Internal Use Only

002

TR-IT-0272

Memory-efficient LVCSR search using a one-pass stack decoder

Mike Schuster

1998年9月3日

This report describes the details of a fast, memory-efficient one-pass stack decoder for efficient evaluation of the search space for large vocabulary continuous speech recognition. A modern, efficient search engine is not based on a single idea, but is a rather complex collection of separate algorithms and practical implementation details, which only in combination make the search efficient in time and memory requirements. Being the core of a speech recognition system, the software design phase for a new decoder is often crucial for its later performance and flexibility. This paper tries to emphasize this point – after defining the requirements for a modern decoder, it describes the details of an implementation that is based on a stack decoder framework. It is shown how it is possible to handle arbitrary order N-grams, how to generate N-best lists or lattices next to the first-best hypothesis at little computational overhead, how to handle efficiently cross-word acoustic models of any context order, how to efficiently constrain the search with word-graphs or word-pair grammars, and how to use a fast-match with delay to speed up the search, all in a single left-to-right search pass. The details of a disk-based representation of an N-gram language model are given, which make it possible to use LMs of arbitrary (file) size in only a few hundred kB of memory. *On-demand N-gram smearing*, an efficient improvement over the regular unigram smearing used as an approximation to the LM scores in a tree lexicon, is introduced. It is also shown how lattice rescoring, the generation of forced alignments and detailed phone-/state-alignments can efficiently be integrated into a single stack decoder.
The decoder named "Nozomi" [a]was tested on a Japanese newspaper dictation task using a 5000 word vocabulary. Using computationally cheap models it is possible to achieve realtime performance with 89% word recognition accuracy at about 1% search error using only 4 MB of total memory on a 300 MHz Pentium II. With computationally more expensive acoustic models, which also cover the for the Japanese language essential cross-word effects, more than 95% recognition accuracy [b]is reached.

[a] "Nozomi" is the name of the fastest, most comfortable and most expensive bullet train in Japan, and also means "hope" in Japanese

[b] which are currently the best reported results on this task

©ATR音声翻訳通信研究所

©ATR Interpreting Telecommunications Research Laboratories

Contents

1 INTRODUCTION 1
1.1 Organization of the paper 1
1.2 General Introduction 1
1.3 Technical Introduction 3
1.3.1 Definitions 5
1.4 Decoder types 7
1.4.1 Transition network decoders 7
1.4.2 Stack decoders 8

2 A MEMORY-EFFICIENT ONE-PASS STACK DECODER 9
2.1 Basic algorithm 10
2.1.1 Word-level search 10
2.1.2 State-level (word-within) search 10
2.2 Stack module 13
2.2.1 Lattice generation 13
2.2.2 N-best list generation 14
2.3 Hypotheses module 14
2.4 N-gram module 15
2.4.1 A disk-based N-gram 16
2.5 LM lookahead 16
2.5.1 Unigram smearing 17
2.5.2 On-demand N-gram smearing 17
2.6 Cross-word models 18
2.7 Fast-match with delay 20
2.8 Using word-graphs as language model constraints 20
2.9 Lattice rescoring 20
2.10 Generating phone-/state-alignments 21

3 EXPERIMENTS 21
3.1 Recognition of Japanese 22
3.2 Recognition results for high accuracy 22
3.3 Recognition results for high speed and low memory 24
3.4 Time and memory requirements for modules 25
3.5 Usage of cross-word models 25
3.6 Usage of fast-match models 25
3.7 Effect of on-demand N-gram smearing 26
3.8 Lattice/N-best list generation and lattice rescoring 27

4 CONCLUSIONS 27

5 ACKNOWLEDGMENTS 28

1 INTRODUCTION

Large vocabulary continuous speech recognition (LVCSR), here defined as the recognition of arbitrary, continuously spoken sentences using a vocabulary of 5000 words or more, is currently limited to workstations and fast high-end laptops with a lot of memory. To make LVCSR work on PDAs, cellular phones, user-interfaces, wrist watches etc., it is necessary find to time- and memory-efficient algorithms. The efficiency of the *search engine* of a speech recognition system, that takes as input an utterance and generates in its simplest form the most probable word string, is unfortunately not based on a single algorithm, but on a complex collection of ideas and implementation details which only in combination make the search efficient. While the basic ideas can often be stated in a few words, their details and the implementation, which is crucial for good performance, is often not obvious and should be explained to the necessary detail in those cases.

Because the search engine combines all parts (pronunciation dictionary, feature vectors, acoustic models, language models) of a speech recognition system, it often defines the formats for module communication and is to a great extent responsible for the overall complexity of the whole system. The author's observation is, that the problem of too marginal improvements of state-of-the-art LVCSR systems has its origin not necessarily in a lack of innovative ideas, but often is due to a lack of possibilities for a scientific procedure to test them. The reason is in general an overwhelming complexity of the complete system, and research has to be aimed at reducing it.

Therefore, the goal for implementation of any search engine must be to minimize **time** and **memory requirements** as well as the overall **complexity** of the system while maximizing its **flexibility** using all available knowledge sources to search for the desired output.

1.1 Organization of the paper

In the first (general) part of the introduction (section 1.2) the term "search" for speech recognition is used in a loose way and necessary requirements for a modern search engine are defined. In the second (technical) part (section 1.3) definitions for the used terms are given and explained using the necessary mathematical equations. In the third part (section 1.4) known decoder types are classified and briefly explained. Section 2 explains the details of a memory-efficient one-pass stack decoder. Section 3 shows experiments and results for a 5000 word Japanese newspaper dictation task using this decoder. Specific problems regarding decoding for the Japanese language are discussed. The paper concludes with section 4.

1.2 General Introduction

The essential content of any search algorithm for the best hypothesis in a LVCSR system can be summarized in simple words as:

1. Consider all possible hypotheses (different word sequences, pronunciations, alignments) using the dictionary
2. Assign a score to each hypothesis using the language model and the acoustic model
3. Put out the hypothesis with the highest score.

If this method would be applied in this form in practice, it would be impossible to find the best hypothesis because of the very large number of possible combinations of words, pronunciations and alignments for any reasonable sized dictionary in combination with the commonly used trigram language model.

As discussed above, the primary goal of any search algorithm must be to minimize the *time* and *memory* requirements for finding the best hypothesis while maintaining a minimal search error. Any practical search implementation (Alleva, 1997), (Gopalakrishnan, 1995), (Ney & Aubert, 1996), (Odell, 1995), (Paul, 1992), (Ravishankar, 1996), (Renals & Hochberg, 1996), (Schwartz, Nguyen & Makhoul, 1996), (Soong & Huang, 1991), (Robinson & Christie, 1998) based on 1st order Hidden Markov Models (HMMs) uses various methods to achieve that. Some of them are: the *Viterbi search* to linearize the search with respect to time, the *beam search* to heuristically reduce the number of hypothesis at any time point, the use of a *tree lexicon* for the pronunciation dictionary to share computations for beginnings of words, the *language model lookahead* (Steinbiss, Tran & Ney, 1994), to approximate LM scores within words, the *fast-match* (Bahl, de Souza, Gopalakrishnan, Nahamoo & Picheny, 1992) (Gopalakrishnan & Bahl, 1996) to generate quickly acoustically likely word hypotheses.

A second goal for a search engine that is used in a research environment, or in cases where the output of the search engine is used as input to post-processing modules like translation engines, is its *flexibility*. It is often not enough to allow as input only a sequence of feature vectors to produce a word sequence with the highest score. In many cases more detailed outputs like lattices, N-best lists or detailed word, phone, or state-alignments are required. As language model search constraints one might want to use arbitrary order N-gram language models, word-pair grammars, word-graphs to simulate finite state automatons, or transcriptions to produce forced alignments. These and other requirements for a modern search engine, from an expert user's point of view, can be listed as:

- **possible inputs:**
  - utterance feature vectors (for on-demand likelihood calculation) or precalculated likelihoods (as often produced by neural network based systems)
  - lattice in standard lattice format (SLF)
- **possible outputs:**
  - first-best hypothesis (text or SLF)
  - N-best (text or SLF))
  - lattice in SLF
  - phone-/state-alignments
- **tree lexicon** (possibly > 65536 words) with multiple pronunciations and optional pronunciation scores
- **possible LM search constraints:**
  - arbitrary order N-gram language models
  - word-pair grammar (with scores)
  - word-graph in SLF

  - word transcription (for forced alignment)

- support for **word-within/cross-word context-dependent acoustic models** of any context order without needing to change the monophone dictionary
- optional **disk-based LM** to save memory
- **efficient LM lookahead** (unigram-smearing or on-demand N-gram smearing) to incorporate LM scores in tree lexicon as early as possible
- optional use of **fast-match models** to speed up search

A third goal is the realization of the search in a single left-to-right pass, using all available search constraints as early as possible. This reduces overall complexity of the search process, is conceptually attractive and is essential for on-line systems. Being able to run the search in one pass of course doesn't imply that it *has* to be run in one pass. In many cases, especially in a research environment, it often turns out that multi-pass strategies are more time-efficient for finding optimal solutions.

1.3 Technical Introduction

Speech recognition relies on the framework of statistical pattern recognition (Bishop, 1995), (Duda & Hart, 1973), (Huang, Ariki & Jack, 1990), which has been shown to work well in practice. The goal for the search engine is to find the word sequence $\hat{W} = w_1, w_2, \ldots, w_M$ with the highest probability among all possible word sequences $\mathbf{W}$, which is conditioned on a feature vector sequence $\mathbf{X} = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{t-1}, \mathbf{x}_T$. Every word of the *dictionary* (see 1.3.1 for definition of terms), is usually mapped to a sequence of Hidden Markov Models (HMMs) (Huang et al, 1990), which themselves consist of *states* q, such that every word is equivalent to a Markov state sequence $Q = q_1, q_2, \ldots, q_{t-1}, q_T$. Using Bayes' rule $P(B|A) = P(A|B)P(B)/P(A)$ and the product rule of probability $P(A, B) = P(A)P(B|A)$ the conditional sequence probability $P(W|\mathbf{X})$ can be broken down to three terms and simplified as:

$$\begin{aligned}
\hat{W} &= \underset{W}{\operatorname{argmax}}\, P(W|\mathbf{X}) && (1)\\
&= \underset{W}{\operatorname{argmax}}\, P(\mathbf{X}|W) \cdot P(W) && (2)\\
&= \underset{W}{\operatorname{argmax}} \sum_{Q} P(\mathbf{X}|W, Q) \cdot P(W, Q) && (3)\\
&\approx \underset{W}{\operatorname{argmax}} \sum_{Q} P(\mathbf{X}|Q) \cdot P(W, Q) && (4)\\
&\approx \underset{W}{\operatorname{argmax}}\, \underset{Q}{\operatorname{MAX}}\, P(\mathbf{X}|Q) \cdot P(W, Q) && (5)\\
&= \underset{W}{\operatorname{argmax}}\, \underset{Q}{\operatorname{MAX}}\, P(\mathbf{X}|Q) \cdot P(W) \cdot P(Q|W) && (6)\\
&= \underset{W}{\operatorname{argmax}}\, \underset{Q \epsilon Q_W}{\operatorname{MAX}}\, P(\mathbf{X}|Q) \cdot P(W) \cdot P(Q) && (7)
\end{aligned}$$

Several assumptions have been made in this derivation: a) The likelihood of the feature vector sequence given the state *and* the word sequence is equal to the likelihood of the feature vector sequence given *only* the state sequence, $P(\mathbf{X}|W, Q) = P(\mathbf{X}|Q)$. This implies that all acoustic information is captured by the state sequence and is independent of the

actually uttered words. b) The sum over all possible state sequences for a particular word sequence is approximated by the single best state sequence, which is termed the *Viterbi approximation*. This assumption in general doesn't effect the result but greatly simplifies the actual search and makes it possible to speak of state alignments and actual word boundaries (which would be fuzzy, if this assumption wouldn't be used).

The remaining three expressions stand for:

a) The *observation likelihood* [1]

$$P(\mathrm{X}|Q) = \prod_{t=1}^{T} P(\mathrm{x}_t|\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_{t-1}, q_1^T) \approx \prod_{t=1}^{T} P(\mathrm{x}_t|q_t), \tag{8}$$

which is generally modeled by a continuous density Gaussian mixture model or by a neural network. The evaluation of $P(\mathrm{x}_t|q_t)$ during the search usually takes a great percentage (typically 40-80 %) (Beyerlein & Ullrich, 1995) of the actual search time, so effort has to be made to reduce the number of likelihood calculations as much as possible.

b) The *transition probability* of the state sequence within words

$$P(Q) = \prod_{t=1}^{T} P(q_t|q_1, q_2, \ldots, q_{t-1}) \approx \prod_{t=1}^{T} P(q_t|q_{t-1}), \tag{9}$$

which is usually approximated by a first order Markov model.

c) The unconditional probability of the word sequence (*language model probability*)

$$P(W) = \prod_{m=1}^{M} P(w_m|w_1, w_2, \ldots, w_{M-1}) \tag{10}$$

$$\approx \prod_{m=1}^{M} P(w_m|w_{m-1}, w_{m-2}, \ldots, w_{m-(N-1)}), \tag{11}$$

which is often approximated by an *N-gram*; the probability of a word given its $N-1$ predecessors.

In practical systems the search is never based on the raw probability estimates, but on their logarithms to stay in the given floating point range of current computers. This also converts the multiplications in (8), (9) and (10) to simpler additions. It is then usual to speak of a *score* rather than of a probability.

In practice it is found, that an exact implementation of (7) is often not optimal to achieve the best word recognition results. In general acoustic and language models are estimated on completely different corpora and many assumptions have to be made to make a practical implementation of a speech recognition system possible. To cope with these assumptions it is usually useful to weight the LM score against the acoustic score, which is often realized by a multiplication of the *language model score* ($log\ P(W)$) by a *language model scale factor* λ. Also, there is often a *word deletion penalty* WDP, which is added to the LM score at every word end. A high WDP encourages word insertions, therefore penalizes word deletions. For M words in the hypothesis the use of these two heuristic parameters can be summarized as:

$$LMscore = \lambda \cdot log\ P(W) + M \cdot WDP \tag{12}$$

[1] throughout this paper there is no distinction made between probability mass and density, usually denoted as P and p, respectively, because it is not necessary for discussion of the search

1.3.1 Definitions

Here definitions of terms are collected, which are frequently used in the context of search for speech recognition, and also in this paper.

word: the ASCII sequence defining a word in the conventional sense, for example "car"

word-ID: a unique identification number or ASCII sequence for any logical word in the dictionary (note that homonyms like "arm" (part of body) and "arm" (weapon) would have a different word-ID)

word-ID list: a list of all word-IDs that are used during the search

(physical) state: smallest units of the acoustic model, which are each characterized by a method (function) to calculate its observation likelihood $P(\mathbf{x}_t|q^{(i)})$ at any time; in typical systems there are between 500 and 30000 different physical states

(logical) state: smallest unit of an HMM model, is characterized by its observation number (from the physical state) and its directed connections to other logical states (transitions)

HMM model: a collection of logical states, typically three to model a phone plus a non-emitting init and exit state; in a tied-state system different HMM models can share several physical states

phone: smallest modeling unit for a word, represented by a single HMM model; there are context-independent phones (monophones) or context-dependent phones (triphones, quintphones etc.) – context-dependent phones that depend on information beyond word boundaries are called *cross-word models*

pronunciation: a sequence of phones which specify the pronunciation of a word; can have a pronunciation weight associated

recognition unit: a word-ID plus its pronunciation; equal word-IDs with different pronunciations (and vice versa) are different recognition units

dictionary: a list of recognition units (word-IDs plus pronunciation), optional outputs and optional pronunciation weights; three example lines:

```
arm_1 [arm] 0.234 aa r mh
arm_2 [arm] 0.456 aa r mh
armageddon 0.55 aa r mh ae g ae dd n
```

a word-ID can occur several times to account for alternative pronunciations of a word

tree lexicon: internal representation of the pronunciation dictionary; a tree-based collection of all pronunciations in the dictionary as *lexical nodes* each representing a phone (HMM model), such that equivalent beginnings of pronunciations are shared

lexical node: smallest unit of the tree lexicon, representing an HMM model; a node is an *end-node* if a pronunciation ends at it – note that end-nodes are not necessarily leaf-nodes of the tree ("arm" and "armageddon" share the first three phones and

"arm" ends within the pronunciation of "armageddon"); equal pronunciations will have the same lexical end-node

acoustic model: collection of HMM models, which allow the computation of $P(\mathbf{X}|Q)$ and $P(Q)$ for any valid state sequence; is typically based either on continuous density Gaussian mixtures, discrete distributions or on neural networks

language model: the module which allows the computation of

$$P(W) = \prod_{m=1}^{M} P(w_m|w_1, w_2, \ldots, w_{M-1})$$

N-gram: language model which makes the approximation

$$P(W) = \prod_{m=1}^{M} P(w_m|w_{m-1}, w_{m-2}, \ldots, w_{m-(N-1)}),$$

with N being typically three (trigram) or two (bigram); usually allows the computation of $P(W)$ for any W using a backoff procedure

word-pair grammar: language model which makes the approximation

$$P(W) = \prod_{m=1}^{M} P(w_m|w_{m-1})$$

for a limited set of word-pairs; $P(w_m|w_{m-1})$ for word-pairs not in the set are zero

hypothesis: a word sequence including its pronunciation and word start/stop times, which is hypothesized by the decoder

language model state: two hypotheses are in the same LM state, if their tail cannot be distinguished by the currently used language model (example: the LM histories "I love you" and "I don't love you" where the last two words are in the same LM state using a trigram LM)

first-best hypothesis: the hypothesis with the highest total score

lattice: a graph made out of *arcs* and *nodes*, containing all hypotheses considered during the search including all different alignments and pronunciation variants

standard lattice format (SLF): a lattice format that can be passed around easily between modules (usually an ASCII string); a useful format is suggested in (Young, Jansen, Odell, Ollason & Woodland, 1997)

node: part of a lattice, that joins partial hypotheses which end at the same time and are in the same LM state

arc: part of a lattice joining two nodes; an arc represents a recognition unit associated with at least its acoustic score

N-best list: the N best hypotheses, which differ by at least one word-ID (different alignments or pronunciations of the same word-ID sequence belong to the same hypothesis for this purpose)

state/phone alignment: every frame of an utterance labeled with a state number and a phone number

pass: one *pass* means to search once from left to right through the utterance (or from right to left) incorporating more of the available knowledge than in the last pass

full search: exhaustive search over all possibilities given the dictionary and the LM constraints ⇒ in general not feasible

beam search: at any time point t only partial hypotheses of a score within a *beam* around some best score at that point are kept ($L_t >= L_{BEST,t} - beam$); heuristic use of beams makes any search non-admissible

admissibility: a search is called *admissible* if the algorithm guarantees to find the best hypothesis

LM lookahead: heuristic approximation of the LM scores within words, usually used with a tree lexicon

fast-match: method to quickly find acoustically likely matches for words

stack: collection of partial word hypotheses

search error: error that is caused by the search algorithm (usually by too heavy pruning) and not by a badly estimated acoustic model or language model

1.4 Decoder types

Every decoder implementation is different and a clear distinction between different decoder types can often not be made. For this paper, it has been tried to distinguish them by their basic search strategy, namely the time-synchronous *transition network decoders* and the usually time-asynchronous *stack decoders*.

1.4.1 Transition network decoders

The majority of the decoders currently in use are transition network decoders (Alleva, 1997), (Murveit, Butzberger, Digalakis, & Weintraub, 1993), (Gauvain, Lamel, Adda, & Adda-Decker, 1994), (Ney & Aubert, 1996), (Odell, 1995), (Ravishankar, 1996), (Schwartz et al, 1996), (Shimizu, Yamamoto, Masataki, Matsunaga & Sagisaka, 1996), (Soong & Huang, 1991) which are based on a *transition network* of words (as HMM state sequences) that incorporates the used language model in its word transitions. In its simple static form all word-ends are connected to all word-beginnings via transitions that contain word bigram probabilities, such that the whole network can be viewed as a large first-order HMM containing thousands of logical states. This makes it possible to use the efficient and admissible *Viterbi algorithm* as well explained in (Rabiner & Juang, 1993, pp. 339-340) and (Young et al, 1997, pp. 11-13) to search for the optimal state sequence time-synchronously. Discarding states with a relatively low score at each time t has proven to efficiently reduce the amount of needed computation time to find the first-best hypothesis at no or little search errors. Pruning of states is often based on a heuristic beam around the best state or/and on a predefined number of states with a high score that remain active.

It is easy and efficient to use word unigrams and bigrams in such a network, because their scores can be incorporated into the transition network before the actual search starts, but it doesn't extend automatically to long-span language models (3-grams, 4-grams), which are necessary to reduce modeling assumptions and to achieve good performance in LVCSR. Long-span LMs are either incorporated through dynamic building of the network during the search or through multi-pass rescoring strategies (Schwartz et al, 1996), which are often also necessary to construct lattices or true N-best lists. These implementations require then a dynamic LM score lookup which is not needed when only unigrams and bigrams are used.

Since transition network decoders are run time-synchronously, meaning the state-space evaluation over for $t+1$ is done after it was done for t, it is possible to run real on-line recognition without any additional delay imposed by the decoding algorithm.

1.4.2 Stack decoders

Stack decoders can be defined as decoders that during decoding use some kind of a *stack* of partial sentence hypotheses each consisting of a certain number of words. In general the partial hypotheses on a stack are expanded by complete words time-synchronously using the dictionary to create new partial hypotheses which are inserted into other stacks. When all stacks except the last (result stack) are empty, the result stack will contain the first-best hypothesis, the N-best hypotheses or the respective lattices depending on the search mode.

Although in the context of decoders the storage container for partial hypotheses is historically called *stack*, which should be a Last-In-First-Out buffer (LIFO) given its name, it is in practice rather often a simple list or a tree of hypotheses ordered by some kind of total score. The total score the hypotheses on the stack(s) are ordered by can be a) the partial hypothesis' log-likelihood, b) an estimate of the log-likelihood of the complete utterance (A^* criterion) (Soong & Huang, 1991), or c) some other score that expresses the belief in the partial hypothesis' correctness (Gopalakrishnan, 1995), (Renals & Hochberg, 1996).

There are at least two different types of implementations for stack decoders: a) with only one stack that contains all partial hypotheses which might have different end-times (Paul, 1991), (Paul, 1992) or b) with one stack for each time point, where each stack contains only hypotheses ending at that time (Renals & Hochberg, 1996). If there are many stacks, the *stack expansion* can either be time-synchronous (expand stack t before expanding stack $t+1$, which has been termed start-synchronous in (Renals & Hochberg, 1996) or time-asynchronous (any stack can be expanded next, completely or partially, depending on some algorithm to pick a stack that will probably lead to the first-best hypothesis (Gopalakrishnan, 1995)). Even when the stack expansion is time-synchronous, stack decoders are often said to search time-asynchronously, because the global state progression through the utterance is in general not time-synchronous like for transition network decoders.

All stack decoders operate at least on two levels of search: a) the outer level, which loops over the stacks (*word-level search*), and b) the inner level, which loops over time and states (or states and time (Robinson & Christie, 1998)) to search for complete words, starting from the end-time of the hypothesis to expand, which is called *state-level search* or word-within search. Every time a potential word-end is found during the time-synchronous word-within search, its language model score is looked up using the found word plus its

history using the hypotheses which are to be expanded. Because the dynamic LM score lookup can take any word history into account, stack decoders can easily make use of any kind of N-th order Markov language model and also of non-Markov language models like link grammars etc. Especially N-gram models of any order are simple to implement (section 2.4), which is one of the major advantages of stack decoders over the transition network decoders.

The decoupling of the language model from the Viterbi search in the state space has several other advantages. Because the hypotheses generation is completely independent of the word-within search, the word-within search can be realized memory-efficiently without the need for token passing or backtrace pointer storage (section 2.1.2). Word lattices can be created easily in the first pass at little computational overhead (section 2.2.1). Using a similar procedure N-best lists can be created, optionally with all different alignments and pronunciation variants in a lattice within each N-best hypothesis, again in the first pass (section 2.2.2). LM lookahead procedures depending on the scores of the word history to expand are easily integrated as a separate module (section 2.5).

In stack decoders there are several ways to implement cross-word context-dependent acoustic models, which are necessary for good recognition results. A procedure shown to be computationally efficient for cross-word models of *any* context order is discussed in section 2.6. This procedure leads naturally to a possible use of fast-match models to generate acoustically likely word candidates quickly. In this paper a novel version of using a fast-match in a stack decoder is discussed (section 2.7), which avoids some disadvantages of earlier implementations.

Historically stack decoders have often been used for lattice rescoring to integrate higher order LMs and to optimize search parameters, often in combination with A^* procedures (Soong & Huang, 1991). This type and other types of often needed lattice rescoring procedures are discussed in section 2.9, which all can be implemented as additions to the regular decoder.

The usage of word-graphs constraining the search using stack decoders is closely related to the usage of word-pair grammars and the generation of forced word-alignments (section 2.8). Detailed phone- and state-alignments, which are not available when, like mentioned above, no state-based backtrace pointers are stored, will have to be created on demand. This turned out to be particularly easy for the implementation described in this paper (section 2.10).

One disadvantage of stack decoders is the fact that they usually evaluate the state space time-asynchronously within a certain range, which makes real online decoding impossible – there will be a time lag being equal to the range of the state evaluation. Although in practice this time lag is short (less than a second) compared to other time limiting factors during a real search and can also be avoided during silences, it might pose a problem in systems that must have a human-like response time.

A second principal disadvantage is that it is not possible to merge logical state theories within words, because the word-level search is separate from the word-within search, which is discussed in more detail in section 2.1.2.

2 A MEMORY-EFFICIENT ONE-PASS STACK DECODER

This section describes the details of a memory-efficient one-pass stack decoder, that is based on a multi-stack implementation with one stack per time frame, which is equivalent

to a one-stack implementation with the stack entries ordered primarily by time and then by score.

2.1 Basic algorithm

As discussed above, a stack decoder works on two levels of search, the word-level search looping over stacks and the state-level search looping over time and logical states.

2.1.1 Word-level search

Looping over stacks for the word-level search can be done time-synchronously (start-synchronously) or time-asynchronously depending on the stack expansion mode. Independent of this mode, which is a function of the *stacklist* (collection of all stacks), the basic word-level search, as shown at the end of this section, works as follows: First an initial temporary stack *stack* containing only an initial empty root hypothesis is generated. Then all partial hypotheses on the temporary stack *stack* are extended by one word using the state-level search that knows about the *stacklist*, such that the new partial hypotheses can be inserted into the correct stacks. When the current temporary stack is finished, a new temporary stack is popped from the *stacklist*. This can be any of the currently held stacks in *stacklist*, which will be the earliest one in time in case of a synchronous stack expansion, and any one of the available ones in case of a asynchronous stack expansion depending on the selection criterion. The temporary stack doesn't necessarily have to contain *all* partial hypotheses of the stack in *stacklist* it was generated from. Again, depending on the selection criterion, these could be only a subset of that stack. When there are no more stacks to be popped, the method finishes with returning the result (first-best, N-best, lattice, etc.). An example implementation using pseudo C++ code would be:

<u>Word-level search:</u>

```
{
  stack = stacklist.GET_INITIAL( hyp.ROOT() );

  do
  {
    statelevel_search.EXTEND( stack );

    stack.FORGET();
  }
  while( (stack = stacklist.POP()) );

  return( stacklist.RESULT() );
}
```

2.1.2 State-level (word-within) search

The search on the state level extends all hypotheses of the passed temporary stack by one word using the pronunciation dictionary and inserts all generated new hypotheses in the corresponding stacks provided they are in the beam. The search is based on the pronunciation dictionary *dict* which is organized in a tree structure such that equivalent

beginnings of pronunciations are shared to save redundant computations. This tree lexicon consists of *lexical nodes*, with each node pointing to its associated HMM and all possible recognition units ending at it. The lexicon has a single root-node that doesn't have an HMM associated and defines the beginnings of all words. A node is called *active* if any of its logical HMM states is within the current beam. If a node is active, it carries its current time t and the log-likelihoods of all its states in a dynamically allocated chunk of memory. This memory is released to be used by other nodes if a node is deactivated.

During the state-level search, as shown at the end of this section, it is necessary to keep a list of all active nodes for the current and the next time slice (*alist, alist_next*), which are accessed by PUSH and POP operations. These lists contain only pointers to the corresponding lexical nodes and have to be ordered by the levels of the tree lexicon, such that the nodes closest to the root-node are popped first. This is necessary to insure that during actual propagation all states within a node are in the same time slice.

The state-level search then works as follows: After both active node lists are cleared, the non-emitting root-node of the tree lexicon is activated with the score of the best hypothesis of the stack to expand. It is then pushed on the current active node list (*alist*). The start time for the word-within search is the end-time of the stack to expand plus one.

The active nodes are propagated time-synchronously through the tree lexicon until the end of the utterance T (or some maximum word length) is reached or all nodes fell out of the beam and have been deactivated. The active nodes are popped from the list and forward-propagated one time step assuming they have been in $t-1$ (`FORWARD()`). Forward propagation involves one Viterbi step within the currently worked on node. Since the node cannot be left during that step, it is sufficient to calculate only the new scores for every node-internal state without using any back-pointers. Although not containing much source code, method (`FORWARD()`) will take the largest part of the actual search time because the time consuming observation likelihood calculation functions for the physical states are called from it. Care should be taken in the loop ordering within `FORWARD()`, such that the expensive likelihood calculation functions are only called when actually needed. Also, already calculated likelihoods should be cached because in time-asynchronous stack decoders they will be used several times even when there are no shared physical states.

After the forward propagation the upper bound of the score at the current time is updated using `UPDATE_UPPERBOUND()`, if the pruning procedure is based on the beam around the best score at any time. It is not necessary when the hypotheses on the stacks are not popped depending on their partial log-likelihood as used in (Gopalakrishnan, 1995) or (Renals & Hochberg, 1996).

If any state of the current node is in the beam, it is a possible candidate for causing a stack expansion, otherwise it is deactivated. If a node in beam has its non-emitting exit-state activated and the node corresponds to a word end, the hypotheses on the temporary stack are expanded by one word (`stack.EXTEND()`), which involves looping over all hypotheses and all recognition units ending at this node, looking up the LM score for $P(rec_unit|hyp_history)$, generating the extension if the new partial hypothesis is within the current beam, and pushing it on the corresponding stack. Then, only if the exit-state is active, all successor nodes in the tree lexicon are activated (`ACTIVATE_SUCCESSORS()`), which involves copying the exit-state score of the current node into the init-state of the successor node, and pushing the node on the active node list for the next time slice (*alist_next*). Also, any nodes that are in the beam regardless their exit-states have to be pushed on this list.

Note that because of the LM lookahead procedure explained in section 2.5, which leads

to an overestimate of scores within words, it is possible to use a *lower* (tighter) beam at word-ends compared to the beam within words.

Finally, when all nodes of the current time slice are finished, the two active node lists are swapped and the time is incremented to be ready for the next time slice.

A possible state-level search implementation, that was found to be efficient, is:

State-level search:

```
{
  alist.CLEAR();
  alist_next.CLEAR();

  dict.ROOTNODE.ACTIVATE( stack.TOPHYP.SCORE() );
  alist.PUSH( dict.ROOTNODE() );

  t = stack.TOPHYP.TIME() + 1;

  while( t < T && alist.NOT_EMPTY() )
  {
    while( (node = alist.POP() )
    {
      FORWARD( node, t );
      UPDATE_UPPERBOUND( node, t );

      if( node.IN_BEAM(t) )
      {
        if( node.EXIT_STATE.ACTIVE() )
        {
          if( node.IS_WORD_END() )
            stack.EXTEND( node, t-1 );

          ACTIVATE_SUCCESSORS( alist, node, t );

          node.EXIT_STATE.DEACTIVATE();

          if( node.NO_STATE_ACTIVE() )
            node.DEACTIVATE();
        }
        if( node.ANY_STATE_ACTIVE() )
          alist_next.PUSH( node );
      }
      else
      {
        node.DEACTIVATE();
      }
      alist.SWAP_WITH(alist_next);

      t++;
```

```
    }
  }
}
```

As pointed out in section 1.4.2, it is not possible to merge logical states within words from a state-level search that started at a different time like it is possible for transition network decoders. One obvious technical reason for this is that the status of intermediate states during any time of the state-level search is not stored, because it would require additional effort, time and memory. Another reason is the explicitly wanted decoupling of the state-level search from the LM, which prohibits any logical state merging, because the same logical state given only the dictionary will be a different one depending on its history, if as LM anything else than a first order Markov model (bigram or unigram) is used.

2.2 Stack module

The collection of *stacks* for each time t are accessed by PUSH() and POP() operations taking partial hypotheses as arguments. Because they are used frequently and usually contain a few to several hundred entries in a typical application, the *stacks* (or more precisely lists as discussed above) have to be set up efficiently. The container types used in other decoders are often special tree-structured lists, which are ordered by score and limited in the number of entries (Renals & Hochberg, 1995b). Here a different method is described which was found to be most efficient and simple to implement.

Pushing a hypothesis on a stack involves a check whether a hypothesis in the same LM state is already on that stack. If yes, the scores of the two hypotheses are compared and the better one is inserted into the stack, the other one discarded. In case of an N-gram LM the LM state check means to compare the last $MAX(N-1,1)$ history word-IDs. One word has to be compared as a minimum to not violate the at least first order Markov assumption for the complete speech model. Although checking for LM state equivalence for N-gram LMs can theoretically be done in $O(1)$ using a hash table with the $N-1$ words history as the key, it was found that it is in practice not more efficient than a simple non-ordered unlimited list that is searched through linearly up to an average stack size of a few hundred hypotheses. Pushing a hypothesis on a stack can also improve the upper bound for the score at this time, which has to be checked for. Popping a hypothesis from a stack is an $O(1)$ process, since it doesn't matter in what order the hypotheses in beam are extended for the implementation described here.

The stacks containing mainly pointers to hypotheses can be set up efficiently in a ring-buffer if a maximum word length is defined, which is necessary for on-line operation.

2.2.1 Lattice generation

Lattices, as defined in section 1.3.1, are a convenient form of storage for the hypotheses that are considered during the search, and their generation is often necessary for systems that need to post-process the recognition output such as for translation engines, information retrieval systems, or multi-pass search strategies. Stack decoders can easily generate lattices with little computational overhead in the first pass by slightly modifying the LM state check procedure. Instead of discarding the hypothesis with the worse score in case of LM state equivalence it can be linked into the lattice. A pointer on the best arc back has

to be updated to not loose the best hypothesis for the current LM state and future reference. Compared to the generation of the first-best hypothesis there is only little overall increase in memory for the storage of the additional arcs in the lattices (section 3).

2.2.2 N-best list generation

The hypotheses in an N-best list differ by at least one word-ID. This can directly be checked for by extending the LM state check procedure to the complete history instead just the $MAX(N-1,1)$ history word-IDs like necessary for obtaining the first-best hypothesis. It can be done either exactly by checking each word, or approximately by using a hash function for the history. A lattice within the N-best list, referred to as N-best lattice, which includes all possible alignments and pronunciation variants for the same word-ID sequence in the possible paths taken backwards from a lattice node, can be produced by merging hypotheses instead of replacing them like discussed above for the first-best lattices. Compared to the lattice generation this procedure uses only little additional memory for the extra nodes of the hypotheses, which are needed because of the increased LM state space, and only little additional time as shown in section 3. Since for the generation of N-best lists only the LM state check procedure was modified, they can be generated in the first pass like lattices.

2.3 Hypotheses module

A hypothesis in memory is made out of objects called *hyp-nodes* and *arcs*, starting at $t = 0$ from a single root node, and ending in either one end-node (first-best or lattice) or in many end-nodes (N-best list, N-best lattice). An example is shown in Fig. 1. Each node contains its time and best total score of the hypothesis up to this point. The arcs connected to ancestor nodes (parent nodes) are set up as a single linked list starting from the current node, which also contains a pointer to the arc belonging to the best hypothesis going back from this node. Every node contains also a counter on how many *kid-nodes* it is connected to (how many arcs contain a pointer to the current node), which is necessary for efficient memory management of these objects.

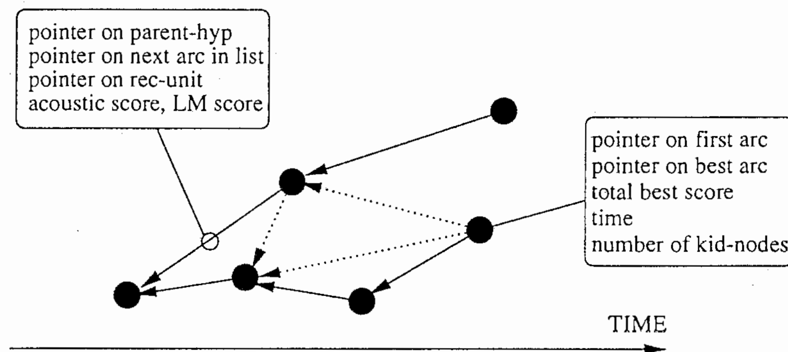


Fig. 1: Example for the memory-efficient storage format for hypotheses made out of *hyp-nodes* (black dots) and *arcs* (arrows). Shown is a lattice, all arcs but the best are dotted.

An arc defining a recognition unit with scores will contain at least a pointer on the recognition unit in the dictionary, the acoustic score for it, a pointer on its parent hyp-node, and a pointer to the next arc of the linked lists of arcs (see Fig. 1).

Memory management of hyp-nodes and arcs is best set up using linked lists, such that getting or forgetting them can be done using simple pointer copying. The usage of OS memory management routines can be minimized by allocating blocks of objects if none are left in a buffer made out of linked lists of the needed objects. Forgetting a pruned hyp-node involves also forgetting all linked arcs. Forgetting an arc means also forgetting all hyp-nodes they point to, if these hyp-nodes have no kid-nodes. If implemented in this recursive manner, the total memory for the hypotheses used during the search will be approximately proportional to the active number of hyp-nodes and arcs, which is generally between 10^2 (first-best mode) and 10^5 (lattice mode) for the average LVCSR application. In the implementation described here one hyp-node occupies on average $\approx$ 30 bytes, one arc $\approx$ 20 bytes.

2.4 N-gram module

The N-gram module is responsible for generating $P(w_N|w_1, w_2, \ldots, w_{N-1})$, the probability of a word given its $N-1$ words history, which is in general stored in a lookup table and might require backing off to lower order N-grams using the approximation $P(w_N|w_1, w_2, \ldots, w_{N-1}) \approx P_{backoff}(w_1, w_2, \ldots, w_{N-1}) \cdot P(w_N|w_2, w_3, \ldots, w_{N-1})$. The N-gram of an average LVCSR system usually occupies the most memory and is accessed on average a few hundred to a few thousand times per frame, so it has to be stored in a format that is memory-efficient and allows fast access.

A useful format was found to be the following, which is shown in Fig. 2: For a back-off N-gram LM store all n-grams with $n = 1, 2, \ldots, N$ in a table for each n. Each entry in a table has a word-ID, its LM probability and back-off probability, and a pointer to the beginning of the list of extension word entries in the table holding the $(n+1)$-grams. For the table with the N-grams the pointers are not necessary, since no higher order $(N+1)$-grams are following. Each part of an entry table holding a particular set of extension words is ordered by its word-IDs to allow fast access using a binary search. The number of a set of extension words on any level n doesn't have to be stored because it can be calculated by subtracting the pointer (on level $n-1$) on the current set from the next pointer (also on level $n-1$) on the next set. If the next set on level n doesn't happen to have any extension words, indicated by a NULL pointer on level $n-1$, the next non-NULL pointer on level $n-1$ has to be searched for, which is usually not more than a few entries away. The last entry on any level n has to be treated as a special case – the number of extension words has to be calculated as the pointer difference between the entry and the entry at the beginning of the next level, if levels are stored consecutively in memory.

The memory requirements for this N-gram representation are 8 bytes per entry for all $\{n < N\}$-grams, and 4 bytes for all N-grams, assuming 4-byte pointers, 2-byte word-IDs and 1-byte representations for the LM probability and the back-off probability, uniformly distributed across their log-scores, which was found to be a sufficient accuracy to not cause any errors. Access time for this storage format is of $O(1)$ for the unigrams and of $O((n-1) \cdot log_2(K))$ for the $\{n > 1\}$-grams using a binary search, with K being the average number of words following any n-gram entry. The average access time can be slightly improved by caching LM states and their scores in a hash table for all $\{n > 1\}$-grams that have been accessed before. This improves average access time to $O(1)$ for already used $\{n > 1\}$-grams, but requires an additional check whether a certain LM state is already in the hash table or not.

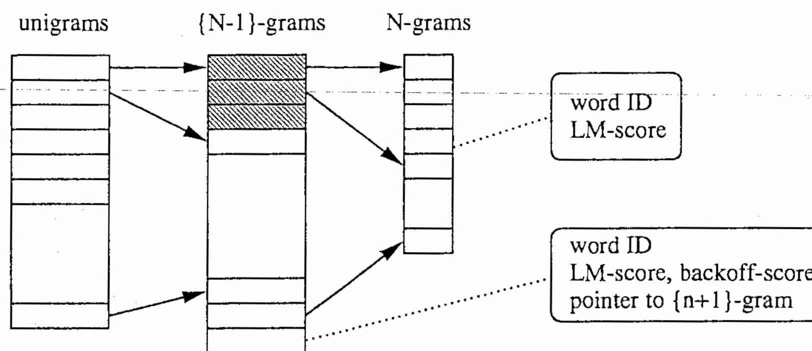


Fig. 2: Storage format for a fast accessible and memory-efficient N-gram, which is used in the same form for its disk-based representation. The shadowed region is an example for N-grams that would be loaded into memory for the disk-based LM to search for the correct entry.

2.4.1 A disk-based N-gram

It has been found that for average LVCSR applications most of the entries in an N-gram are never actually used and a disk-based representation of the N-gram can limit memory requirements to a few hundred kB for N-grams of *any* size (Ravishankar, 1996). The search for the N-gram scores on disk during the search is of course very time-consuming and has to be minimized using an efficient caching scheme. An efficient implementation was found to be the following: Unigrams are stored in memory and all $\{n > 1\}$-grams are stored on disk in the exact same format that was used for the representation in memory from section 2.4, such that looking up an n-gram can be done using the same algorithm. A set of extension words following an n-gram is loaded into temporary memory to run the binary search for the correct word-ID in memory and not on disk. Care must be taken in making this temporary buffer large enough to definitely include all information that is necessary to calculate the number of extension words for any entry within the set of extension words. The LM states that have been used once are cached in a memory-based hash table to minimize disk access. An alternative to caching only the used LM state is to cache all LM states that belong to any set of extension words loaded during the search for the required LM state.

2.5 LM lookahead

Most current LVCSR systems use some kind of LM lookahead to approximate the LM scores of the possible current LM states within words and use the exact LM scores only at word-ends. When a tree lexicon is used, the exact LM state often cannot be known until reaching a word-end node. The use of LM lookahead probabilities $p_{lookahead}$, which belong to every node in a tree lexicon, can speed up the search considerably, because nodes with a weak LM score can be pruned early. Suppose the LM lookahead probabilities are already set, they are used during the search through the tree lexicon as:

- When a node is entered, add $p_{lookahead}(node)$ to current total score.
- When a node is left, subtract $p_{lookahead}(node)$ from current total score.

2.5.1 Unigram smearing

Unigram smearing (Steinbiss, Tran & Ney, 1994), (Alleva, Huang, Hwang, 1996), (Ortmanns, Ney & Aubert, 1997) is a commonly used procedure and heuristically sets $p_{lookahead}$ for each node in the lexicon is as follows:

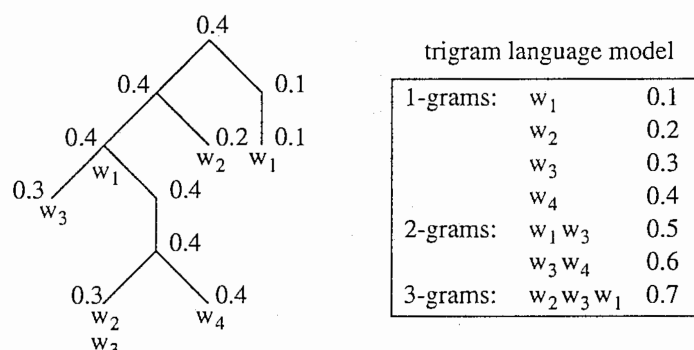
1. Calculate for each word-end node in the lexicon the maximum of all unigram scores of the words that end at this word-end node (a set of words denoted as $W_{word-endnode}$). Note that several words could end at one word-end node because of homonyms and multiple pronunciations.

$$p_{lookahead}(\text{word-end node}) = MAX\{P(w)\} \quad \text{with} \quad w \in W_{word-endnode}$$

2. For all non-word-end nodes set $p_{lookahead}$ recursively to the maximum of all child-nodes.

$$p_{lookahead}(\text{non-word-end node}) = MAX\{p_{lookahead}(\text{child-nodes})\}$$

Note that unigram smearing is independent of the currently extended word hypothesis and is therefore a *static* procedure – it has to be calculated only once which can be done in advance. An example is shown in Fig. 3.



1-grams:	w1	0.1
	w2	0.2
	w3	0.3
	w4	0.4
2-grams:	w1 w3	0.5
	w3 w4	0.6
3-grams:	w2 w3 w1	0.7

Fig. 3: Example for unigram smearing using a tree-lexicon and a trigram LM, which is independent of the hypotheses to extend.

2.5.2 On-demand N-gram smearing

On-demand N-gram smearing is a LM lookahead procedure that incorporates the LM state constraints of the currently extended hypotheses including their scores (Neukirchen & Willet, 1997). This results in better estimates of the real LM probabilities, compared to the regular unigram smearing procedure, which leads in turn to more accurate pruning and therefore can lead to a faster search. The algorithm works as follows:

1. Initialize all $p_{lookahead}$ with the unigram smearing procedure shown above before the search starts.

2. Calculate, for each set of word hypotheses H_i to expand, the maximum N-gram probability $P(w|H_i)$ of all *existent* N-gram entries (H_i, w) in the language model excluding the unigrams, because they were already set during the unigram initialization

(step 1). Identify the corresponding word-end nodes belonging to w (which could be several because of homonyms and multiple pronunciations) and set $p_{lookahead}$ to the maximum of the calculated probability and the unigram $p_{lookahead}$ probability already set.

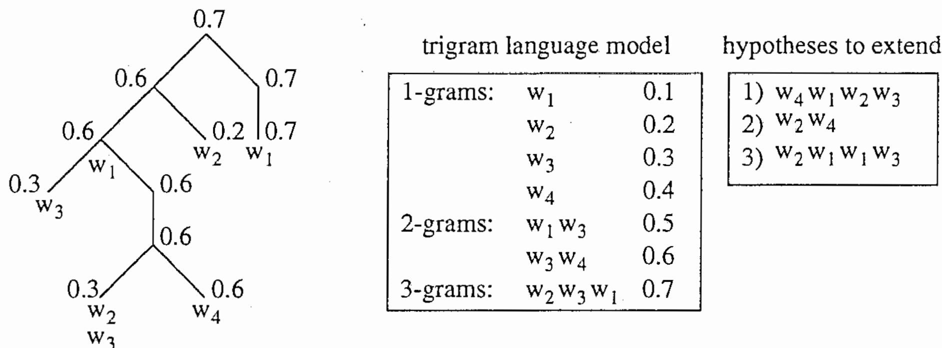
$$p_{lookahead}(\text{word-end node}) = MAX\{P(w|H_i)\} \;\forall\, H_i \quad \text{and} \quad \forall\, w \in \{(H_i, w) \text{ existent in N-gram}\}$$

To use not only the LM states but also the relative scores of the current hypothesis to the best hypothesis in the current set to extend use $MAX\{P(w|H_i)\} - \text{score}(H_{top}) + \text{score}(H_i)$ instead of $MAX\{P(w|H_i)\}$.

3. For all non-word-end nodes set $p_{lookahead}$ to the maximum of all child-nodes.

$$p_{lookahead}(\text{non-word-end node}) = MAX\{p_{lookahead}(\text{child-nodes})\}$$

Note that this procedure has to be invoked each time a new set of word hypotheses is extended, which cannot be done in advance like for unigram smearing. In spite of the additional computation the more accurate LM probabilities lead to more accurate pruning which can lead to a speed-up of the whole search. An example for this procedure is shown in Fig. 4, which should be compared to Fig. 3 illustrating the unigram smearing procedure.



1-grams:	w_1	0.1
	w_2	0.2
	w_3	0.3
	w_4	0.4
2-grams:	$w_1 w_3$	0.5
	$w_3 w_4$	0.6
3-grams:	$w_2 w_3 w_1$	0.7

hypotheses to extend
1) $w_4 w_1 w_2 w_3$
2) $w_2 w_4$
3) $w_2 w_1 w_1 w_3$

Fig. 4: Example of on-demand N-gram smearing using a tree-lexicon and a trigram LM using the constraints from the hypotheses to extend.

Results from experiments showing the impact of on-demand N-gram smearing compared to unigram smearing are shown in section 3.

2.6 Cross-word models

Cross-word models are context-dependent acoustic models that go over word boundaries. Their use in any decoder is complicated and time consuming. Various implementations have been described (Bahl, de Souza, Gopalakrishnan, Nahamoo & Picheny, 1993), (Alleva, 1997), which are in the case of transition network decoders often limited to cross-word triphones. Whether cross-word modeling is really necessary or not depends heavily on the speaking style and on the definition of a word in the language to be recognized, and is more likely to make a difference when there is no pause at word boundaries. In this paper it is shown that cross-word modeling is essential for recognition of read newspaper articles in Japanese (section 3).

A procedure to deal with cross-word models of *any* order (triphones, quintphones, etc.) incorporating cross-word effects in a delayed manner was found to be very efficient in time and memory requirements, and is especially well suited for a stack decoder:

- Run the state-level search for any set of hypotheses to expand with only word-internal context-dependent models.
- When popping the hypotheses from a stack to expand, realign the last M words using cross-word models at the word boundaries before entering the state-level search to find the extension words.
- Because cross-word effects are incorporated with a one-word delay, it is also necessary to realign the last M words for all hypotheses on the final result stack.

This procedure as illustrated in Fig. 5 incorporates all cross-word effects within the last M words, and is optimal for cross-word triphones with $M = 2$ for most cases and possibly $M = 3$, if the word before the last word is a one-phone word. To capture all cross-word effects with quintphones theoretically $M = 5$ is necessary, if all words in the dictionary would be one-phone words.

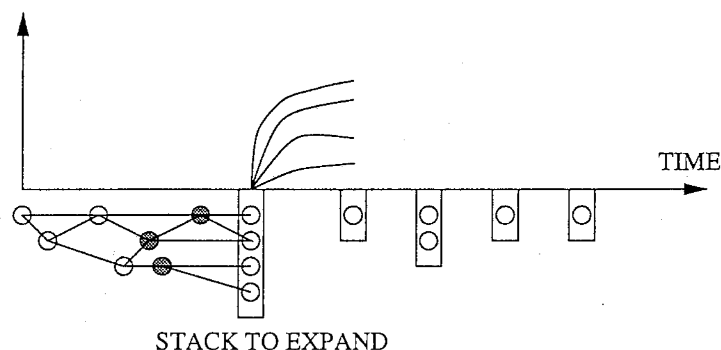


Fig. 5: Visualization of the method to incorporate cross-word models of any context order. Circles denote hyp-nodes, filled circles are the word boundaries that are corrected by the procedure using cross-word models *before* the stack (box) is expanded. In this example only two words are realigned, but there could be more like discussed in the text. The same method is used for the fast-match to rescore acoustically likely word candidates (section 2.7).

The realignment for each hypothesis to extend is in detail done as follows: Take the last M words and find the correct (cross-word) HMMs for each phone at the word boundaries which don't already cover the maximum available context given the acoustic model set. Use a local Viterbi search to find M new acoustic scores and possibly $M - 1$ new word boundaries. Generate M new arcs and $M - 1$ new hyp-nodes and replace the old hypothesis end-hyp-node by the new one.

The *correct* cross-word HMM model is defined as the model which covers the most context around the current center-phone. This definition is also used for finding the correct context-dependent HMM within words during construction of the tree lexicon containing context-dependent models given only a monophone pronunciation dictionary.

Compared to the procedure described in (Bahl et al, 1993), which locally rescores every word that is found during the state-level search, the method described here rescores only words that have been found to be considerably likely being part of stacks to expand. The

average number of hypotheses to expand per frame is in general between five and one-hundred and cross-word rescoring is only applied to those few. This requires only very little temporary memory and is fast, because of the low number of hypotheses and because of the fact that most of the states to be evaluated during rescoring for their observation likelihood are already in cache.

A potential drawback of this method is that because cross-word effects are incorporated delayed, scores might vary more during the lookahead, which might require larger beams than if this delay wouldn't be used.

2.7 Fast-match with delay

The method to handle arbitrary cross-word effects from section 2.6 is easily extended to allow an efficient acoustic fast-match with a one-word delay, which in a similar form without delay is described in (Bahl et al, 1992), (Gopalakrishnan & Bahl, 1996). The basic idea of a fast-match in a stack decoder is to use simple acoustic models to find possible extension words, and rescore them locally with better, but computationally more expensive models. This avoids the use of expensive models for the initial state-level search and can speed up the complete search substantially.

The fast-match procedure described here (see Fig. 5) keeps the use of the expensive models at a minimum and is almost identical with the method to incorporate cross-word models. Instead of using word-within context-dependent (CD) models for the state-level search, simple monophones with a low number of Gaussians per mixture or small neural-network based models are used in a context-independent tree-lexicon, and the found words are inserted in the corresponding stacks. Rescoring of the last M words including all cross-word effects is done later using the accurate, but expensive CD models, but only when a stack is expanded, such that many of the previously found words will be out of the beam. The difference to the cross-word procedure from section 2.6 is, that *all* phones of the last M words have to be mapped to their correct CD HMM model, and not only the ones at the word boundaries. As described above, this can be interpreted as local rescoring with a one-word delay, which limits the number of necessary rescoring turns per frame to less than ten to one-hundred for most applications, and requires very little additional memory.

2.8 Using word-graphs as language model constraints

For some applications it is necessary to constrain the search by a finite state grammar, a word-graph or a word-pair grammar, possibly with transition scores. This can be done efficiently in a stack decoder by activating only the pronunciation paths in the tree lexicon that correspond to possible word extensions of the hypotheses to expand. This has to be done on demand before entering the state-level search every time a stack is expanded. The state-level search will only consider the limited number of activated paths which will speed up the search substantially (section 3).

The generation of forced word alignments can be interpreted as a search constrained by an extremely simple word-graph consisting of the transcription of word-IDS, which might have several pronunciation variants.

2.9 Lattice rescoring

There are two types of often needed lattice rescoring procedures:

I) Use a given word-graph plus the word alignments and the acoustic scores, and change only the LM (often a higher order N-gram) or/and change LM parameters (LM scale and word deletion penalty).

II) Use only a given word-graph ignoring alignments and acoustic scores to constrain the search – use new acoustic models and a new LM. This has been described in section 2.8.

Type I is often done using A^* procedures in a separate search module, because the existence of the complete word-graph with scores allows an efficient estimate of score of the remainder, which is necessary for any A^* procedure (Nilsson, 1971), (Soong & Huang, 1991). In this case there is usually one stack, which is ordered by the A^* score. For the stack decoder implementation with many stacks like it is described here, a simple replacement of the state-level search makes it possible to integrate lattice rescoring of type I within the stack decoder framework. Instead of the original state-level search through the tree lexicon the possible extension words and their scores for every hypothesis to extend are already calculated in the lattice, so they just have to be located and inserted into the corresponding stacks. Because all other modules stay the same, implementation is simple and all outputs that have been possible before for sequences of feature vectors as inputs (first-best, N-best, first-best lattice, N-best lattice), are then possible for lattices as inputs. Because the time-consuming state-level search doesn't have to be done, this type of lattice rescoring is fast also for large lattices of a few thousand arcs, usually taking between 1/100 and 1/10 realtime. Memory requirements are the same as for the regular search minus the memory that is needed for the state-level search.

2.10 Generating phone-/state-alignments

Because state-level backpointers are not stored, phone- or state-alignments have to be created on demand. After a first-best word hypothesis is created, every word is state-aligned using the same routines which are necessary for the cross-word rescoring from section 2.6. For a forced state-alignment the word transcription has to be provided as additional input as a word-graph (section 2.8).

3 EXPERIMENTS

All experiments were conducted using the described one-pass stack decoder for the recognition of read sentences from a Japanese newspaper using a 5000 word pronunciation dictionary with on average 1.5 pronunciations per word. Larger pronunciation dictionaries for Japanese are currently not publically available. The acoustic models are gender-dependent decision tree state-clustered Gaussian mixture models trained on 20k sentences per gender from the ASJ and JNAS database of approximately 60 h of speech. Acoustic preprocessing is standard 12-dimensional MFCCs plus log energy, with applied cepstrum mean subtraction per sentence and first derivatives every 10 ms. A trigram and fourgram language model were trained on around 45 million words from the RWC corpus containing four years of newspaper articles from the Mainichi Shinbun, a regular daily newspaper in Japan. The standard test data are the first ten sentences from the speakers 006, 014, 017, 021, 026, 089, 102, 115, 122 from the JNAS database. All acoustic models, the initial language models and the initial pronunciation dictionary have kindly been provided by the IPA group (Kawahara et al, 1998), which also defined the test set.

3.1 Recognition of Japanese

Speech recognition of Japanese adds a few problems not occurring in Western languages. Japanese has no spaces between words, so the definition of a word for the dictionary is in general not obvious, but the databases used here are already subdivided into words, and word error rate is calculated using these word definitions. Unfortunately the subdivision in words is often ambiguous, which leads to recognition errors (example in English: 'awhile' recognized as 'a' 'while' and vice versa), that shouldn't be counted as errors in Japanese since there are no spaces. Because words are defined by grammatical analysis, there are often no pauses between them, which makes it essential to use cross-word models for Japanese, if this currently common word definition is used (section 3.5). These errors are here referred to as *type I* errors.

A second problem is that there are three different alphabets plus the western letters in use, which makes it possible to write the same word with the exact same meanings and pronunciations using different symbols, a phenomena that occurs in English only for numbers. It is correct and common to mix alphabets in sentences and use different spellings of the same word (example: there are at least six common ways to spell the Japanese word for 'I', meaning myself, some of them with the exact same pronunciation). This makes the evaluation of Japanese using a word error rate, which is based on word-IDs, more difficult than in Western languages, because different word-IDs shouldn't be counted as errors if their meanings and pronunciations are exactly the same. These errors are here referred to as *type II* errors. For the experiments of this paper some of the results were cleaned of type I and type II errors to show their relevance.

A third problem specific to decoding is, that because of the many short words and the many homonyms the number of found word-ends, which make stack operations and N-gram accesses necessary, is higher than for example in English. The many short words resulting in on average more word boundaries increase also the need for cross-word modeling.

3.2 Recognition results for high accuracy

Table I shows the results, for which the parameter settings in Table II were optimized to reach a low word error rate. The acoustic models are monophones with 129 states and triphones with 2000 and 3000 states with 16 mixture components each. The experiments of this task were run in two modes, a *Katakana* mode, where all word-IDs and all transcriptions are written only in the Katakana alphabet, and in a *Kanji* mode, where all word-IDs and transcriptions are written in a mixture of the three alphabets like they occur in a regular newspaper. Best recognition results in Kanji recognition mode are 5.2% word error rate (WER) for the male speakers using 3000-state models and 4.8% WER for the female speakers using 2000-state models, if the results are cleaned from errors that shouldn't be counted as errors in Japanese like discussed above. The raw outputs from the recognizer are about 15% relative (1% absolute) worse, showing that these errors, which are specific to Japanese, shouldn't be neglected. The Katakana results, which hide misrecognition of homonyms occurring in Japanese more frequently than for example in English, overestimate the score of interest by about 1% absolute on average.

The parameter settings in Table II show that the word-end-beam can be chosen lower than the word-within-beam like discussed in section 2.1.2. The average number of competing HMM model nodes at any time determines to a large extent the overall speed of the search and is a suitable measure to compare different implementations of stack decoders. Note that because active nodes cannot be merged, this number generally will be

Table I: Recognition results for high accuracy

states x *mixtures*	cross-word models	MALE Kat/Kan	FEMALE Kat/Kan
129 x 16 (cleaned)	no	88.7/87.5	91.8/90.8
2000 x 16 (cleaned)	yes	95.2/93.3	**96.9/95.2**
3000 x 16 (cleaned)	yes	**96.4/94.8**	95.9/94.5
129 x 16 (not cleaned)	no	87.9/86.7	91.0/90.0
2000 x 16 (not cleaned)	yes	94.4/92.6	**96.1/94.4**
3000 x 16 (not cleaned)	yes	**95.6/94.0**	95.0/93.6

The upper part shows results which were cleaned of errors that shouldn't be counted in Japanese, the lower results weren't cleaned. All results are given for the Katakana (Kat) and the Kanji (Kan) recognition mode as discussed in the text.

lower in transition network decoders. The stack statistics show that on average about 75% of the time the language model state of the hypothesis to be inserted is already on the stack, and only 5-10% of the hypotheses remain in the beam to get actually expanded. This implies that at least the number of N-gram accesses could be reduced drastically by a completely time-synchronous scheme, where word- and state-level search both run time-synchronously, which hasn't been tried here.

The average number of N-gram accesses including all back-offs compared to the number of cache accesses within the N-gram module show that many N-grams are used more than once and a cache will be very useful in cases where the N-gram access is slow like for a disk-based LM.

Table II: Parameter settings and average search statistics for results from Table I

	129 x 16	2000 x 16	3000 x 16
word-end-beam	30	50	50
word-within-beam	40	80	80
LM-scale	6	11	12
word-deletion-penalty	0	0	0
realtime factor (RTF)	8.4	24	25
active model nodes/frame	2213	10045	8324
pushed hyps/frame	1215	1196	1113
inserted/replaced hyps/frame	243/972	246/950	211/902
extended hyps/frame (average stacksize)	41	25	20
on-demand N-gram smearing	no	yes	yes
N-gram accesses/frame	27519	21029	18749
cache accesses/frame	27268	20834	18600

These results are based on 25-dimensional feature vectors, all log-likelihoods base 10, the realtime factor is for 300 Mhz Pentium II and includes observation likelihood calculation. All results in this table are averaged over genders.

3.3 Recognition results for high speed and low memory

Table III and Table IV show results and parameter settings for experiments that were run to maximize decoding speed at a low (about 1%) search error and minimize memory requirements, with (a) a regular memory-based trigram LM and (b) a disk-based LM. Almost realtime performance including all observation likelihood calculations is possible with around 90% recognition rate using between 10 and 20 MB of memory. The disk-based LM slows down the search by about a factor of three.

The trigram LM has 5k unigrams, 330k bigrams and 720k trigrams, occupying in total about 6 MB of memory using the techniques of 2.4. An N-gram entry occupies on average 6 bytes, if the complete LM is held in memory, and about 100 kB total for the disk-based LM with bigrams and trigrams on disk which are loaded on demand and cached in a hash table of limited size.

Table III: Results for high speed and low memory

states x *mixt.*	disk-LM	cross-word models	MALE Kat/Kan	FEMALE Kat/Kan	MEMORY	RTF
129 x 16	no	no	87.0/86.0	90.2/89.2	10 MB	1.3
129 x 16	yes	no	87.0/86.0	90.2/89.2	4 MB	3.9
2000 x 16	no	yes	93.3/91.5	95.0/93.8	20 MB	9
2000 x 16	yes	yes	93.3/91.5	95.0/93.8	14 MB	14

Results with parameter settings optimized for high speed and low memory, not cleaned of type I/II errors. Memory and realtime factor are for a 300 MHz Pentium II.

Table IV: Parameter settings and average search statistics for results from Table III

	129 x 16	2000 x 16
word-end-beam	20	40
word-within-beam	30	70
LM-scale	6	11
word-deletion-penalty	0	0
active model nodes/frame	685	2993
pushed hyps/frame	149	408
inserted/replaced) hyps/frame	44/105	97/311
extended hyps/frame (average stacksize)	7.9	12.3
on-demand N-gram smearing	no	yes
N-gram accesses/frame	2927	8196
cache accesses/frame	2882	8114

3.4 Time and memory requirements for modules

The relative time and memory requirements of the different modules are summarized in Table V. Most of the time is spent on the likelihood calculation and the state-level search, which includes all operations for the active node list. The time for the tree lexicon includes activating and deactivating HMM nodes. The cross-word rescoring procedure includes the on-demand lookup for the correct cross-word HMM model and the local Viterbi search as its most time-consuming parts. The LM state comparison is included in the time listed for the stack operations, which is surprisingly low given the simple linear list implementation shown in section 2.2.

Memory requirements are listed for a 5000 word vocabulary with on average 1.5 pronunciations each, giving about 200 bytes/entry. The acoustic model takes most of the memory because of its uncompressed 4-byte mean/variance parameters and the cache for the likelihood calculation. The hypotheses generation itself takes almost no memory but what is needed to represent the currently active hyp-nodes and arcs, which are in the case of first-best recognition not more than a few hundred. Similar, the stack module contains mainly pointers to hyp-nodes, which also don't use more than a few kB.

Table V: Relative time and memory requirements for modules

MODULE	RELATIVE TIME	MEMORY
stack	2%	≈ 0
hyp	1%	≈ 0
state-level search	33%	0.5 MB
word-level search	3%	≈ 0
tree lexicon	5%	1.4 MB
N-gram	12%	5.1 MB
acoustic model	31%	13.0 MB
cross-word rescoring	11%	–
SUM	100%	20 MB

Relative time and memory requirements split up for modules using the 2000 x 16 acoustic model from Table III.

3.5 Usage of cross-word models

Given the word definition for Japanese, which was used for this paper, the use of cross-word models is essential for the recognition of read newspaper sentences, as Table VI shows. The additional search time for the local rescoring using cross-word models doesn't effect the overall search time at all for this experiment, possibly because of more accurate partial hypotheses at any time during the search.

3.6 Usage of fast-match models

Table VII shows the effect of using fast-match models to find acoustically likely word hypotheses quickly like described in section 2.7. In the case tested here their use required fine tuning of several search parameters to make a difference in recognition time.

Table VI: Effect of cross-word effects

CROSS-WORD MODELS	RTF	REC-RATE
yes	24	93.5
no	24	87.0

Recognition of Japanese newspaper articles with and without cross-word 2000x16 models using the same beam settings, but optimized LM-scales and word-deletion penalties. Results are averaged over genders in Kanji recognition mode with search parameters of Table II, not cleaned.

Table VII: Effect of fast-match models

FAST-MATCH MODELS	RTF	REC-RATE
yes	7	92.5
no	9	92.6

Use of fast-match models to find acoustically likely word hypotheses quickly, averaged over genders in Kanji recognition mode with search parameters of Table IV, not cleaned. Fast-match models were 3-state monophones with four mixture components each.

3.7 Effect of on-demand N-gram smearing

On-demand N-gram smearing (section 2.5.2) can efficiently reduce the number of active model nodes, as Table VIII shows. In the cases tested here the reduction of active nodes does not necessarily reduce the search time because of the overhead of the procedure that has to be invoked before each stack is expanded. If the likelihood calculation of the acoustic models would take longer, this method would have a greater effect on the total recognition time.

Table VIII: Effect of on-demand N-gram smearing

states x *mixt.*	LM lookahead	active models	N-gram accesses	RTF	REC. RATE
129 x 16	unigram	685	2927	1.3	87.6
129 x 16	N-gram	593	2252	1.9	87.5
2000 x 16	unigram	2993	8196	10	92.6
2000 x 16	N-gram	2817	5486	9	92.6

Shows the effect of on-demand N-gram smearing versus unigram smearing. Results are averaged over genders in Kanji recognition mode with search parameters of Table IV, not cleaned.

3.8 Lattice/N-best list generation and lattice rescoring

The results shown in Table IX compare the time and memory requirements for generating the first-best hypothesis with the time for generating lattices or N-best lists in the first pass. It can be seen that the more complicated LM state check for the N-best lists creates only little overhead, and is almost independent of the length of the N-best lists.

Lattice rescoring as discussed in section 2.9 was tested for the generated lattices for both lattice resoring modes. Type I lattice rescoring refers to using only the word-graph as an LM constraint, but all alignments, acoustic scores including cross-word effects and LM scores are recalculated. For type II lattice rescoring only the LM scores are recalculated, which usually includes a new LM scale factor and a new word deletion penalty.

Table IX: Relative time and memory for different search modes

SEARCH MODE	RTF	MEMORY
first-best (absolut)	9	20 MB
first-best	100%	100%
lattice	107%	106%
N-best list, N = 10	113%	100.4%
N-best list, N = 50	116%	100.4%
N-best list, N = 100	117%	100.5%
lattice rescoring type I	0.1%	77%
lattice rescoring type II	5.6%	93%

Relative time and memory (as measured by the UNIX `top` command) for several search modes with beams leading to lattices of about 2500 arcs and 500 hyp-nodes, and an average N-best list length of 90 hypotheses, for parameter settings as in Table IV. All N-best hypotheses differ by at least one word like defined in section 2.2.2.

4 CONCLUSIONS

This paper presented a detailed description of a memory-efficient one-pass stack decoder applied to recognition of sentences from a Japanese newspaper. The architecture of the time-asynchronous stack decoder made it easily possible to integrate lattice and N-best list construction as well as arbitrary order N-gram LMs and arbitrary order cross-word context-dependent acoustic models in a single decoder. Also, various forms of lattice rescoring and the generation of forced alignments fits well into the framework of the time-asynchronous search technique. Memory requirements at around 1% search error are between 4 and 20 MB using the techniques from the paper.

In summary, it can be concluded, that a time-asynchronous stack decoder is a conceptually attractive framework for integrating many often needed procedures for speech recognition tasks. Although very efficient in memory and faster than the decoder mentioned in (Kawahara et al, 1998) for the same task, it should be noted that the speed of a time-asynchronous stack decoder like implemented here is probably not optimal for the specific task of generating a first-best hypothesis or a lattice from a feature vector sequence, because the globally time-asynchronous search over the state space results in

the generation of many partial hypotheses that are later not expanded. This could be avoided by using a time-synchronous stack decoder with multiple trees, which hasn't been tried here.

5 ACKNOWLEDGMENTS

The author would like to thank the Japanese IPA group (Kawahara et al, 1998) for their supply of acoustic models, initial language models and the initial pronunciation dictionary, and especially Prof. Shikano from the Nara Institute of Technology, Japan, for pointing out the need for cross-word models for Japanese. Without the many fruitful discussions concerning stack decoders with Christoph Neukirchen and Daniel Willett from the Gerhard-Mercator University, Duisburg, Germany, the implementation of the on-demand N-gram smearing procedure wouldn't have been possible.

References

[1] Alleva F., Huang X. & Hwang M. (1996). Improvements on the pronunciation prefix tree search organization. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Atlanta, GA, Vol. I, pp. 133-136.

[2] Alleva F. (1997). Search Organization in the Whisper Continuous Speech Recognition System. Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding. Santa Barbara, CA, pp. 295-302.

[3] Aubert X., Dugast C., Ney H. & Steinbiss V. (1994). Large vocabulary continuous speech recognition of Wall Street Journal Data. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Adelaide, Australia, Vol II, pp. 129-132.

[4] Bahl L.R., de Souza P.V., Gopalakrishnan P.S., Nahamoo D., Picheny M. (1992). A fast match for continuous speech recognition using allophonic models. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, San Francisco, CA, Vol I, pp. I-17 - I-20.

[5] Bahl L.R., de Souza P.V., Gopalakrishnan P.S., Nahamoo D. & Picheny M. (1993). Word lookahead scheme for cross-word right context models in a stack decoder. Proceedings of the European Conference on Speech Communication and Technology, Berlin, Germany, Vol. II, pp. 851-854.

[6] Beyerlein P. & Ullrich M. (1995). Hamming distance approximation for a fast log-likelihood compuation for mixture densities. Proceedings of the European Conference on Speech Communication and Technology, Madrid, Spain, Vol. II, pp. 1083-1086.

[7] Bishop C.M. (1995). Neural Networks for Pattern Recognition. Clarendon Press, Oxford.

[8] Duda R.O & Hart P.E. (1974). Pattern Classification and Scene Analysis. New York: John Wiley & Sons.

[9] Gauvain J.L., Lamel L.F., Adda G. & Adda-Decker M. (1994). The LIMSI Continuous Speech Ditation System: Evaluation on the ARPA Wall Street Journal Task. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Adelaide, Australia, Vol I, pp. 557-560.

[10] Gopalakrishnan P.S. (1995). A tree search strategy for large vocabulary continuous speech recognition. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Detroit, MI, Vol I, pp. 572-575.

[11] Gopalakrishnan P.S., & Bahl L.R. (1996). Fast matching techniques. Automatic Speech and Speaker Recognition, Advanced Topics (Chin-Hui Lee, Frank K. Soong & Kuldip K. Paliwal eds.) pp. 413-428. Kluwer Academic Publishers, Boston.

[12] Hetherington I.L., Phillips M.S., Glass J.R., Zue V.W. (1993). A^* Word Network Search for Continuous Speech Recognition. Proceedings of the European Conference on Speech Communication and Technology, Berlin, Germany, Vol. III, pp. 1533-1537.

[13] Huang X.D., Ariki Y., Jack M.A. (1990). Hidden Markov Models for Speech Recognition. Edinburgh University Press, Edinburgh.

[14] Kawahara T., Kobayashi T., Takeda K., Minematsu N., Itou K., Yamamoto M., Utsuro T. & Shikano K. (1998). Sharable Software Repository for {Japanese} Large Vocabulary Continuous Speech Recognition. Proceedings of the International Conference on Spoken Language Processing, Sidney, Australia. to appear.

[15] Murveit H., Butzberger J., Digalakis V., & Weintraub M. (1993). Large vocabulary dictation using SRI's Decipher^{TM} Speech Recognition System: Progressive Search Techniques. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Minneapolis, MN, Vol II, pp. 319-322.

[16] Neukirchen C., Willet D. (1997). Gerhard-Mercator University Duisburg, Germany, personal communication.

[17] Ney H. (1993), Search Strategies For Large-Vocabulary-Continuous-Speech Recognition. NATO Advanced Studies Institute, Bubion, Spain, June-July 1993. Speech Recognition and Coding – New Advances and Trends (A.J Rubio Ayuso & J.M. Lopez Soler, eds.), pp. 210-225. Springer, Berlin.

[18] Ney H. & Aubert. X. (1996). Dynamic Programming Search: From Digit Strings to Large Vocabulary Speech Recognition. Automatic Speech and Speaker Recognition, Advanced Topics (Chin-Hui Lee, Frank K. Soong & Kuldip K. Paliwal eds.) pp. 385-412. Kluwer Academic Publishers, Boston.

[19] Ney H. & Ortmanns S. (1997). Progress in Dynamic Programming Search for LVCSR. Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding. Santa Barbara, CA, pp. 287-294.

[20] Nilsson N.J., (1971). Problem Solving Methods of Artificial Intelligence. McGraw Hill, New York.

[21] Odell J.J. (1995). The Use of Context in Large Vocabulary Speech Recognition. Doctor Thesis, Cambridge University, Cambridge, England.

[22] Ortmanns S., Ney H., Aubert X. (1997). A word graph algorithm for large vocabulary continuous speech recognition. Computer, Speech and Language, Vol. 11, pp. 43-72.

[23] Paul D. (1991). Algorithms for an Optimal A^* Search and Linearizing the Search in the Stack Decoder. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Toronto, Canada, Vol I, pp. 693-696.

[24] Paul D. (1992). An Efficient A^* Stack Decoder Algorithm for Continuous Speech Recognition with a Stochastic Language Model. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, San Francisco, CA, Vol I, pp. I-25 - I-28.

[25] Rabiner L. & Juang B.H. (1993). Fundamentals of Speech Recognition. Prentice-Hall, New Jersey.

[26] Ravishankar M.K. (1996). Efficient Algorithms for Speech Recognition. Doctor Thesis, Technical Report CMU-CS-96-143, Pittsburgh.

[27] Renals S. & Hochberg M. (1995a). Decoder technology for connectionist large vocabulary speech recognition. Technical Report CUED/F-INGENG/TR186, Cambridge University Engineering Department, Cambridge, England.

[28] Renals S. & Hochberg M. (1995b) Efficient search using posterior phone probability estimates. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Detroit, MI, Vol. I, pp. 596-599.

[29] Renals S. & Hochberg M. (1996) Efficient evaluation of the search space using the NOWAY decoder. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Atlanta, GA, Vol. I, pp. 149-153.

[30] Robinson T. & Christie J. (1998). Time-first search for large vocabulary speech recognition. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, WA, Vol. II, pp. 829-833.

[31] Schwartz R., Nguyen L. & Makhoul J. (1996). Multiple-pass search strategies. Automatic Speech and Speaker Recognition, Advanced Topics (Chin-Hui Lee, Frank K. Soong & Kuldip K. Paliwal eds.) pp. 429-456. Kluwer Academic Publishers, Boston.

[32] Soong F.K. & Huang E.F. (1991). A tree-trellis based fast search for finding the N-best sentence hypotheses in continuous speech recognition. Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Toronto, Canada, Vol I, pp. 705-708.

[33] Shimizu T., Yamamoto H., Masataki H., Matsunaga S. & Sagisaka Y. (1996). Reduction of Number of Word Hypotheses for Large Vocabulary Continuous Speech Recognition (in Japanese). IEICE Transactions, Vol. J79-D-II, No.12, pp. 2117-2124.

[34] Steinbiss V., Tran B. & Ney H. (1994). Improvements in Beam Search. Proceedings of the International Conference on Spoken Language Processing, Yokohama, Japan. pp. S36-5.1 - S36-5.4.

[35] Young S., Jansen J., Odell J., Ollason D. & Woodland P. (1997). The HTK Book (Version 2.1). Distributed with the HTK toolkit.