more complex word-graph format that contains many ranked hypotheses in a compact graph format optionally including the phonetic alignments.

Signal processing

The first block of Figure 2 has an analogue in human speech perception analogue in the early processing at the cochlea (von Békésy, 1960; Greenberg, 1988; Seneff, 1988). Here, the time-domain speech-signal is transformed to a representation in the frequency domain in a rather standard way. The short-time spectrum of the signal is computed using the FFT (Fast Fourier Transform). We have implemented a particularly fast version of the FFT, described by Bergland & Dolan (1979). The FFT algorithm is further modified to benefit from the constraint that the speech signal is real-valued, yielding a reduction by almost half of the processing time (Press et al., 1992).

The speech signal is divided into short overlapping frames as shown in Figure 4. The frame-rate is set to 100 frames per second. Thus, it is roughly one order of magnitude higher than the phoneme-rate and the same order as the glottal-pulse rate of a male speaker. The DC offset in each frame is removed and pre-emphasis is applied to the signal in the frame using the following difference equation

\[ s'_n = s_n - k s_{n-1} \]

on the samples \( s_{n \in \{1, N\}} \) of the digital signal, where \( N \) is the number of samples and \( k \) is a constant that we have set to 0.97. The signal is then Hamming windowed, i.e.,

\[ s''_n = \left( 0.54 - 0.46 \cos \left( \frac{2\pi (n-1)}{N-1} \right) \right) s'_n \]

Finally the signal is padded with zero-valued samples to the nearest power of two samples and the FFT transform is applied to get the magnitude spectrum of the signal in each frame. Currently, we are using speech signals sampled at 16 kHz, a Hamming window of 25 ms and a step size of 10 ms, i.e., a 512 point FFT is used where the first 400 points are non-zero. In Figure 4, the different windows are shown together with an example of a speech signal.

This partitioning into frames is the dominating method today, but other approaches have been studied and implemented in ASR systems (Zue et al., 1989; Stevens, 1995; Lewin, 1996). In the phone-graph mode, our ASR engine supports such more general feature extraction schemes (see the section: “Phonetic classification”).

In addition to the frequency representation, the logarithm of the energy of the signal in the frame is also computed using the formula

\[ E = \log \sum_{n=1}^{N} s_n^2 \]

Both the energy computation and the magnitude spectrum computation have been designed so that if appropriate values are chosen for the parameters, the processing of the feature extraction in the wide-spread HTK tool-kit (Young et al., 1995) can be reproduced. This is useful for comparisons with main-stream ASR technology.

---

**Figure 4. Signal processing time constants and a sample speech signal. The three different time scales are 1) sample points, the sample frequency is 16k Hz, 2) ms and 3) frames, the unit used by the higher level modules. The features of a frame are computed from the signal convoluted with a Hamming window of 25 ms giving an effective window of about 10ms.**
Feature extraction

In the second block in Figure 2, the so called feature extraction, the short-time spectra of each frame are mapped to a more compact representation by a compressing transform. This is done in two steps: first a filterbank is applied to the FFT spectrum and then the cosine transform is applied to the vector of filterbank outputs. The filterbank consists of a number of overlapping triangular, equidistantly spaced filters on one of the perceptually motivated frequency scales Mel or Bark (Fig. 4). The Bark scale (Zwicker & Feltkeller, 1967) is approximated by the formula

\[ 7 \cdot a \sinh \left( \frac{f}{650} \right) \]

where \( f \) is the frequency in Herz, and

\[ 2595 \cdot \log_{10} \left( 1 + \frac{f}{700} \right) \]

is used for the Mel scale (Young et al., 1995, p. 69). This is similar to the formula given by Schroeder, Atal & Hall (1979). Currently, 24 Mel spaced filters covering the frequency range 0-8000 Hz are used. The filterbank representation is similar to a standard spectrogram, but the resolution, both in the time and frequency dimension, is lower (Fig. 5).

![Figure 5. Top: Mel scaled filterbank with triangular filters. Bottom: Illustration of the resulting time and frequency resolution in a spectrogram-like plot. The uttered word is “Waxholm” (våkhɔ:lm).](image)

The filterbank outputs of the speech signal are statistically highly correlated, indicating that further data reduction is possible. Therefore, the cosine transform (in this context often called the cepstrum transform) (Rabiner & Schafer, 1978), is applied to the output vector, i.e., the cepstrum coefficients \( c_r \) are computed from the filterbank outputs \( o_j \) using

\[ c_r = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} o_j \cos \left( \frac{\pi}{N} (j - 0.5) \right) \]

where \( N \) is the number of filters. This second step is important particularly when the coefficients are used for standard HMM phonetic output probability estimation, because the cepstrum coefficients are closer to being statistically orthogonal than the filterbank outputs. However, the data reduction is also useful in the case of ANN probability estimation to reduce the problems with overfitting to training data (see the section: “Phonetic classification”). Currently, the first twelve cepstrum coefficients are used and together with the energy of the frame they constitute the compressed feature representation.

The goal of the feature extraction is not simply to reduce the amount of redundant information but more importantly to transform the information to a representation that optimises the performance of subsequent decoding. Therefore, as the next step in the recognition chain has a relatively weak modeling of the short-time dynamics, the first and second time-derivatives of the transformed spectra are also computed. In the HMM case, the inclusion of derivatives is more important than when using ANNs. The reason is that recurrent time-delay ANNs can compute dynamic features internally, Time-delay ANNs have a window of frames that are used in the computation of phoneme probabilities. Still, the derivatives are a more compact representation of the speech signal in the neighbourhood of a frame than a window of surrounding frames, and can therefore help to reduce problems with overfitting the training data. The standard method for computing the derivatives, \( d_t \), is

\[ d_t = \frac{1}{10} \left( 2c_{t+2} + c_{t+1} - c_{t-1} - 2c_{t-2} \right) \]

derived from linear regression. In the WAXHOLM system we have also implemented an alternative estimate,

\[ d_t = \frac{1}{12} \left( c_{t+2} + 4c_{t+1} - 4c_{t-1} - c_{t-2} \right) \]
derived by applying Richardson extrapolation to the simple central difference approximation $0.5(c_{t+1} - c_{t})$ (Dahlquist et al., 1974). In both formulae, $d_t$ is the estimate of the time-derivative at time $t$ of the coefficient $c$. The first formula is used by default in the HTK tool-kit (Young et al., 1995) and is derived from linear regression analysis, i.e., the inclusion of points further from $t$ ($c_{t+2}$ and $c_{t+2}$) serve the purpose of reducing local errors in the measurement of $c$. In contrast, in the second formula those points serve to reduce the truncation error in a series approximating $d_t$.

Our intuition is that the second formula should perform better because it gives higher weight to frames close to $t=0$. However, in preliminary experiments, the recognition performance was not significantly different for the two estimates and for compatibility with HTK toolkit, the first estimate is currently used.

The sequence of cepstrum coefficients, energies and their first and second derivatives for each frame is the stream passed to the phonetic classification block. Just as the case with the signal processing block, the feature extraction block can reproduce the features computed by the HTK toolkit by using the Mel scale, derivatives derived by linear regression and choosing all parameters appropriately.

Phonetic classification

The third block in Figure 2 is the mapping from acoustic observations to phonetic classes. We have chosen the standard continuous density HMM paradigm, i.e., the models estimate the probability density functions (pdf) for the observed energy, cepstrum and derivative features given the hypothesized phonetic class for each frame (Lee, 1989). For a feature vector $o$, and a class $c_i$, the probability to estimate is: $p(o|c_i)$. Again following the standard HMM paradigm, the temporal patterns are modelled by a probabilistic finite state automaton (FSA) (Fig. 9). Two different modes for the probability estimation are implemented and a third mode for a more general feature representation also exist. The modeling of the dynamic patterns also varies in the three modes.

Artificial neural network mode

ANN based phonetic classification has continuously been developed at the department (Elenius & Takács, 1990; Elenius & Blomberg, 1992; Elenius & Trovén, 1993; Ström, 1992, 1994b, 1995b, 1996). In the ANN mode, probabilities for phonemes are estimated using a feed-forward neural network, trained with the back-propagation algorithm. It has been shown (Gish, 1990) that ANNs trained with back-propagation and an error function of a particular class, can estimate the probability $p(c|o)$ given that the functional complexity of the network is sufficient. We use the cross entropy error measure and large networks with high functional complexity, trained in a truncated training scheme to prevent over-learning of training data. A more extensive coverage of this training scheme can be found in Bourlard & Morgan (1993). To evaluate the probability estimates, the estimated probabilities can be plotted against the real probabilities in a histogram as shown in Figure 6. We see that the estimates on the training data are very similar to the real probabilities, indicating that the conditions of the result of Gish (1990) are fulfilled.

The ANN has both a time-delay window, first suggested by (Waibel et al., 1987) and recurrent time-delayed connections (Robinson & Fallside, 1991) in the hidden layer (Ström, 1992, 1994b, 1995b, 1996). This unconstrained network topology allows for experiments with many different time-delay windows and types of recurrence. Initial experiments has resulted in a topology and dynamic connection scheme that
allows for fast training with standard UNIX workstations, i.e., no special hardware. The topology of a typical ANN is shown in Figure 7. The node activations and the error gradient are computed in the same way as the classic ANNs of (Rumelhart et al., 1986), with the exception that the sigmoid function is replaced by the computationally more convenient tanh function. It is easy to show that besides the change to an activation function with a symmetric range (-1;1) this is equivalent to a linear transformation of the weight space:

\[
\tanh\left(\frac{x}{2}\right) + 1 = \frac{1}{2} \left( \frac{e^\frac{x}{2} - e^{-\frac{x}{2}}}{e^\frac{x}{2} + e^{-\frac{x}{2}}} + 1 \right) = \frac{e^\frac{x}{2} - 1}{e^\frac{x}{2} + 1} = \text{sigmoid}(x)
\]

Thus, the activation \( a_j \) of unit \( j \) is defined by:

\[
a_j = \tanh(\text{net}_j)
\]

where

\[
\text{net}_j = \sum_i w_{ij} a_i
\]

and \( w_{ij} \) is the connection weight from unit \( i \) to unit \( j \). The gradient can now be written:

\[
\delta_j = -\frac{\partial E}{\partial \text{net}_j} = (1-a_j)(1+a_j)\sum_i (\delta_i w_{ji} - e_j)
\]

\[
\frac{\partial E}{\partial w_{ij}} = -\delta_j a_i
\]

where \( e_j \), the error induced directly from the deviation from the target values, is defined by:

\[
e_j = \begin{cases} 
0 & \text{when the unit is not an output unit} \\
2 & \text{when the units’ phoneme is the correct phoneme} \\
\frac{2}{a_j + 1} & \text{otherwise}
\end{cases}
\]

Here we have modified the cross entropy error to fit the new activation range. Note that if an output unit has no outflowing connections, then \( e_j \) cancels one of the factors of \( \delta_j \) and the resulting \( \delta_j \) is simply the difference between \( a_j \) and the target (1 for the correct phoneme and -1 otherwise).

To handle the recurrent connections, the back-propagation algorithm is extended by propagating errors not only in the spatial dimension, but also backwards in time (back-propagation through time, (Pearlmutter, 1990). Learning is speeded up by restarting the propagation-through-time every 20th-30th frame (200-300 ms) and updating the weights. This procedure is also used by Robinson (1994). Weights are updated according to standard back-propagation with momentum term:

\[
\Delta w_{ij}(t) = \gamma \Delta w_{ij}(t-1) + \eta \frac{\partial E}{\partial w_{ij}},
\]

where \( t \) is the update-iteration number and \( \gamma \) and \( \eta \) are learning-rate parameters. The error \( E \) in the last equation is the accumulated error for all frames after the last weight update.

The probabilities \( p(c_i|o) \) are converted to the desired estimates \( p(o|c_i) \) using Bayes’s rule,

\[
p(o|c_i) = \frac{p(c_i|o) p(o)}{p(c_i)} \approx \frac{a_c + 1}{2} \frac{p(o)}{p(c_i)},
\]

where \( p(c_i) \) are the \( a \ priori \) class frequencies that can be estimated off-line from the training data and \( a_c \) is the activation of the output unit for phoneme \( i \). \( p(o) \) is constant for all classes and is therefore dropped in the computations.

In the ANN mode, a simple minimum-duration model is used. This is implemented by using an FSA with \( m \) nodes and a self-loop on the last node only, as shown in Figure 9. The minimum duration, \( m \), for each phoneme is selected such that about 5% of the phones in the training data are shorter than \( m \). The fraction 5% was chosen, after some experimenting, to optimise the recognition performance.

**Hidden Markov model mode**

The HMM mode is an implementation of a standard continuous density HMM recogniser. Currently this module is dependent on the HTK toolkit for training the parameters (Sjölander, 1995). Multiple mixture Gaussian pdf’s and the transition matrices are trained using the toolkit and the parameters are then transferred to our system for real time decoding. We refer to the HTK documentation (Young et al., 1995) or a text book on HMM (Rabiner & Juang, 1993; Huang et al., 1990; Waibel & Lee, 1990) for the training of the model parameters. Fig. 9 shows
how the HTK transition matrix is mapped to our FSA where the observation pdf’s are associated with arcs instead of nodes.

The computation of the pdf’s during the real-time decoding deserves some attention. During the search, many different hypotheses evaluate the same phoneme, but for different words. Also, because of parameter tying, different triphones can have the same pdf or share some of the Gaussians of their mixtures. To take advantage of this sharing, the computations are arranged so that each mixture and each Gaussian is computed at most once for each frame. This is done by saving the frame number of the last computed value for the mixtures and Gaussians. When the value is required, it is first checked if its frame number is the current frame, in which case the value can simply be looked up, otherwise it must be computed. The fraction of all Gaussians that is computed depends on the beam-width of the Viterbi beam search (see the Decoding section) and on the amount of parameter tying in the acoustic models, but typically the pdf computation is speeded up by almost one order of magnitude.

**General phone-graph mode**

Although the partitioning into short frames of about 10 ms is the dominating method today, other approaches have certainly been studied and implemented in ASR systems (Zue et al., 1989; Stevens, 1995; Lewin, 1996). To support development and evaluation of such approaches, a third mode for phonetic input exists. Instead of a linear stream of short frames, the input can be described by a phone-graph (Fig. 8) where nodes represent points in time and arcs are acoustic segments between them. All arcs have a vector of probability estimates \( p(o|c_i) \) for each of the phonemes. In the phone graph mode, the FSA is particularly simple; a phone is modelled by two nodes and an arc connecting them (Fig. 9).

**Figure 8. Illustration of a phone graph.** The arcs of the graph represents phone segments and also carries information about the probabilities for each phoneme on each arc. The task for higher levels of the system is to find a probable path through the graph that also meets the constraints from the pronunciation lexicon and the grammar.
The phone graph mode of the WAXHOLM system is a powerful tool for evaluating new methods for feature extraction and has been used with some success in a study of phone graphs computed in an external phonetic analysis procedure (Lewin, 1996) based on dendrograms (Glass & Zue, 1988).

**Note on the Markov assumption**

By using HMMs for the dynamic modelling, as is done in the HMM mode by definition, and also in the hybrid ANN/HMM mode, we make the Markov assumption about the output probability distributions, i.e., that the output probabilities for different frames are independent random variables. This is a very coarse approximation (perhaps the most severe source of error at the acoustic level in contemporary main-stream ASR). It is easy to see why the phonetic output probabilities of different frames are correlated. Consider for instance two consecutive frames in the steady state of a vowel. The phonetic identities of the two frames are clearly highly correlated – if the first frame is part of an /a/, then so is the second frame with high probability and vice versa. The extraction scheme itself also adds to the correlation. In the signal processing block, the FFT is computed on overlapping Hamming windows, i.e., the same data-points are used for consecutive frames. Further, in the feature extraction block, the first and second time-derivative are added to the representation, increasing the correlation between close frames.

The problem arises also in the phone-graph mode but is less severe. The phonetic output probabilities of segments are less correlated than those of frames, because segments are defined to be acoustically different. But the phonetic output probabilities are nevertheless correlated due to, for example, the fact that the same speaker produces all segments of an utterance.

**Lexicon and grammar**

The lexicon and grammar provide the top-down constraints that are combined with the acoustic representation of utterances in the decoding (Fig. 3). The lexicon includes pronunciations in terms of phoneme sequences for the words. To enhance the phonetic modelling of frequent words, alternative pronunciations are also included. Each word is also tagged with a word-class, used by the grammar. We have chosen a class bigram grammar, i.e., the probability estimates are smoothed over the word-class of the previous word.

**The WAXHOLM pronunciation lexicon**

The lexicon of the WAXHOLM project is of medium size and task dependent (Bertenstam, 1994a,b). The total number of words is around 850 and about 260 of those are ports for the boat traffic in the Stockholm archipelago. Each word entry in the lexicon has one or more pronun-

---

### Figure 9.

The three different acoustic scoring modes are reflected by different dynamic models of phonemes. Left: in the ANN mode, the dynamic modelling is constrained to a minimum condition on the phoneme duration. Middle: in the HMM mode, the standard transition matrix trained by the HTK toolkit is used but the model is transformed to a representation where the observation probabilities are on the arcs. Right: the phone graph mode has a particularly simple dynamic FSA model.
Ström: Continuous speech recognition in the WAXHOLM dialogue system

The lexical stress information in the lexicon is ignored by the ASR module (but see Höberg & Sjölander, 1996) and the phoneme inventory is reduced to 45 classes. Many words have multiple pronunciations in the lexicon resulting in a total number of pronunciations close to 1000. For example, the Swedish word, “skärgården” (the archipelago) has three entries:

\[ \text{färgo:den} \]
\[ \text{färgo:n} \]
\[ \text{färgon} \]

The word-classes are defined in the lexicon by tagging each word with its class. The classes are based on both semantic and grammatical features and general features are mixed with task dependent ones. For example, the class PORT is a semantically based class, suitable for the particular task whereas VERB is based on a grammatical feature. The small class GO is again a semantically based class that contains only the word “äka” (go, travel), which in another domain could have been part of the VERB class but deserves a class of its own in this domain because of its high frequency and relevance. The generic versus domain-specific aspects of the system is discussed more thoroughly in (Carlson & Hunnicut, 1996).

A special class of “non-linguistic” words is also defined in the lexicon. It includes acoustic events such as breath pauses, coughs and laughs. The “non-linguistic” words are composed by special purpose “phonemes” trained on the database.

The class bigram grammar

A class bigram grammar is used in the ASR module of the WAXHOLM system. This is a probabilistic grammar where the probability for word \( w_i \) following word \( w_j \) is estimated by smoothing over the word-class of \( w_i \), i.e., \( p(w_i \mid \text{previous word is } w_j) \) is approximated by \( p(w_i \mid \text{previous word is in } c_j) \), where \( c_j \) is the word-class of \( w_j \). The smoothing gives robust estimates when the training corpus is of a limited size. For convenience, from here we use the notation: \( p(w_i \mid c_j) := p(w_i \mid \text{previous word is in } c_j) \).

The probabilities \( p(w_i \mid c_j) \) are factored into \( p(c_i \mid c_j) p(w_i \mid p(c_i)) \), where \( p(c_i \mid c_j) \) is the class bigram and \( p(w_i \mid p(c_i)) \) is the within-class word frequency. This factoring makes the use of the grammar in the decoding computationally more efficient, as described in the following sections.

The perplexity of the class bigram grammar on sentences of the WAXHOLM database (Bertenstam et al., 1994a,b) is 28.

The lexical graph

The lexical, grammatical and phonetic constraints are merged into a large probabilistic FSA – the lexical graph. The phonetic output probability distributions, phonetic symbols as well as transition probabilities are associated with arcs in the graph. The nodes are marked to guide the decoding, e.g., nodes can be marked: "word-start", "word end" or "grammar node".

First, for each pronunciation of each word of the lexicon, phoneme-models are joined to form the strings of phonemes (Fig. 10). At this stage, all word-start and word-end nodes are marked. Then one node is added for each word-class and arcs from all word-class nodes to all word begin nodes are created. Finally, arcs are created from the word-end node of each word to its word-class node (Fig. 11). The transition probabilities from word-class nodes to word begin nodes are the class bigram probabilities \( p(c_i \mid c_j) p(w_i) / p(c_i) \) and the transition probabilities from word-end nodes are set to unity. It is easy to see in Figure 11 that this lexical graph generates word strings in agreement with the constraints of the lexicon, the grammar and the phonetic models.

![Figure 11](image-url)
To be able to control the relative influence of the different knowledge-sources, two heuristic parameters are introduced that alters the transition probabilities from word-class nodes to word-begin nodes. Therefore, the logarithmic transition probability added for a new word is: $\alpha \log(p(c_i | c_j)p(w_i)/p(c_i)) + \beta$, where $\alpha$ controls the influence of the grammar and $\beta$ is the “word-penalty” that controls the number of words in the output word strings and can be used to balance the number of errors due to insertion and deletion (see the section “Evaluation”).

**Estimating the word frequencies**

In our experience, it is often the case that the within-class word frequencies, $p(w_i)/p(c_i)$, of the bigram grammar are very small and therefore difficult to estimate robustly. In the WAXHOLM system, we use a frequency estimate based on properties of human language to solve this problem. The extreme case that words in the lexicon does not occur in the training data even once is not uncommon. This leads to the conclusion that an *a priori* distribution for the word frequencies is necessary. Zipf’s law states that the logarithm of the word frequency is approximately proportional to the logarithm of the rank of the word (Zipf, 1949; Pierce, 1961; Li, 1992), where rank is defined such that the Nth most frequent word has rank N. We get:

$$\log(p) \propto \log(R) \Rightarrow R = p^\rho$$

for some constant $\rho$, where $R$ is the rank and $p$ is the word frequency. If the word class is large then the difference in word frequency is small between two words whose rank differs by only one. Therefore the relation between rank and word frequency can be approximated by a continuous function. Noting that the word frequency density is the differential of the rank and

$$dR = d(p^\rho) = \rho p^{\rho-1} dp,$$

we see that the density function of the word

![Figure 11. The lexical graph. This graph is a unified description of the phonetic, lexical and grammar constraints of the system. Note that, as indicated by the dotted line, the set of word-class nodes on the left and right are the same nodes.](image-url)
frequency is:

\[ f_p(x) = \rho p^{\rho-1} \]

To determine the unknown constant \( \rho \) we use the constraint that the sum of the probabilities of all words in the class equals unity, i.e,

\[ N_w E(p) = 1 - \xi_{OV} \]

where \( N_w \) is the number of words in the class included in the lexicon and \( \xi_{OV} \) is the out-of-vocabulary rate, i.e, the rate of words in the class not in the lexicon (in many cases it is reasonable to let \( \xi_{OV} = 0 \)). Expanding \( E(p) \), the expected value of \( p \), we get the desired equation for \( \rho \):

\[
N_w E(p) = N_w \int_0^1 \rho x^{\rho-1} x \, dx = N_w \frac{\rho}{\rho+1} = 1 - \xi_{OV} \Rightarrow \rho = \frac{1 - \xi_{OV}}{N_w + \xi_{OV} - 1}
\]

and thus

\[
f_p(x) = \frac{1 - \xi_{OV}}{N_w + \xi_{OV} - 1} x^{\frac{2(1 - \xi_{OV}) - N_w}{N_w + \xi_{OV} - 1}} \]

Now, assuming there is a statistical sample of \( N \) words where a particular word occurs \( n \) times, the a posteriori distribution is proportional to the a priori distribution times a binomial distribution, and we get a word frequency estimate by minimising the mean square error:

\[
\hat{p} = \arg \min_{p} \int_0^1 f_p(x) x^n (1 - x)^{N-n} (x - p)^2 \, dx
\]

\[
\Rightarrow \int_0^1 x^{\frac{2(1 - \xi_{OV}) - N_w}{N_w + \xi_{OV} - 1}} x^n (1 - x)^{N-n} \, dx = \int_0^1 x^{\frac{2(1 - \xi_{OV}) - N_w}{N_w + \xi_{OV} - 1}} x^n (1 - x)^{N-n} \, dx
\]

where \( \Gamma(x+1) \) is Euler’s standard generalisation of the factorial function to the real numbers.

Zipf’s law is not always a good approximation. In particular, it is worse for frequent words, but this is not a problem in practice because there are many occurrences of those words, i.e, \( \hat{p} = n/N \). Also when the word-class is small, Zipf’s law does not hold. However, because we normalised \( f \), so that the sum of the probabilities of the class equals unity, the estimates are still useful. For instance, if \( N_w = 2 \) and \( \xi_{OV} = 0 \) we get:

\[
\hat{p} = \frac{n+1}{N+2}
\]

This is the same expression as we would get if we assumed a uniform a priori distribution (\( f = 1 \)).

The Viterbi search pass

The last block in the ASR chain of Figure 2 is the search for a word sequence. The search is constrained by the lexicon and the probabilistic grammar of the lexical graph. The objective is to find the word sequence that gives the highest output probability for the current acoustic input. However, it is practically impossible to compute probabilities for all word-strings without making approximations, because that is a task of exponential computational complexity. Instead, it is common practice to adopt the Viterbi
approximation. Many different paths in the lexical graph correspond to the same word string and therefore the output probability for a given word string is a sum over many paths. The Viterbi approximation is based on the assumption that one path has much higher probability than the sum of the others. Thus, the total sum is approximated by the probability of the most probable path, reducing the problem to one that can be solved by dynamic programming (Viterbi, 1967). In the following two subsections we show how this is done in practice.

The product graph

The decoding is a search in a space determined by two sources of knowledge – the lexical graph and the acoustic input. In this sub-section it is shown how these two sources can be merged to one unified object – a product graph. The concept of product graphs was introduced in (Ström, 1994a). First we note that the acoustical input can be represented in the form of an FSA. One of the three supported representations of the acoustic input, the phonetic graph, already is an FSA and the other two can easily be transformed to FSAs by identifying one arc with each frame and adding nodes between the frames as the input graph in Figure 12.

The product graph is constructed as follows: The nodes of the product graph are the Cartesian product of the nodes of the acoustic graph and the lexical graph, i.e., there is one node in the product graph for each pair of one node in the acoustic graph and one node in the lexical graph (Fig. 12). Further, there is an arc in the product graph between two nodes if and only if there are arcs both in the lexical graph and in the acoustic graph between the respective nodes. The output probability densities are inherited from the lexical arcs and the transition probabilities are the product of the transition probabilities on both lexical and acoustic arcs.

With some thought, it is realised that if and only if a word-sequence and its corresponding sequence of acoustic frames satisfy both the constraints of the lexical graph and the acoustic graph, they satisfy also the constraints of the product graph. The output probability of a path in the product graph is the product of the output probabilities of the two graphs. Thus, instead of

![Figure 12. The product graph. The product graph combines the constraints of the lexical graph and the acoustic input graph. The nodes of the product graph are the Cartesian product of the nodes in the lexical and input graphs. There is an arc in the product graph if and only if there are arcs in both the input graph and in the lexical graph between the corresponding nodes. Two of the arcs in the lexical graph are indicated in the figure and all corresponding arcs in the product graph are shown. See the main text for details.](image-url)
performing a simultaneous search in two graphs, it is sufficient to find the most probable path through the product graph – a classic dynamic programming problem.

The forward Viterbi search
It is easy to see that an acyclic acoustic graph implies an acyclic product graph, i.e. there is no set of arcs that define a loop. In particular, when the nodes are arranged as in Figure 12, it is seen that all arcs flows from left to right and therefore no loop is possible. This suggests the dynamic programming procedure in Figure 13 to find the most probable path through the product graph.

The path with the highest probability can be found by back-tracking in the back-pointer array from $B_{ti}$ and backwards, where $T$ is the last frame and $f$ is the end-node of the lexical graph (if there are more than one legal end-node in the lexical graph, the end-node with the highest $H^*_v$ is chosen). During the back-tracking, the pivot arcs of the words of the word sequence with the highest probability will be traversed and can be recorded (see the section on pivot arcs below).

Note that the product graph is never explicitly constructed. It is merely a conceptual tool and is too large for allocating memory for in the computer implementation. Another implementation detail is that log probabilities are used to avoid floating point overflow. Note also that the algorithm is formulated for the frame-based acoustic graphs of the HMM mode or the ANN mode. The extension to allow for general phonetic graphs is straightforward: simply add an inner loop over all arcs of the acoustical graph pointing to the current acoustical node $t$.

When the Viterbi search is used solely as a means to compute the heuristic, $H^*$, no back-tracking is necessary and the array $B$ is not maintained. The word-sequences are then obtained in the subsequent $A^*$ search pass (see the section “The $A^*$ stack-decoding search” below).

Beam pruning
The Viterbi algorithm, as formulated in the previous section, uses both CPU time and memory in proportion to the size of the product graph, i.e. the size of the lexical graph times the size of the acoustical graph. This is unsatisfactory from a computational point of view in all but special cases. The standard solution is beam-pruning (Loverre & Reddy, 1980). The idea is to process only the most promising lexical nodes $i$ at each frame. Common terminology is that those nodes are alive at the particular frame and that they are inside the beam. Thus, after processing a frame in the Viterbi algorithm, the scores for all nodes are inspected and only those that pass some evaluation criterion are considered in the computation of the scores for the next frame.

The CPU time is proportional to the mean width of the beam. However, some book-keeping is necessary to keep track on which nodes are alive, which adds to the CPU time. The memory used by the Viterbi search is dominated by the score matrix $H^*$ and (if present) the back-pointers $B$. With beam-pruning, the sizes of $H^*$ and $B$ can be reduced to being proportional to the number of frames times the mean beam-width. This is accomplished by so called “skyline storage”, i.e. lists of: node-index, score, and back-pointer for only the alive nodes at each frame are stored (Fig. 14). To be able to find the score of a particular node efficiently, the lists are sorted by node index (node look-up can then be performed by binary search).

The classical criterion for determining if a node is inside the beam is to specify a probability threshold relative to the probability of the currently top scoring node. A variation is to limit the number of alive nodes at each frame and an obvious generalisation is to use both a relative probability criterion and an absolute bound on the number alive nodes. We have found that pruning based on absolute number of nodes can give a better trade-off between recognition accuracy and CPU-time, in particular when the beam is relatively narrow. Table 1 shows the recognition performance for various combinations of thresholds and Figure 15 shows the beam width for an utterance with varying thresholds. An important advantage with limiting the number of alive nodes, in particular in real-time applications, is that a bound on the maximum CPU time used for each frame can be determined in advance. Another advantage is that the mean beam-width can be quite low without risking that all nodes are pruned at some frame – a problem that often occurs when only the classical criterion is utilised.

The problem of selecting the top $N$ nodes is of linear complexity and does not contribute more than marginally to the CPU time of the Viterbi search. Steinbiss et al. (1994) proposed an approximating algorithm, “histogram pruning”, to solve the problem. Here we give the exact, “divide-and-conquer” algorithm with
Initialise the score matrix:
let $H^*_{ti} = 0$ for all $t$ and $i$

for all frames $t$ do
    for all lexical nodes $i$ do
        for all lexical arcs $a$ flowing from node $i$ do
            let $j =$ the index of the node who arc $a$ points to

            Compute the probability cost of the current arcs:
            let $c = p_a p(o_t | \text{symbol}(a))$
            where $p_a$ is the transition probability of the lexical arc and $p(o_t | \text{symbol}(a))$ is the output probability given the phonetic symbol of the lexical arc at frame $t$.

            Compute the probability cost from the beginning of the utterance to this pair of one lexical node and one acoustical node:
            let $h = c H^*_{(t-1)i}$

            Update the score matrix:
            if $h > H^*_{tj}$ then let $H^*_{tj} = h$
            If needed, update the back-pointer array:
            if $h > H^*_{tj}$ then let $B_{tj} = a$

        end
    end
end

Figure 13. Pseudo code for the Viterbi search.

Table 1. Effect of the two different pruning criteria. The top table shows the word recognition accuracy in percent for different combinations and the bottom table shows the corresponding run times (the times are not normalised, thus only relative comparisons are relevant). Depending on the desired time/performance trade-off, different combinations are optimal, but it seems clear that combining the two criteria is often fruitful. If the degradation of the word recognition accuracy from 75.9 to 75.3 is considered acceptable, the shaded entry of the table is a good choice. Lowering either of the two thresholds results in a large reduction in accuracy.

![Skyline storage](image)
the same computational complexity, used in the WAXHOLM system:

1. Let the desired number of alive nodes be $N$.
2. While iterating over lexical nodes, record the probability of the node with highest probability ($=p_{\text{high}}$) and the lowest probability ($=p_{\text{low}}$) at the current frame.
3. Let $x = (p_{\text{high}} + p_{\text{low}})/2$, let $i = 0$ and let $G_0$ be the set of all nodes.
4. Split the nodes of $G_i$ into two sets: $g_+$, those with probability $> x$ and $g_-$, those with probability $< x$.
5. If the total number of nodes with probability $> x$ is equal to $N$ then quit.
6. If the total number of nodes with probability $> x$ is less than $N$, then let $p_{\text{high}} = x$, $x = (x + p_{\text{low}})/2$ and $G_{i+1} = g_+$, otherwise let $p_{\text{low}} = x$, $x = (p_{\text{high}} + x)/2$ and $G_{i+1} = g_-$. 
7. Let $i = i + 1$ and go to step 4.

In step 4 of the algorithm, all nodes must be considered in the first iteration, but in subsequent iterations only one of the divisions from the last iteration needs to be split. Because step 4 itself has linear complexity with respect to the number of nodes to split, and the number of nodes to split decreases approximately by half in each iteration, the whole algorithm has linear computational complexity.

**Note on the Viterbi approximation**

A theoretical problem of the Viterbi approximation is that the assumption that one path dominates the probability is usually not fulfilled. Thus, the estimates of word-string probabilities cannot be expected to be correct. However, from experience we know that the estimates can be used for relative comparison to find the most probable word-string. If the estimates are utilized for any other purpose, empirical evidence is necessary to justify that particular use.

The Viterbi approximation also explains why the probabilities on the arcs from a grammar node does not sum to unity. Consider a word with $N$ different pronunciations with equal frequency. To make the sum of probabilities from its grammar node equal unity, we would need to multiply the word-frequency by $N^{-1}$. However, because only the most prominent path is used in the Viterbi search, this would lead to a disadvantage for this word in the competition with words with only one pronunciation in the lexicon.
The A* stack-decoding search

The Viterbi search gives the most probable path through the lexical graph and the corresponding word string. In some applications, this is all that is required from the ASR module, but in many cases it is desired that the ASR module outputs multiple hypotheses of probable word strings. In the WAXHOLM system for instance, the NL-parser requests an N-best list of hypotheses to re-score and re-sort based on a robust context free grammar (Carlson, 1996; Zue et al., 1991) conditioned by the current dialogue state. Re-sorting N-best lists have also been used within the WAXHOLM project in experiments with speech production oriented ASR (Blomberg & Elenius, 1996).

The A* stack-decoding search can produce sets of multiple hypotheses in several different formats as is discussed in the following subsections. This second search pass can be thought of as a generalized back-tracking. In fact, if only the top word string is required, the A* search can be used with a high pruning threshold yielding a very fast search. Of course, the search is never quite as fast as simple back-tracking, but a computational advantage is that the back-pointers, \( B \), are not needed, reducing the memory requirement of the Viterbi algorithm to about half.

Stack decoding

The Viterbi algorithm has polynomial (linear) computational complexity for both CPU-time and memory. Searching for multiple paths is a harder and in general an exponential problem. This forces us to choose a best-first strategy. Stack decoding is a scheme for bookkeeping the different hypotheses to be evaluated during a best-first search (Nilson, 1971).

In our case, the hypotheses are paths in the product graph. Starting from the possible end-nodes of the graph, a search tree is constructed where the partial path with the current best probability is always expanded first. To keep track of the current best path in the search tree, a priority queue (the stack) is used. The stack can be efficiently implemented using the heap data structure that requires only logarithmic time both for insertions and for retrieval of the best path (Sedgewick, 1988). Basic stack decoding is illustrated in Figure 16.

A problem with stack decoding in this form is that all extensions lower the observation probability of the path. Consequently, short paths will be expanded much more often than long paths, making the search resemble a breadth-first search that is computationally too inefficient. Thus, a more sophisticated strategy to choose which path to expand is necessary.

The A* heuristic

To be able to compare partial paths of different lengths it is not enough to know the observation probability of the partial paths themselves. A fair comparison also includes some estimation of the observation probability of the continuation of the paths. The success of an A* algorithm depends on the choice of this estimate called the A* heuristic (Nilson, 1971). Here it will be shown how the probability score matrix \( H^* \) computed in the Viterbi pass can be used in a particularly well behaved such heuristic.

Consider a partial path from an end-node to some interior node \( s_i \) of the lexical graph at frame number \( t \) (Fig. 17). Let the observation probability of this partial path be \( g \). From the other side, the probability of the best path from a start-node to \( s_i \) is \( h^*_{s_i} \), where \( h^* \), is column \( t \) of the matrix \( H^* \), computed in the Viterbi search described above. Now we define an A* heuristic, \( f^* = g + h^*_{s_i} \). This very useful heuristic was first used in ASR by (Soong & Huang, 1991) and it has the property that it is greater than or equal to any other complete path containing the partial path. This is called A*-admissibility and it will be clear in the following why it is an important property.

Word extension

Extension from a node of the search-tree is performed as a Viterbi search backwards from the appropriate word class node of the lexical graph. In this embedded Viterbi search, whenever a word-start node is reached, a new branch is created in the search tree. No arcs connecting different words are expanded because the object is to search one word backwards only. The probability score of the partial path including the new arc is the sum of the score of the best path to the end-node and the best score in the embedded Viterbi search, i.e, the best score from the end of the utterance to the current node of the product graph (Fig. 16). The embedded Viterbi search is terminated when no lexical nodes are alive. At that point, the node of the search-tree with the highest A*-heuristic, \( f^* \), is selected for the next word extension.
A consequence of the A*-admissibility is that extensions that occur from the same product graph node will have descending $f^*$, i.e., the best path will be expanded first. Consider the special case of the utterance start-node of the product graph. When a search-tree node associated with this node is selected for extension, the corresponding “partial” path is actually a complete path and because paths are expanded in order of best output probability, the corresponding word string can be recorded and put in a list of hypotheses. However, this simple algorithm has the weakness that different time-alignments of the same word string at the same product-graph node will all be expanded individually, yielding a search with exponential computational complexity. Fortunately, this problem can be solved, and in the following sections we describe our solution.

### Pruning

In the Viterbi pass, pruning is relative to the locally best $h^*$ value at each frame. This means that it is possible for the globally most probable path to be pruned if it falls outside the beam at any frame. In contrast, during the A* search, pruning is relative to the probability of the globally best path, i.e., partial paths with an A* heuristic below some threshold are not extended. This implies that the top path is never lost, and pruning affects only the possible number of output hypotheses.

The mean beam-width of the embedded word-Viterbi searches is typically several orders of magnitude lower than that of the first Viterbi search because the A* heuristic yields such an efficient pruning. Therefore, in practice, the stack decoding phase is typically more than one order of magnitude faster than the Viterbi search.
The heap data-structure

The heap is a well-known data structure in the field of computer science. It is particularly suited for priority queues and it is remarkable that it is not used to a greater extent in ASR systems. Here we use it to implement the priority queue of the stack search (Fig. 16). The heap has the structure of a partially ordered binary tree. The ordering relation is that a parent has always higher priority than its children. A heap is balanced – all nodes at a particular depth are used before nodes at the next depth can be added. Therefore the heap can be stored in a vector and no pointers are necessary to indicate parent-child relations. Figure 19 shows schematically the heap structure. For stack decoding purposes, three operations on the heap are necessary: i) insertion, ii) popping of the node with highest priority and iii) improvement of the priority of a node already in the heap.

Following Sedgewick (1988), all three operations can be implemented with two subroutines, upheap and downheap. Upheap “bubbles” nodes with high priority, and downheap “sinks” nodes with low priority in a similar fashion to the popular bubblesort algorithm. Upheap and downheap can be implemented as recursive subroutines (Fig. 18). Note that when the heap is stored in a vector with start index 1, the parent of a node has index $i/2$, where $i$ is the index of the node. Further, the children of the node have index $2i$ and $2i+1$. The insert operation is performed by inserting the new node in the next free position of the heap (Fig. 19) and then calling upheap with the new node. Popping is implemented by removing the root node of the tree (index 1), moving the node in the last occupied position of the heap to the root and then calling downheap with the new root node. Finally, improvement of a node already on the heap, is performed simply by a call to upheap with the node.

All heap operations have logarithmic computational complexity with respect to the heap-size and contribute very little to the total runtime of the stack decoding search that is dominated by the embedded Viterbi searches.

Path merging and the word lattice

The word-lattice is an intermediate representation of sets of word string hypotheses. It is a directed, acyclic graph with a word-label on each arc and time-points on the nodes. The word-lattice is built during the stack search by associating a potential lattice-node with each node of the product graph (Hetherington et al., 1993). When an expansion from a node in the search tree occurs, for all new branches created in the tree, arcs are created in the lattice. The difference is that in the lattice, all nodes associated with the same product-eranb node are merged (Fig. 21). Because the objective is to

```
upheap(node n)
   if the parent of n has lower priority than n then
      Swap n with its parent
      upheap(n from its new position)
   else return
end

downheap(node n)
   if any child of n has higher priority than n then
      Swap n with its child with highest priority
      downheap(n from its new position)
   else return
end
```

Figure 18. Pseudo code for the two main subroutines for maintaining the heap.
build the lattice, the search-tree is not explicitly constructed in the implementation of the algorithm. \((H^*\) is tied to the nodes of the product graph, so all information necessary for the stack decoding is available anyway). This has the advantage that the computational complexity of the stack decoding search is brought down to linear because word extension occurs at most once from each product-graph node. In contrast, it is easy to see that constructing the search-tree is an exponential complexity problem with respect to utterance duration. Our approach differs from that of Hetherington et al. (1993) in that we construct the complete word lattice, containing all paths, including all segmentations that pass the pruning criterion. An analysis of the algorithm shows that the only extra computation needed is the actual construction of the arcs, the embedded Viterbi searches are of the same size. This method also has the advantage that no representation of the partial word strings is necessary at this stage of the search. In fact, that would lead to the same exponential complexity as explicitly building the search tree. However, in the section “N-best lists” we see that to compute an N-best list from the word lattice, it is necessary to keep a hash table of partial word strings.

At this point, we are prepared to formulate the complete A* decoding algorithm. Figure 20 is a

---

Figure 20. Pseudo code for the complete A* stack decoding search.

```plaintext
main-stack-decoding
  Insert the utterance-end node into the heap.
  while the heap is not empty do
    Pop the node \( n \) with the highest priority from the heap
    word-extension(\( n \))
  end

word-extension(\( n \))
  Perform an embedded Viterbi search backwards from the product-graph node associated with \( n \).
  Prune nodes with \( f^* \) below a pruning threshold.
  whenever a word-start node \( s \) is reached in the Viterbi search do
    if the lattice-node corresponding to the product-graph node of \( s \) does not yet exist then
      Create it
      Insert it on the heap
    else if the score of the current partial path is an improvement at this lattice node then
      Improve the node on the heap
    end
  end
  Create an arc for the word of \( s \) in the word-lattice
end
```

---

Searching in the word-lattice

The word-lattice is a rather dense graph as it contains all time alignments of the word sequences that meets the pruning conditions. In some cases this is desirable, e.g. when the lattice is used for acoustic re-scoring (Blomberg & Elenius, 1996). However, in most cases a symbolic representation where the alignment information is discarded is the desired output. For example, in a dialogue system such as the WAXHOLM demonstrator, the output from the ASR module is sent to the natural language component that does not utilise word alignment information. The post processing of the lattice that is performed for extracting the desired output can be seen as a third search pass, but the computational effort required by the post processing is negligible compared to the first two search passes.
Retrieving the most likely word string

The most fundamental output of the ASR system is the most likely word sequence. It can be retrieved by backtracking after the first Viterbi search or as the first complete path encountered in the A* search as discussed above. However, it can also be retrieved from the word-lattice.

After the A* search, score information has been computed for the nodes of the lattice, both forward to the end of the utterance and backward to the beginning. The score of the best path from the start of the utterance to the node is computed in the Viterbi search and the score of the best path from the node to the end is a by-product of the A* search. The total score of the most likely path is also known from the Viterbi search. Therefore, it is easy to determine if a particular node is part of the most likely path – simply check if the sum of the forward and backward scores equal the score of the most likely path. Similarly, it can easily be determined for each arc if it is part of the most likely word sequence, and consequently the word sequence itself can be retrieved. Note that it is not necessary to consider all arcs of the lattice in the search – the best path is found in a search from the start-node and only arcs from nodes of the best path are evaluated.

N-best lists

The output format currently used in the communication with the parser in the WAXHLOM demonstrator, is N-best lists, i.e, lists of the N most likely word sequences for each utterance. We have already suggested a route for constructing the N-best list in the A* search phase – whenever a partial path is selected for expansion from the utterance start-node, it is de facto a complete path and the corresponding word string is the next entry in the N-best list.

The algorithm outlined above needs to be refined to be practically useful. The most prominent problem is that when the search tree is built, a massive number of branches with identical word strings but slightly different time alignments will be created. A solution to this computational obstacle is to keep track of which word strings have been expanded to each product-graph node. Each word string is then allowed to be expanded at most once from each product-graph node. The A* admissibility condition assures that this is the path with the highest likelihood for the word string. In particular, word strings are selected for expansion at most once from the utterance start-node, i.e, put in the N-best list only once. The necessary bookkeeping of word strings is handled in a hash table.

A problem of creating the N-best list directly in the first A* search, is that the same word search needs to be performed many times from the same product graph node. But because in our case the N-best search is done after the word-lattice is built, word searches are simply look-ups in the lattice, instead of the computationally more costly embedded backward Viterbi searches.

Another problem that needs to be addressed is the “non-linguistic” words that are recognised by the ASR module but are not significant in the parsing, e.g, pauses, breaths, coughs, etc. Our
solution is to define word strings to be identical if they differ only by such words. Otherwise the N-best list would typically be composed of a large number of word sequences differing only by the number of inserted pauses.

**Returning word-graphs**

Although the word-lattice is a rather dense word-graph, word graphs are potentially more compact than N-best lists for representing large sets of word string hypotheses. The number of generated word sequences grows exponentially with the graph size, but obviously only linearly with the size of the N-best list. Graph density is used to compare sizes of word graphs (Ney & Aubert, 1994). The graph density is the total number of arcs in the graph divided by the number of words in the correct transcription of the utterance (Fig. 22).

The size of the lattice can be reduced extensively without altering the set of generated word-strings. In Ström (1995b) we proposed to use graph minimisation algorithms (Hopcroft & Ullman, 1979), well known in computer science, to reduce the lattice to a minimal deterministic graph generating exactly the same word sequences as the lattice. In an experiment with utterances from the WAXHOLM database, the size of the word-graph was on average reduced to 1.4% of the lattice, but the amount of reduction is dependent on the pruning threshold. The minimisation algorithm has exponential computational complexity with respect to lattice size for general graphs, but the experiments indicated that the special structure of word-lattices implies that the complexity is approximately linear with respect to utterance length and exponential only with respect to the pruning threshold. Thus, for reasonable pruning thresholds, the algorithm is feasible.

If the computational cost of the exact minimisation algorithm is still considered too high, approximate solutions are available. The so called word pair approximation (Ney & Aubert, 1994), states that the location of the boundary between two words, depends solely on the identity of the two words and not on words more distant from the boundary. By applying this condition, the lattice can be reduced considerably with only a small loss of generated word strings. An algorithm for applying the word-pair approximation to the lattice is:

```
for each node n
  for each word pair, wp, that can be generated from node n do
    Mark the arc pair with the highest likelihood that generates wp
```

Figure 22. Different representations of multiple hypotheses. In recognition experiments with the WAXHOLM database with different pruning thresholds, the average sizes of different representations was computed. To the right it is seen that the minimal deterministic word graph is about 50 times smaller than the original word lattice and the exponential growth of the N-best list size is clearly seen in the left figure. Word graph density is the number of arcs divided by the number of words in a correct transcription of the utterance.
end

Remove all unmarked arcs

Optimising the lexical graph

The lexical graph constructed above is one of many possible realisations of an FSA that generates the allowed phone-strings with the appropriate probabilities. To reduce the computational effort in the decoding, it is preferred to use an as small graph as possible. Graph minimisation algorithms exist that minimise any FSA to a minimal deterministic FSA (Ström, 1995b), but the generated minimal deterministic graphs are not suitable for our decoding algorithms. Therefore, a minimisation scheme that gives slightly larger graphs is pursued.

Creating the backward-trie graph

The first minimisation applied to the graph is to transform the set of sub-graphs for the words in each word-class to a graph with a tree structure, a so called trie (Fig. 23). It is easy to see that the trie generates the same phone-strings as the original graph. A difference is that, in the trie, when the current state is a word-end node, the information about the word identity is lost. However, the grammatical probabilities are on the arcs from the class-bigram nodes to the word-start nodes and are therefore unaltered. The following algorithm performs the minimisation:

1. Merge all word-end nodes in the word class and call the new node \( N \). 
2. For all nodes \( n_i \) that have an arc \( a_i \) to \( N \), merge those that have identical \( a_i \) and if they have a self-loop, it should also be identical. Exception: do not merge word-start nodes. 
3. For each new node created in step 2, call it \( N \) and go to step 2.

Arcs are identical if and only if:
- The phonetic symbols are identical
- The transition probabilities are identical
- The output probability distributions are identical.

By merging nodes it is meant:
- Keep all arcs from the nodes to merge and move them to the new node.
- Keep only one representative arc of all the (identical) arcs to the node from other nodes.
- Keep only one representative arc of all the (identical) self-loops (if any).

The trie-graph is kept and is used in the \( A^* \) phase of the decoding. In the Viterbi pass, however, the graph can be further minimised because the only significant output from the

Viterbi search is \( H^* \), the score of the best path to lexical nodes at all times, i.e., the word identities are not used. To be able to work with different graphs in the two phases, the concept of quotient graphs is introduced.

Quotient graphs

By defining an equivalence relation, \( \sim \), on the nodes of the lexical graph the very useful concept of quotient graphs can be introduced. Quotient graphs was used by (Kenny et al., 1991), and we follow their definition, but use a slightly different terminology.

Given a graph \( L \) and an equivalence relation \( \sim \) on the nodes of \( L \), i.e., the nodes are partitioned into a number of equivalence classes, the quotient graph \( L' \) can be defined as follows:
- Identify the nodes \( \{ n_i^* \} \) of \( L' \) with the equivalence classes of \( L \), i.e., nodes of the quotient graph, are associated with disjoint sets of nodes in the original graph.
- Let there be an arc in \( L' \) from node \( n_i^* \) to \( n_j^* \) if and only if there is an arc in \( L \) from \( n_i \) to \( n_j \), and \( n_i^* \sim n_i \) and \( n_j^* \sim n_j \). The phonetic symbol, the transition probability and the output probability distribution are copied to the arc of the quotient graph.

Note that for every path in \( L \), there is a corresponding path with the same sequence of phonetic symbols and the same probability. In general, \( L' \) overgenerates \( L \), i.e., \( L' \) generates sequences of phonetic symbols that \( L \) can not generate, but as we shall see in the next section, it is not difficult to define an equivalence relation so that \( L' \) generates exactly the same sequences as \( L \).

A quotient graph for the Viterbi pass

Having merged nodes from the end of words and backwards, it is tempting to do the same with nodes in the beginning of words. As will be seen,
this introduces several problems, but first we describe the algorithm used in the system.

With some modifications, the algorithm that created the trie-graph can be used for this minimisation too. The created, minimised graph will be a quotient graph to the trie-graph. We define a relation between the two graphs: each node of the trie-graph has a quotient node in the quotient graph.

The minimisation procedure starts by creating a copy of the trie graph (Fig. 24a). The new copy is called the quotient graph. At this point the two graphs are identical and the quotient nodes are simply the copies of the nodes in the quotient graph. During the minimisation, when nodes in the quotient graph are merged, the new node inherits the property of being quotient node from the merged nodes (Fig. 24). The modified algorithm is:

1. Merge all word-start nodes in the word class and call the new node $N$.
2. For all nodes $n_i$ that have an arc $a_i$ from $N$, merge those that have identical $a_i$ and if they have a self-loop, it should also be identical. Exception 1: do not merge word-end nodes. Exception 2: Do not merge nodes that have arcs to them from more than one node (that is nodes that was already merged in the first minimisation).
3. For each new node created in step 2, call it $N$ and go to step 2.

Figure 24. Rectangles represent arcs in the quotient graph and circles represent nodes in the backward trie graph. Dashed arcs are arcs in the trie and solid straight lines are arcs in the quotient graph. The symbols a-f on the arcs represent different phonemes. The quotient node of a trie node is the enclosing rectangle.

a) First, let the quotient graph be a copy of the trie graph.

b) Merge all word start nodes. At this point, nodes 1-5 of the trie has the same quotient node. This is step 1 of the algorithm.

c) Merge nodes 6 and 7 because they both have only an a-arc to the merged node. This is step 2 of the algorithm. Note that node 8 is not a candidate for merging because of Exception 2 of the algorithm.

d) This is the final quotient graph. The word pivot arcs are indicated by solid lines.
The modification of the merge operation is:

- Keep all arcs to the nodes to merge and move them to the new node.
- Keep only one representative arc of all the (identical) arcs from the node to other nodes.
- Keep only one representative arc of all the (identical) self-loops (if any).
- Let the new node be the quotient node to all nodes in the trie-graph that had any of the merged nodes as its quotient node.

Figure 24 shows an example of the construction of the quotient graph. It is easily seen that the quotient graph generates exactly the same phonetic sequences as the trie-graph. Furthermore, the two graphs have the important property that, in the Viterbi search, the score of the best path in the trie-graph from the beginning of the utterance to node \( n_x \) at time \( t \), is equivalent with the score of the corresponding best path in the quotient graph from the beginning of the utterance to node \( n'_x \), the quotient node, at time \( t \). Thus, the quotient graph can be used in the Viterbi search even if the trie-graph is used in the A* search pass. Figure 25 shows the two optimised graphs.

Note that the quotient graph can also be used independently if only the best path is requested. The word pivot arcs, defined in the next section, can be recorded while performing the simple back-tracking through the best path to get the most likely word string.

**Word pivot arcs**

One problem with the quotient graph arises: the information of word identity will be lost also when the current state is a word-start node. Thus, there will be no obvious arc to use for the within-class word frequency, \( p(w)/p(c) \). Our solution to this problem is based on the novel concept of pivot arcs. Intuitively, it seems clear that for each word there is some arc in the graph that is used only for this word and therefore \( p(w)/p(c) \) can be moved to this arc. In fact, Exception 2 of step 2 of the algorithm above guarantees that this is the

---

*Figure 25. The two reduced lexical graphs. Left: the trie graph used in the A* search. Right: the quotient graph used in the Viterbi search. Compare with Figure 11.*
case. We call these special arcs word-pivot arcs and a recursive algorithm to find a set of pivot arcs such that any path from a word-start node to a word-end node contains exactly one pivot arc is:

1. While performing the two minimisation algorithms in the previous sections, mark all arcs that are merged – they can not be pivots.
2. For each word-start node, let it be node $N$ and do step 3.
3. For all arcs $a_i$ from the node $N$ (except the self-loops) do step 4.
4. If $a_i$ is not marked as merged, it is a pivot-arc, else let the node which $a_i$ flows to be node $N$ and go to step 3 recursively.

The pivot arcs generated by the algorithm comprises a cutting set such that if all pivot arcs are removed, the graph is split into two unconnected graphs, both with a tree structure. Figure 24d shows the pivot arcs in a small graph. When the pivot arcs are identified, the within-class word frequencies are added to the transition probability of the respective pivot arcs. Of course, to preserve the quotient graph relation, the same operation must be performed also on the larger, trie-graph that is used in the A* phase.

Although pivot arcs were introduced here to solve the problem of assigning the word frequency score to an arc of the lexical network, pivot arcs have also another advantage. By recording the pivot arcs during back-tracking in a simpler search without the A* stack decoding phase, the most probable word sequence can be retrieved even if the small quotient network described here is used.

**Evaluation**

The evaluation of the output from the recognition engine is an essential part of a complete ASR development environment. Although the popular so called word accuracy measurement is important, there are many other important characteristics to consider. For instance, in the evaluation of a real time system such as the WAXHOLM dialogue system, it is necessary to also take the CPU time into consideration (Table 1). Intuitively, user satisfaction is the most relevant measurement of the performance of the complete dialogue system. However, this is difficult to formalise in precise terms and therefore, experiments with more well defined measures as task completion have been performed (Bertenstam, 1994b).

In all evaluations, the WAXHOLM database (Bertenstam et al., 1994a,b) has been partitioned into a training set of 1418 utterances and a test set of 327 utterances from ten different speakers that are not represented in the training set. Four of the test speakers are female whereas only thirteen of the fifty-six training speakers are female.

**Standard DP word matching**

The word accuracy measurement is based on aligning the correct word sequence with the output from the ASR engine. Three different types of errors are possible, *insertions* of words in the ASR output, deletions of words in the correct string and *substitutions*, where the correct word is aligned with an erroneous word. The evaluation is performed by defining an error function which is the sum of a penalty for each alignment error. The three different types of error have different penalties; typically a substitution has higher penalty than insertions and deletions. The alignment that gives the lowest error is found for each utterance through a dynamic programming (DP) search. The word accuracy is the number of errors divided by the number of words in the correct word strings.

We have chosen the penalties: substitution 10, deletion 7 and insertion 7. This is identical with the evaluation tools of the HTK toolkit. The NIST scoring software uses slightly different penalties, but the resulting difference in accuracy is very small. A problem with specifying only the word accuracy, is that the balance between insertions and deletions, controlled by the word-penalty parameter (discussed in the section: “The lexical graph”), influences the word accuracy heavily. Thus, reducing the number of insertions by raising the word-penalty parameter increases the number of deletions by a smaller number. Therefore the best accuracy is achieved when the insertions/deletions ratio is small. However, it is not likely that the optimal performance of the dialogue system is achieved at this operating point. In our evaluations, the word-penalty parameter is set to give approximately the same number of insertions and deletions – an intuitively more sensible ratio.

The “non-linguistic” words (pauses, breaths etc.) pose special problems in the evaluation. They are recognised by the ASR engine but not utilised by the parser, i.e., errors involving non-linguistic words are insignificant for the performance of the system. We have chosen to use a special alignment penalty for alignment errors involving non-linguistic words equal to one tenth of the normal penalty. Further, these errors are not counted in the statistics for word accuracy. This procedure gives almost identical results as if all non-linguistic words where simply removed.
from the word strings before the evaluation. The reason for the more complex approach is that it is easier extended to evaluation of word graphs that will be discussed in the next section.

The current best word accuracy achieved in the HMM mode is 86% using triphones and 15 Gaussians/mixture (Sjölander, 1996). In the ANN mode, 77% has been achieved using a network with the structure of Figure 7 and a hidden layer of 200 units (this will be reported more thoroughly in a future QPSR).

Evaluating multiple hypotheses

The word accuracy of the most likely word sequence gives no information about the quality of the remainder of the hypotheses in the case of N-best list or word-graph output. In the N-best case, a straightforward measurement is to record the best and worst word accuracy among the sequences of the list. For instance, in a study of the recogniser in the ANN mode, the word accuracy of the most likely sequence was 77% and the best and worst accuracy of entries in 10-best lists was 87% and 49%, respectively. This evaluation is of value to determine the potential gain that re-evaluation methods can yield (Blomberg & Elenius, 1996).

Evaluating the set of hypotheses of a word-graph is slightly more complicated. The solution that we have chosen is to perform a DP search to find the highest scoring word sequence generated by the graph. This search is analogous to the standard string alignment discussed in the previous section. The resulting “best-in-graph” word accuracy can not be interpreted in isolation – it must clearly be accompanied by some measurement of the size of the graph. The total number of word-strings generated by the graph is not a practical measure because it grows exponentially with the graph size. Instead, graph-density is useful (Ney & Aubert, 1994). The graph-density is the number of arcs divided by the number of words in the correct word-string. Figure 26 shows the average accuracy of the best word string covered by the word graph as a function of the graph density. Compare also with Figure 12 to see how large the corresponding N-best list and unreduced word lattice would be.

Summary

The ASR component of the WAXHOLM system

![Graph Density vs. Word Accuracy](image1.png)

![Graph Density vs. CPU Time](image2.png)

*Figure 26. Evaluation of sets of multiple hypotheses. Left: the word accuracy of the path that matches the correct transcription closest as a function of word graph density of the minimal deterministic word graph. Right: the computation time of the A* search relative to the Viterbi search as a function of graph density. The Viterbi pruning thresholds of the demonstrator are tuned to give a real time search. Thus, the times in the figure are the delay after the end of the utterance due to the stack search. The graph minimisation is also included in the times. This experiment was not performed with the currently best recognition system.*
has been described by focusing on four main blocks of the process. Our implementation of the first block, the signal processing, does not deviate much from other systems. In fact, by choosing appropriate values for a few parameters, the output of the HTK toolkit can be replicated. The main difference is that the implementation is much faster, mainly due to a fast FFT implementation. The situation is similar for the second block, the feature extraction, where the original feature of the WAXHOLM system is the optional variation of the derivative estimation.

It is in the phonetic classification block of the system that the advantages of our flexible modular approach becomes evident. It is easy to switch between three different modes for computing phoneme probabilities in this block. This allows for fast evaluation of new ideas and different technologies without modifying the entire system. In the hybrid ANN/HMM mode, we use an ANN topology with a dynamic connection scheme that is possible to train on standard UNIX workstations. We showed empirically that the ANN successfully approximates phoneme probabilities. The HMM mode of the block can be used to give results identical with the HTK toolkit, useful for comparisons with mainstream technology. The third mode is the general phone graph – a powerful tool for evaluating new ideas for phonetic classification.

Most of the report is devoted to the last block, dynamic decoding, where the words are retrieved from the phoneme probability estimates on the frame level. The lexicon and the class bigram grammar used by the ASR component are described. The probabilities of the grammar are estimated using a novel method based on Zipf’s law for word frequency to give robust estimates even when training data is very sparse. The merging of knowledge sources is controlled by merging graphs. All top-down knowledge is merged to a lexical graph and this graph is further merged with the acoustic evidence giving the product graph that is a representation of the search-space for the utterance. The search in the product graph is performed in two phases. The first phase is a standard Viterbi beam search with a slightly more elaborated pruning criterion than is common. We use both a bound on the span between the currently highest and lowest scoring hypothesis and a bound on the absolute number of alive nodes at each time. The combination of the two often gives a better trade-off between CPU time and recognition accuracy than any one can give alone. The second search phase is an A* stack decoding search that utilises the result of the first Viterbi search as search heuristic. Using the heuristic for pruning, an efficiently implemented priority queue, and avoiding to explicitly construct the search tree makes our implementation very fast and small in computer memory.

The optimisation of the lexical graph is perhaps the most original aspect of the decoding block of the system. The resulting graph that is used in the Viterbi search – the computationally most important part – is much smaller than the original lexical graph and in particular it has only one word-end and one word-start node for each word class. This results in a very small number of word connecting arcs (the square of the number of classes). Without the graph reduction, the algorithm would spend a large part of its CPU time searching over the word connecting arcs. When describing the reduction process we introduced the novel concept of word pivot arcs, an elegant solution to the problem that the word identities are ambiguous at both the word-start and word-end nodes.

Several different output formats are implemented in the system. Besides simply returning the most likely word string and N-best lists, the minimal deterministic word graph is a possible output. This graph contains the same word strings as the word lattice but exactly one alignment for each word string and the graph size is reduced by a factor of about 40. Various evaluation tools are used and again it is the evaluation of word graphs that is the most interesting. A characteristic that is easily computed from the deterministic graph is the size of the equivalent N-best list, but the most important measurement of the quality of the graph is the word accuracy of the path closest to the correct transcription, computed in a DP search. This measurement is given together with the graph density to give an indication of what can be gained by increasing the graph size.

Acknowledgements
The author wishes to acknowledge the researchers of the speech group at KTH involved in the WAXHOLM project, and in particular Rolf Carlson, Kjell Elenius and Jesper Högberg, for their valuable comments on an early manuscript of this report.

References


Peckham J (1993). A new generation of spoken dialog systems: results and lessons from the


