TR - H - 194

Dynamic Programming for the Prototype-Based Minimum Error Classifier

Erik McDermott

1996. 5. 23

ATR人間情報通信研究所

〒619-02　京都府相楽郡精華町光台2-2　☎ 0774-95-1011

ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Telephone: +81-774-95-1011
Facsimile: +81-774-95-1008

Dynamic Programming for the Prototype-Based Minimum Error Classifier

Erik McDermott

ATR Human Information Processing Research Laboratories,

Hikari-dai 2-2, Seika-cho, Kyoto 619-02, Japan

Abstract

Currently, most speech recognition technology involves some form of Dynamic Programming (DP) to cope with the non-linear compressions and expansions of speech event durations. In choosing the particular version of DP now used in the Prototype-Based Minimum Error Classifier (PBMEC) (McDermott and Katagiri, 1992, 1993, 1994), we aimed to use an approach that was conventional, yet which also incorporated some of the latest advances in the fast evolving field of search.

In this report we describe the use of one-pass DP (Sakoe 1978, Ney 1984), histogram based pruning (Ney et al., 1994) and A* based N-best search (Soong & Huang, 1991, 1994), in the context of PBMEC. Our primary goal in this report is to explain these different methods and their use in PBMEC. The motivation for choosing this particular configuration comes from considerations of speed and search accuracy. In addition, we contrast two different approaches to speeding up the search. The first, proposed by Soong & Huang (1994), consists in using a simple forward phase with a reduced grammar, followed by a detailed backward A* search using the full task grammar. The second is a simple time-synchronous pruning method proposed by Ney et al. (1994). Our results suggest that the latter method is more effective in reducing search time for the tasks we examined.

The recognizer used throughout this study is the Prototype Based Minimum Error Classifier.

Contents

List of Figures

1 Introduction

The application of Dynamic Programming (DP) to speech recognition has enjoyed widespread use since it was proposed [39] [34]. The purpose of the technique, in the context of speech recognition, is usually to generate the most likely path through an array of local matches between a given acoustic model and individual speech frames from an unknown utterance, in an efficient manner. For a known model sequence (e.g. it is known that the utterance should be modelled by the acoustic models for a particular word or sentence), DP provides a solution to the time-alignment problem of how to assign the speech frames in the utterance to acoustic models in the model sequence. When the model sequence is not known, (for example, if the utterance could be any of a large set of words or sentences), DP allows one to compare different model sequence assignments and find the best model sequence, e.g. the best word or sentence model for the utterance. DP does not directly address the additional question of finding not just the best string, but the top N-best strings; however, several schemes have been proposed to do so [44] [42] [48]. Another key issue in the use of DP is speeding up the search when large, complex grammars are used. Typically, some form of pruning is used.

Our primary goal in this report is to describe the particular techniques adopted in the Prototype Based Minimum Error Classifier [32] system developed at ATR Human Information Processing Laboratories. In addition, we contrast two approaches to speeding up the search: 1) pruning the forward search, i.e. limiting the search to a dynamically changing subset of the finite state machine representing the task grammar, and 2) using a simpler grammar (i.e. a smaller finite state machine) in the forward search (with no pruning), followed by A* based search using the full set of grammatical constraints for the task, in the backward search. Based on results for the ATR HIP Telephone Task, which involves the recognition of names spoken in isolation, we tentatively conclude that the first approach is simpler and more efficient than the second.

2 Outline of search module configuration

Before presenting a detailed description, we outline the system configuration and present our motivation in choosing this particular configuration. The PBMEC system's search module is an implementation of the following algorithms:

Forward search: time-synchronous one-pass DP

Time-synchronous search appears to offer more flexible and efficient pruning strategies than phoneme-synchronous search; furthermore, "immortal node" trace-back can be used to generate partial recognition results that are guaranteed to be part of the final recognition result, before the end of the utterance is reached.

Histogram-based pruning

This pruning method [35] is a cheap and simple way of maintaining a constant number of active DP grid points (alternatively, "DP states") during the forward search. This is in contrast to pruning methods which allow the number of active states to grow larger or smaller depending on the "difficulty" of the search. The latter approach has some appeal, but the required computational effort is utterance dependent; the pruning parameters are thus somewhat harder to tune. In the following we present a comparison of these two approaches.

Backward search: phoneme/word synchronous A* based N-best search

Of the many N-best algorithms proposed, Soong & Huang's [44] [13] [14] is one of the few that is guaranteed to generate the true N-best strings - assuming a full search was used in the forward phase. It is also not expensive computationally. We will see in the following that the A* based backward search offers an interesting alternative to pruning for speeding up the forward search.

3 Forward search, backward search, CPU time, pruning and optimality

A key variable in the tandem of forward and backward (N-best) search will be the grammatical constraints used in each phase. By grammatical constraints, we mean the allowed model sequences, as specified (in our implementation) by a finite state machine representing a grammar for the task. In this light, we now outline the computational requirements of the search procedure, ways in which the search can be speeded up, and how this may affect optimality.

(It should be clear that the word "optimal" in the context of DP-based search describes a search that yields the string (acoustic model sequence) (or N strings) with the best (or N-best) accumulated path distance(s) through the DP grid. This notion of optimality is (unfortunately) not directly related to the correctness of the recognition output; the recognized string could be optimal in this sense, yet incorrect.)

3.1 CPU time

Forward search

The time-synchronous DP algorithm we use in the forward phase can be outlined as a loop over frames of speech, and for each frame, a loop over states of the finite state machine. Thus, the complexity of the grammar, and ensuing size of the finite state machine, typically has a linear effect on search time of the forward phase if no pruning is used.

Backward search

The A* based backward phase uses a stack of partial string hypotheses to direct its search. The search time of this phase will depend on how well the hypotheses can be evaluated. We will see that if the same grammatical constraints are used for forward and backward phases, the evaluation function will be exact; i.e., for a given partial string hypothesis, the evaluation function can tell us the score of the best possible way of completing that particular partial hypothesis. This allows an optimally efficient search with no backtracking. In this scenario, the CPU time required for the backward search is typically much smaller than that required by the forward search. (This is still the case if pruning was used in the forward phase - though the guarantee of optimality is lost.)

On the other hand, if the A* phase uses a different set of constraints than the forward phase - say, a more complex set of constraints - then the evaluation of partial hypotheses is no longer exact, and a considerable amount of backtracking may be necessary. The closer the forward constraints are to the backward constraints, the less backtracking is necessary.

3.2 Accelerating the search while preserving optimality?

Pruning the forward search

As the backward search, which uses scores computed in the forward phase, is typically much faster than the forward search, the most obvious approach to speeding up the search is to prune the forward search. This can be achieved by limiting the states searched at each frame to a subset of "active" states, rather than considering the whole set of states in the finite state machine. The histogram-based method we adopted is one way of mainting such a "beam" of active states. Of course, pruning comes at a cost: pruning too heavily will harm optimality. Though the degree of pruning can be tuned, the optimality of the search can no longer be guaranteed. The reason for this is that the true best path may start out with a poor score to a particular state in the beam, which may be pruned out of the search.

Using a simpler grammar in the forward search

An alternative approach is to use a simpler set of constraints in the forward phase (without pruning), and then to perform a backward A* based N-best search using the full set of grammatical constraints for the target task. Since the grammar used in the forward phase is much simpler than before, the forward search is much faster. Furthermore, when certain conditions are met, the A* algorithm is able to use the scores obtained from the reduced forward search to guide the backward search effectively, *while preserving the guarantee of optimality.* However, as mentioned above, since the scores from the forward search are based on a simpler grammar, the backward phase is not as efficient as before, and will usually require more computational effort. It is not clear exactly how much more time will be required for the backward phase. (If the total resulting search time becomes unmanageable, pruning of the A* stack may be desirable. This, however, may seriously impact optimality.)

The concept of simple forward search followed by detailed backward search is described in [2], [13] and [14].

The question we address here is whether this tandem of forward search with a simple grammar followed by backward search with the full grammar is more efficient than using the full grammar in both forward and backward phases, but with pruning in the forward phase.

In the following, we review the PBMEC recognizer assumed here, before launching into a detailed description of time-synchronous DP, the A* algorithm for N-best string search, and experiments investigating the two different approaches described above.

4 The Recognizer

The recognizer used here is the Prototype Based Minimum Error Classifier (PBMEC) system described in [30] [32]. A discriminant function for each category to be recognized (e.g. a phoneme, word or sentence) is defined in terms of a DP procedure to link reference vector based phoneme models together according to the grammar of the task at hand.

Local distance

At the lowest level, the phoneme models are taken to consist of a connected sequence of sub-phonemic states, illustrated in Figure 1. Each state is assigned a number of reference vectors, analogous to the mean vectors used in a continuous hidden Markov model. These are used to generate an L_p norm-based *state distance* $e(\mathbf{x}_t, s)$, which is a function of a single feature vector $\mathbf{x}_t$ at time t (from an utterance $\mathbf{x}_1^T = (\mathbf{x}_1, ..., \mathbf{x}_t, ..., \mathbf{x}_T)$) and reference vectors belonging to the state s:

$$e(\mathbf{x}_t, s) = \left[\sum_{i=1}^{I_s} [(\mathbf{x}_t - \mathbf{r}_i^s)^t (\Sigma_i^s)^{-1} (\mathbf{x}_t - \mathbf{r}_i^s)]^{-\zeta} \right]^{-\frac{1}{\zeta}}, \tag{1}$$

where $\mathbf{r}_i^s$ denotes the (adaptable) i-th reference vector of state s, Σ_i^s is an adaptable positive definite "weighting" matrix corresponding to $\mathbf{r}_i^s$, and I_s is the number of the reference vectors assigned to s. For a large ζ, the state distance becomes the distance to the closest reference vector, and each state can then be seen to correspond to a category in a Learning Vector Quantization classifier [20] [30].

Discriminant functions for phoneme/word/phrase categories

A matrix of distances $D_{j,T,S}$ is defined to be a matrix where each position (t, s) contains $e(\mathbf{x}_t, s)$ for the states of category j. The discriminant function for each phoneme/word/phrase j can then be defined as:

$$g_j(\mathbf{x}_1^T) = \left[\sum_{\theta} [V_\theta(D_{j,T,S})]^{-\xi} \right]^{\frac{1}{-\xi}} \tag{2}$$

where $V_\theta(D_{j,T,S})$ represents an accumulated sum, or *path distance*, along a possible DP path θ through a region of $D_{j,T,S}$, and where S is the total number of states in category j. The decision rule here will be to choose the category with the smallest discriminant function value:

$$\text{decide } C_j \text{ if } g_j(\mathbf{x}_1^T) < g_k(\mathbf{x}_1^T) \text{ for all } k \neq j. \tag{3}$$

In previous work [30] [31] [32] we have described in detail how the Minimum Classification Error / Generalized Probabilistic Descent framework can be used to train the above classifier. Here, however, our focus is purely on the computational aspects of the DP calculation that is necessary to find the best sequence of model states for a given, possibly unknown, utterance.

In the following we refer to our implementation of PBMEC as the "*pbmec* program". This is the main recognizer program, that loads model files and grammar files, and can be used

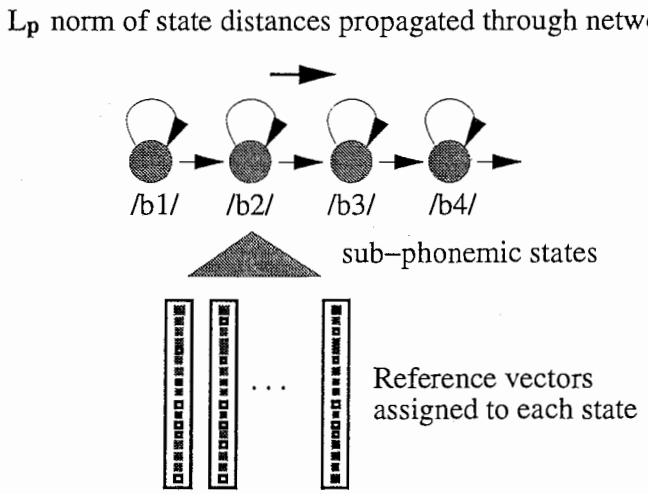

Figure 1: PBMEC at the finest grain.

for recognition of utterances or MCE/GPD training over a set of training utterances. In the ATR HIP computing environment, typing /usr/hearing/recog/bin/$OS/pbmec by itself will generate an extensive help message.

5 One pass DP through Finite State Machine

The grammar for any given task is represented as a tree structured finite state machine (FSM). This is illustated in Figure 5 for a grammar of names of ATR HIP Laboratories researchers . The finite state machine representation is well suited for use in a DP framework, as the DP grid can be viewed as a set of connected states in a directed graph. The DP calculation is guaranteed to find the lowest cost path through the graph, from start state to end state. Note that this representation is also consistent with the classifier representation. The PBMEC classifier states map directly onto states in the FSM. Thus, a state occuring in a particular instance of the vowel /a/ in the FSM represents a part of the grammatical/lexical constraints as well as a particular set of acoustic model parameters. If multiple states, or "sub-states" are used to represent the fine temporal structure of a phoneme, they are linked serially within that phoneme, with each sub-state usually able to loop onto itself, as shown in 1.

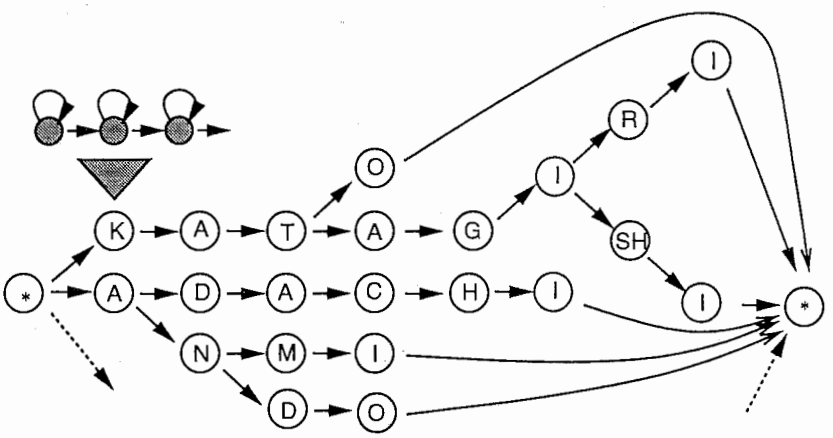

Figure 2: Finite state machine representing a vocabulary of ATR researchers' names.

DP [39] [34] is the search for the shortest path through a FSM (such as the one shown in 5) given a set of local acoustic scores (representing the match between a frame of speech and all the states in the classifier) in time. It is convenient to visualize the calculation in terms of a score grid, with time going left to right from the beginning to the end of the utterance, and with the model states represented vertically, from the first to the last state (see Figure 5).

In our use of DP, a path through the grid must start at the first frame and first state (i.e. the bottom left of the grid) and end at the last frame and last state ("fixed end-points"). The other constraints on what paths are legal are defined by the connectivity of the FSM.

Given that a path is a sequence of state-frame assignments in time, the overall path score is the sum of the local acoustic scores corresponding to that sequence. The DP method for finding the shortest such path is to apply the following rule to each grid point, moving over

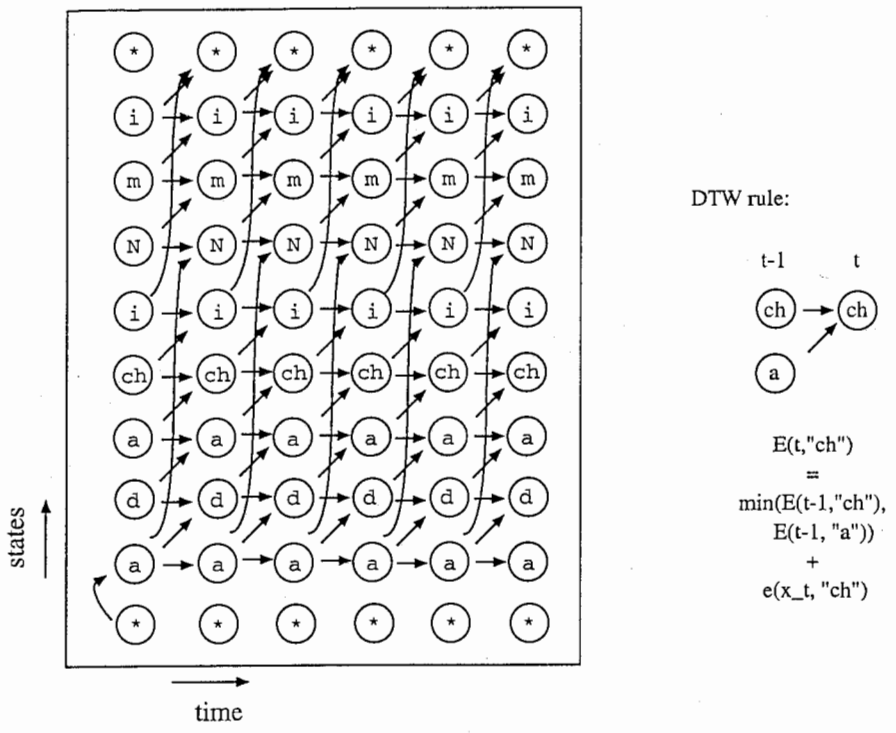

Figure 3: DP grid corresponding to a part of the finite state machine shown above. The best DP path scores are propagated in time through the FSM.

the length of the utterance $\mathbf{x}_1^T$ from beginning to end:

$$E_{(t,s)}(\mathbf{x}_1^T) = e(\mathbf{x}_t, s) + \min_{r \in R_s} E_{t-1,r}(\mathbf{x}_1^T) \tag{4}$$

where R_s is the set of all states that may legally precede state s according to the grammar. Application of this rule ensures that at each grid position, the cumulative distance $E_{t,s}(\mathbf{x}_1^T)$ corresponds to the distance of the shortest path to that grid position. The distance of the shortest path for the whole utterance is thus contained in $E_{T,S}(\mathbf{x}_1^T)$, i.e. in the grid position of the last frame and last state.

The overall DP computation can thus be outlined as the following procedure:

loop over frames x_t in utterance
    loop over all phonetic states p in FSM
        loop over *pbmec* sub-states s of phoneme p
            calculate $E_{(t,s)}(\mathbf{x}_1^T) = e(\mathbf{x}_t, s) + \min_{r \in R_s} E_{t-1,r}(\mathbf{x}_1^T)$
        end
    end
end

Using R to denote the average number of legal predecessors R_s for each sub-state, the computational effort required in this computation is of order $O(TSR)$, i.e. the number of frames

in the utterance, times the number of states in the finite state machine, times the average number of predecessors to each state. The latter number will vary with the FSM topology. (The FSMs we use are all left-right tree-structured within grammatical node expansions, so all states within an expansion (e.g. between "*" nodes in fig 5) have just two predecessors: themselves (via a self loop), and a single up-stream state. The self loops are not represented in the FSM description, but are assumed by the *pbmec* program, for all phonetic states. At grammatical nodes, such as the right-most "*" node in Figure 5, the number of predecessors may be large.)

6 Generating a finite state machine description file from a grammar description file

The practice so far has been for the *pbmec* program to read an ASCII file that explicitly describes a finite state machine. A separate program generates this FSM description file from a regular grammar description file. The grammar description language was inspired by HTK's and is very similar to HTK's in expressive power. We describe the generation and structure of FSMs to illustrate more concretely the discussion of DP and A* search in the following.

An example of a grammar description file follows.

```
mcd@vesuvius: cat foo.rgra

/* To convert into a finite state machine descriptor (.fsm) file,
   use new_fsmify <grammar file name stem>. */

( <root> = silence <n> silence )

( <n> =  o n e | t w o | t h r e e )
```

This can be converted to a FSM description file as follows:

```
mcd@vesuvius:
mcd@vesuvius: new_fsmify foo
Got definitions in foo.rgra.
Fsmifying definitions.
Expanding <root_>
Expanding <n_>
Printing FSM to file foo.fsm
Done.
mcd@vesuvius:
```

```
mcd@vesuvius: cat foo.fsm
Number of states = 20

state 0 ( <root_> ) num_pred 0 pred ( ) num_succ 1 succ ( 2 )
state 1 ( <_root> ) num_pred 1 pred ( 19 ) num_succ 0 succ ( )
state 2 ( silence ) num_pred 1 pred ( 0 ) num_succ 1 succ ( 3 )
state 3 ( <n_> ) num_pred 1 pred ( 2 ) num_succ 2 succ ( 5 9 )
state 4 ( <_n> ) num_pred 3 pred ( 8 12 17 ) num_succ 1 succ ( 18 )
state 5 ( o ) num_pred 1 pred ( 3 ) num_succ 1 succ ( 6 )
state 6 ( n ) num_pred 1 pred ( 5 ) num_succ 1 succ ( 7 )
state 7 ( e ) num_pred 1 pred ( 6 ) num_succ 1 succ ( 8 )
state 8 ( @ ) num_pred 1 pred ( 7 ) num_succ 1 succ ( 4 )
state 9 ( t ) num_pred 1 pred ( 3 ) num_succ 2 succ ( 10 13 )
state 10 ( w ) num_pred 1 pred ( 9 ) num_succ 1 succ ( 11 )
state 11 ( o ) num_pred 1 pred ( 10 ) num_succ 1 succ ( 12 )
state 12 ( @ ) num_pred 1 pred ( 11 ) num_succ 1 succ ( 4 )
state 13 ( h ) num_pred 1 pred ( 9 ) num_succ 1 succ ( 14 )
state 14 ( r ) num_pred 1 pred ( 13 ) num_succ 1 succ ( 15 )
state 15 ( e ) num_pred 1 pred ( 14 ) num_succ 1 succ ( 16 )
state 16 ( e ) num_pred 1 pred ( 15 ) num_succ 1 succ ( 17 )
state 17 ( @ ) num_pred 1 pred ( 16 ) num_succ 1 succ ( 4 )
state 18 ( silence ) num_pred 1 pred ( 4 ) num_succ 1 succ ( 19 )
state 19 ( @ ) num_pred 1 pred ( 18 ) num_succ 1 succ ( 1 )
mcd@vesuvius:
mcd@vesuvius:
mcd@vesuvius:
```

The *pbmec* program will treat FSM states as either 1) phonetic states that are associated with a set of PBMEC parameters (i.e. reference and weight vectors) necessary to calculate the state distance defined above 1, or 2) grammatical states that have no PBMEC parameters, and are merely used to connect the finite state machine and collect DP scores. The grammatical symbols typically correspond to the beginning and end of the expansion of a grammatical category, such as for "<n>" above: the symbol for state 3 is "<n_>", beginning the expansion for "<n>" (marked by the underbar to the right), and the symbol for state 4 is "<_n>", the end of the expansion (marked by the underbar to the left). The symbol "@" is a grammatical symbol inserted at the end of a sequence of phonetic states to tell the search module in the *pbmec* program to remember the DP exit scores along that arc. Note that the FSMs generated by new_fsmify are left-right tree-structured, i.e. symbols that are shared by the initial portions of alternative definitions are represented by the same states in the FSM. (This tree-structuring is perhaps the only significant difference in functionality between the FSM representation/generation we use and that used in HTK).

The program /usr/hearing/recog/bin/SunOS/new_fsmify, run with no arguments, will provide the following help message:

```
------------ Usage: new_fsmify <grammar file stem> -------------

            e.g. new_fsmify my_grammar
            (where my_grammar.rgra is a grammar file)

Will generate a finite state machine (.fsm) corresponding to a
regular grammar defined in <grammar file> (.gra).

Here are some of the regular grammar definition guidelines:
1) All grammatical symbols must be written as '<symbol>'
2) The highest level symbol must be a symbol called '<root>'
3) A definition must be of the form ( <symbol> = [...] )
4) The right hand portion above ('[...]') can include:
   - references to other grammatical symbols, <other_symbols>
   - terminal symbols (these can have any characters OTHER than
     '<' '>' '?' '*' '+' '|')
5) References to grammatical symbols MUST AVOID RECURSION.
Something like ( <np> = word <np> ) will cause the program to crash.
The program handles regular grammars, not context free grammars.
6) <symbol>? means that this symbol is optional. Using '?' with
a terminal symbol will NOT work, eg 'b?' cannot be used.
7) <symbol>+ means one or more repetitions.
8) <symbol>* means 0 or more repetitions.
PBMEC does NOT handle this case gracefully; for now, using the
markers '|', '+' and '?' to represent 0 or more repetitions is recommended,
e.g. (<phrase> = <word>? | <word>+ ) instead of (<phrase> = <word>* )
 9) Again, the characters ? + and * cannot be used with terminal symbols
10) The '|' character means 'or', as in
( <name> = e r i k | a l a i n | sh i g e r u )
A definition must not end with a '|', eg ( <v> = a | i | u | e | o | )
is not correct.
11) There must be at least one space between the
characters '(' ')' '=' and |. (<v>=a|i|u|e|o) will not work.

example 1:

A C-V pair grammar can be defined using rules such as:
```

(<root> = <cv>)
(<cv> = <c>? <v>)
(<c> = b | d | g | p | t | k ....)
(<v> = a | i | u | e | o)

example 2:

A connected word grammar can be defined using rules such as:
(<root> = <word>+)
(<word> = i ch i | n i | s a n2 | y o n2 | g o)

There is some redundancy, as
(<cv> = <c> <v> | <v>) is equivalent to
(<cv> = <c>? <v>);
however, the latter produces a more compact finite state machine.
The reason for this is that the FSM is LEFT-RIGHT tree structured, and
without an explicit declaration that something is optional, it will not
group identical right-hand portions.)

12) (added Jan. 1996) Definition of context loops:

Context loops (e.g. phoneme pair/triple grammars) can be defined
using the special category <CLOOP-something>. Such symbols will be expanded
so as to loop all arcs back to _legal_ entry states, as defined by the presence
of left context <L-something> and right context <R-something> markers.
Thus, <L-a> b <R-c> refers to a 'b' in the left context of an 'a' and
right context of a 'c'. It will be looped back to any 'c's that will accept
a 'b' as their left context, via the use of a <L-b> marker, or the use of no
marker at all. Similarly, this 'b' will refuse the connection (from the left)
of any states that are not 'a' states.

More examples:
<L-a> b <R-*> will only accept 'a' states on the left, but can loop back to anything.
<L-*> b <R-c> will accept anything on the left, but will loop back only to 'c' states.

Specifying start/exit states:
'<CLOOP_START> something' will allow a connection from the start of the
loop. 'something <CLOOP_EXIT>' will allow a connection to the end of the
loop. Only arcs that have these special hooks will be connected to the
start/end of the loop. However, if <CLOOP_START> is not used at all

```
all arcs will be connected to the start of the loop, and similarly for
<CLOOP_EXIT>.

Example of a simple phoneme context grammar:

( <root> = pau <CLOOP-phon> pau )

( <CLOOP-phon> = <CLOOP_START> a <R-*>
                | <L-a> b <R-c>
                | <L-b> c <R-d>
                | <L-c> d <R-a>
                | <L-d> a <CLOOP_EXIT> )


bugs, problems to mcd@hip.atr.co.jp
```

Of course, the representations and language definitions used here are subject to rapid change; the above describes only the choices current at the time of this writing. Some issues that may influence future changes include using a grammar definition language that is compatible with that used in other systems at ATR and elsewhere (e.g. HTK) and modifying the *pbmec* program to read the grammar description file directly and generate the finite state machine on the fly (as is done in HTK), rather than generating the latter in a separate step.

7 Beam Search

A typical approach to pruning time-synchronous DP search is, for each frame that is being processed, to maintain a list of active states and remove states with scores below a certain threshold. The main DP loop described above (5) is thus modified to be:

loop over frames x_t in utterance
    loop over active phonetic states p in FSM
        loop over *pbmec* sub-states s of phoneme in p
            calculate $E_{(t,s)}(\mathbf{x}_1^T) = e(\mathbf{x}_t, s) + \min_{r \in R_s} E_{t-1,r}(\mathbf{x}_1^T)$
        end
    calculate new threshold
    using new threshold, generate new list of active phonetic states
    end
end

Beam search involves a number of additional steps, of course, that are not diplayed here. For instance, within the loop over the sub-states in a phoneme, a is checked to test whether the sub-state being considered is active or not. Furthermore, there must be a test whether the predecessor considered is active or not. Also, DP scores must be propagated through grammatical states that connect phonetic states together. This propagation occurs after one loop over the active phonetic states. Backpointers are also propagated. We should also mention that there are two vectors of DP scores, old and new; pointers to these vectors are switched at the end of the loop over active phonetic states in order to update the old scores.

Figure 4 illustrates which states in the finite state machine are active over time when using pruning of this sort. Figure 5 shows both the state identities and their accumulated DP scores over time.

Maintaining a Constant Beam Size

The method proposed by Ney et al. [35] is to choose this threshold so as to preserve a constant number of states (corresponding to a desired beam width) during the DP process. This can be done by forming a cumulative histogram of state scores, i.e. that represents how many states have a score greater than a given score - see figures 7 and 7. The cumulative histogram can then be used to choose a threshold corresponding to a particular number of states. The advantage of this method is that it is very inexpensive computationally to form the above histogram and choose the threshold. Contrasted to methods where ranked lists or heaps of active states are maintained, this method is very simple.

The accuracy of this pruning method depends on the number of bins used to form the histogram; in practice, we have found that using a number that is 10 percent of the desired beam width is sufficiently accurate for a broad range of beam widths. More bins could be used at negligible computational cost.

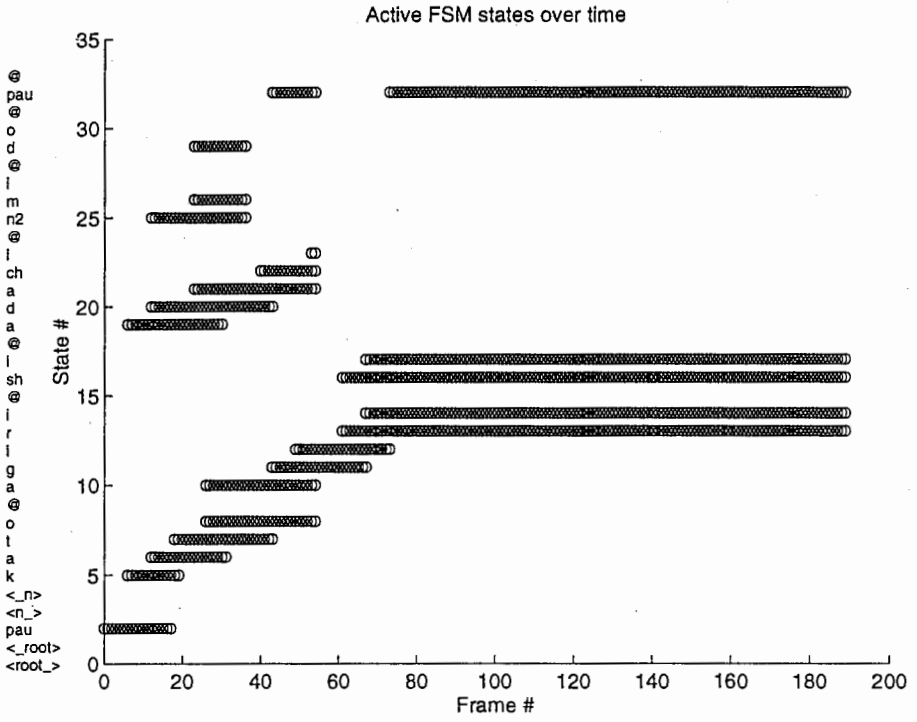

Figure 4: Active states over time when using a beam of 5 states for a finite state machine representing 4 isolated words. "Active states" are states which remain after score-based pruning, OR the successors of such states.

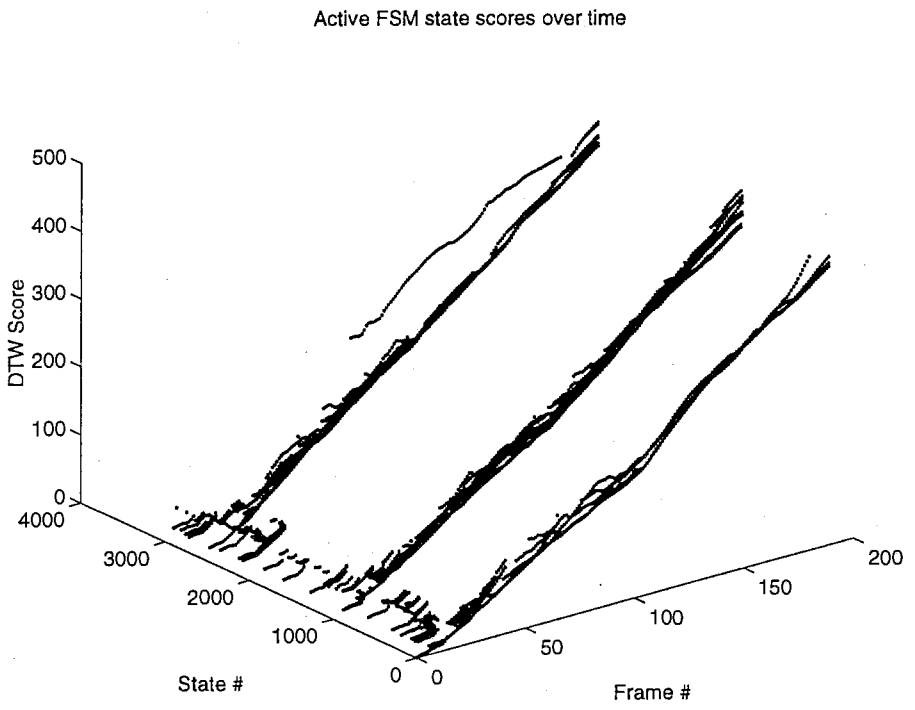

Figure 5: Active FSM state scores over time. The states in the finite state machine are lined up on one axis, and the DP scores of states that have not been pruned is plotted over time. Task: 311 names task.

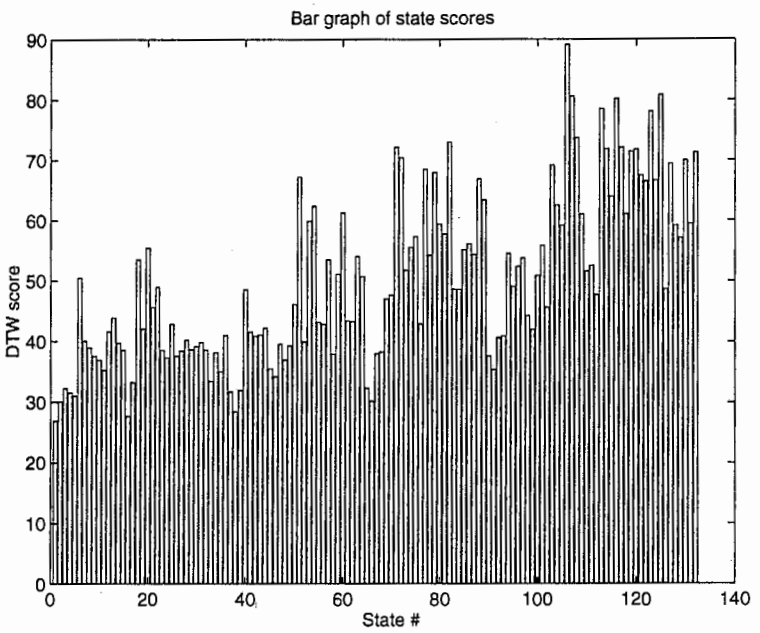

Figure 6: Beam search for time-synchronous DP (One Pass DP): maintain a list of active states; remove states with scores below a certain threshold.

Variable Beam Size

The histogram pruning method is effective given the goal of choosing a threshold that maintains a constant number of active states. Our motivation for adopting this scheme is simply to control the computational requirements of the search, which will be directly proportional to the specified beam width. This goal, of course, is not an absolute. Choosing a threshold is not necessarily tied to controlling the beam width accurately. It might be desirable to prune heavily when the top scores are much better than the other scores (i.e. the system is "confident" in its top scores and doesn't need to examine other DP paths), and prune much less when the top scores are not much better than the other scores. In other words, the aggressiveness of the pruning could be a function of the difficulty in recognizing a particular utterance. Schemes that use the score of the last state in the DP network (such as that used in HTK) are similar in spirit in that one is typically less confident in a DP path at the beginning of the search than at the end, and thus the pruning can be increasingly aggressive as one progresses through the utterance.

In order to illustrate the difference between these schemes, we compared the number of active states over time for a search using histogram pruning with a search using the score of the last state as the pruning threshold (Figure 8). One can see from this figure that histogram pruning effectively sets an upper limit on the number of active states at any frame of the search, while the number of active states varies widely for pruning based on the score of the last state.

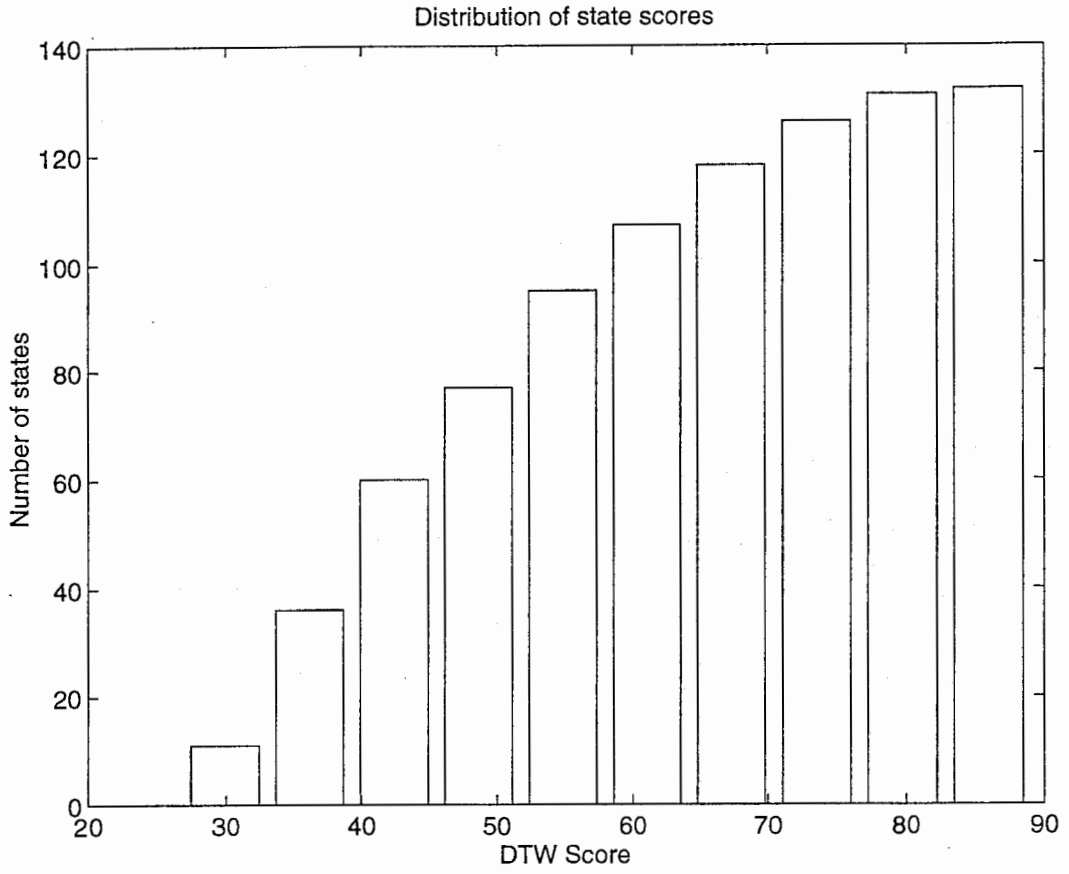

Figure 7: Ney's histogram pruning method: use score corresponding to the desired beam width (number of states) as the threshold.

8 The A* algorithm for speech recognition

8.1 A* search

The premise of A* search [33] is that the problem at hand is some sort of *graph search* problem. Loosely defined, a graph is a set of nodes, some pairs of which are connected by a (directed) arc, from one member of each pair to the other. In the context of speech recognition, nodes will typically represent a phonetic symbol, and the arcs in the graph will be determined by the lexical and grammatical constraints of the task. Furthermore, the cost of a particular path through the graph will typically be related to a probability or distance of a match between observed (input) speech and an acoustic model for the phonetic symbol of each node along the path.

A general procedure for finding a complete path through the graph is to maintain a stack of partial paths (paths that originate with the start node, but have not yet reached the end node), use path costs to order this stack somehow, and expand the path at the top of the stack. Expansion of a path consists in creating new paths that now contain, in addition to the parent path, all the successor nodes that can be reached from the nodes on the parent path.

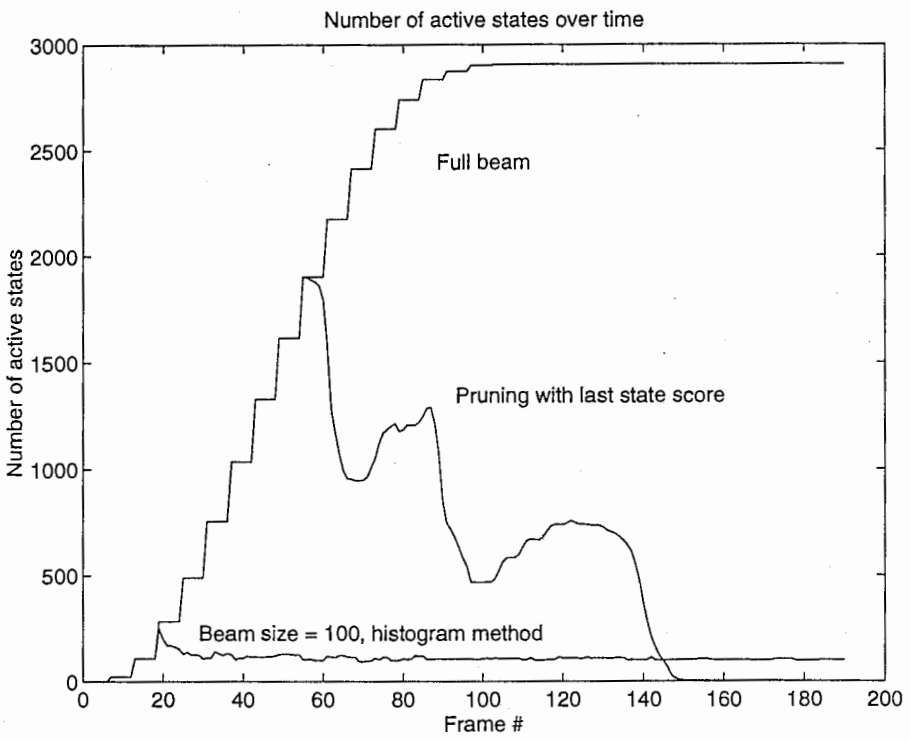

Figure 8: Number of active states over time for DP search using 1) (top) no pruning, 2) (middle) last state score pruning 3) (bottom) histogram pruning. Histogram pruning is a simple way of maintaining an approximately fixed number of actives states during the search. Task = 311 isolated words.

The new paths are then inserted into the ranked stack. When a path popped from the stack contains the end node (or "goal" node), it is output as a complete path, and the procedure terminates . If the stack is ranked in an arbitrary or inefficient manner, there is of course no guarantee that the path found through this procedure is a minimum cost path through the graph. If the stack is ordered by putting the longest paths at the top of the stack, the procedure ressembles a depth-first strategy, and if the shortest paths are put at the top of the stack, the procedure corresponds to a breadth-first strategy.

Algorithm A

More specifically, the TOTAL score estimate $f'(p)$ of a complete path hypothesis p constrained to go through node n is defined as:

$$f'(p) = g(n) + h'(n), \tag{5}$$

where $g(n)$ is the actual score of an optimal path from the start node to node n (corresponding to the partial path that has been explicitly generated), and $h'(n)$ is the score *estimate* from node n to the end node (corresponding to the remaining, not-yet-generated path between node n and the end node). $f'(p)$ is the evaluation function that will be used to rank hypotheses during the search. The search procedure can then be outlined as follows:

- Initialize a stack of partial path hypotheses with the start node.
- LOOP:
    1. If the result stack contains N complete paths, exit.
    2. Take the top partial path off the hypothesis stack.
    3. If it is a complete path, put it on the result stack.
    4. If it is not a complete path, generate all its successor nodes, and put the resulting paths (original path plus successor node) on the stack. Use $f'(p)$ to rank the stack.
    5. goto LOOP

This general procedure, using an evaluation function to rank the stack, is referred to in [33] as *algorithm A*.

Algorithm A*

If the estimate $h'(n)$ is a lower bound (i.e., a better score) of the true value $h(n)$, the first complete path (i.e., the first path to reach the end node) output to the result stack is guaranteed to be the true best path through the graph; the second complete path output is guaranteed to be the next best path through the graph, and so on. When $h'(n)$ satisfies this condition for *admissibility*, the search algorithm is called *algorithm A**.

The reason A* outputs the best path first can be described informally by considering that the total path score estimate, $f'(p)$, is always lower in cost than the true path score $f(p)$. Since $f'(p)$ is used to rank the stack, the top path on the stack always has a score that is at

least as good as that of the true best path. Since A* always chooses the top path on the stack for expansion/output, if we suppose that the first path output by the A* search was NOT optimal, i.e. that $f'(p)$ was larger than $f(p)$, we reach a contradiction [33].

It is shown in [33] that the closer the estimate $h'(n)$ is to the true value, $h(n)$, the smaller the number of hypotheses that need to be checked. If the estimate is *equal* to the true value, the search will find the best path(s) in the smallest possible number of partial path extensions.

(Note: in [33], the general graph search procedure actually generates a *search graph*, not a stack of paths. The best path is found at the end of the search by tracing back through the search graph. A search graph is a more compact representation than a list of paths, as the overlapping parts of different partial paths are not duplicated in the graph, whereas they are duplicated in the list. Many current applications of N-best string search in speech recognition use a search graph to represent the N-best candidates [45] [12].)

8.2 A* for finding the exact N-best string candidates given an utterance and a speech recognizer

A* search can be used in the context of speech recognition to generate the N-best string candidates given an unknown utterance. Given a finite state machine representing the grammatical/lexical constraints for a speech recognition task, the recognition problem can be formulated as a graph search problem. The scoring of paths through the graph will be related to the local matches between input frames and the acoustic models used by the recognizer. It is shown in [44] how Viterbi scoring can be used to generate partial path score estimates that satisfy the A* admissibility constraint.

The "tree-trellis (forward-backward) algorithm" can be outlined as follows:

- 1. For a given utterance, perform a time synchronous DP forward pass through the finite state machine, storing accumulated forward scores to each grammar node for each frame.
  2. Perform an A* search starting at the last state of the finite state machine. Extend and score paths backwards, adding the backward scores to the forward scores generated in (1) to obtain a total string score.

The A* search component here is essentially as described above, except that here the search is performed backwards, from the end node to the start node.

Size of the A* stack

If the same finite state machine is used for both forwards and backwards scoring, the total path scores will be exact, i.e. the estimates will be equal to the true scores, and the search can directly follow the best path backwards, with no need to investigate paths of unknown ultimate merit. In this case, a stack of size N will be sufficient to generate the N-best hypotheses.

If the forward estimates are NOT exact, no restriction must be placed on the size of the stack in order to generate the true N-best hypotheses. Conversely, if in the interest of reducing

computation time, a restriction on stack size is enforced, the N-best hypotheses found by the A* search are not guaranteed to be the true N-best hypotheses.

In the following we describe this procedure in more detail.

Generating a new partial backward string hypothesis

A parent string hypothesis, taken from the top of the stack, is extended by generating all possible backward word contents through the finite state machine (i.e., all possible successors of the hypothesis), starting from the left-most side of the parent hypothesis.

For example, let us say the stack contained the following partial hypotheses:

1. "jimukyoku desu"
2. "desu"
3. "kaigi desu"
4. "desu ka"

Each of these hypotheses has a single evaluation score ($f'(p)$ below), as well as a *vector* of backward DP scores over time. The backward scores represent the DP scores from the end of each hypothesis string to its beginning (e.g., for the first hypothesis, the DP scores through "desu" and then through "jimukyoku"), matched backward (i.e. starting at the top right of the DP grid) over the entire length of the utterance.

Being on top of the stack, "jimukyoku desu" will be removed, and all its successors generated. Assuming that "kaigi" is a possible successor, then it could be extended to "kaigi jimukyoku desu."

Scoring a new partial backward string hypothesis

The new word content ("kaigi", in the example) is *backward*-DP matched against the entire utterance, in an isolated word fashion, but using the parent hypothesis' vector of backward scores over time to initialize the top row of the backward-DP match.

The backward DP match generates a new backward score vector in the bottom row of the DP grid. This vector is added to the forward score vector at the edge of the new word content, yielding a *total* score vector. The best score in this vector is used as the overall score of the new hypothesis. This score will be used to insert the hypothesis into the stack in the right position. Figure 8.2 describes how the forward and backward scores are added to generate a complete path score.

The total score of the new path, "kaigi jimukyoku desu", is used to reinsert the new path back into the stack, which might now be:

1. "desu"
2. "kaigi jimukyoku desu"
3. "kaigi desu"

4. "desu ka"

The same procedure is performed for all the other successors of the parent hypothesis, "jimukyoku desu." The expanded and scored successor paths are all re-inserted into the stack, and the procedure is repeated.

More formally, the total score of a path extension p to grammatical node m is defined as:

$$f'(p) = min_t(g(m,t) + h'(m,t)) \tag{6}$$

This definition follows that in equation (5), with the addition of the time index (corresponding to the utterance frames). Here, $g(m,t)$ corresponds to the explicitly, backward-generated scores over time for the best partial path from the end node back to node m, and $h'(m,t)$ corresponds to the best forward scores (from start node to node m) over time, generated in the time-synchronous forward DP search. While the string content of the backward-generated $g(m,t)$ is known (in the above example, this corresponds to "kaigi jimukyoku desu"), it is not clear what the string content of $h'(m,t)$ is. What matters is that $h'(m,t)$ is a lower bound on the cost of the optimal path between the start node and node m. Since DP was used in the forward phase to generate $h'(m,t)$, this admissibility condition is satisfied.

Using $f'(p)$ to rank the stack of hypotheses, the A* search proceeds from the end node to the start node, continually extending partial paths backwards by one lexical unit at a time. The first complete path(s) output by the search will be the best path(s) through the finite state machine, given the particular utterance and the set of model parameters.

Reducing the number of hypothesis expansions

As each expansion of a partial hypothesis corresponds to a backward DP matching (in the above example, of the word "kaigi"), it is very desirable to reduce the number of expansions. It is shown in [13] how the search can be made more efficient by performing a kind of look-ahead (or look-back?), which explicitly considers the successors to a new partial path.

Instead of using the overall best forward scores to node m, as described above, one can consider the best forward scores from all the possible successors (moving backwards through the finite state machine) to node m, taken individually. In other words, a separate forward score vector $h'(m,t)$ is used for each possible successor. The merging of $h'(m,t)$ and the backward score vector $g(m,t)$ is then performed for each of the successors, giving each successor a score. The evaluation score of the new partial hypothesis is based on the best successor (as before), but a ranked list of the other successors is attached to the new partial hypothesis, which is re-inserted back into the stack. When, at a later stage of processing, that partial hypothesis is removed from the top of the stack for expansion, the score of the *next* best successor is used to give the same partial hypothesis a new evaluation score; this is in turn re-inserted into the stack, along with the 1-best successor extension of the hypothesis.

This more detailed scoring of a new hypothesis allows the search to avoid expanding successors which turn out not to be very good. Thus, the overall number of expansions is reduced. We have not yet implemented this procedure.

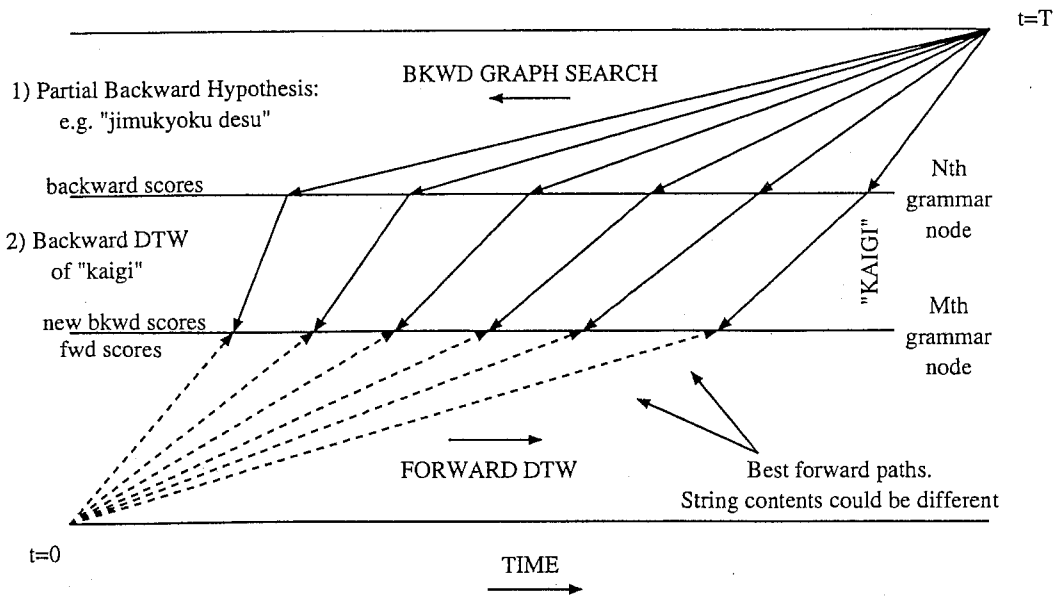

Figure 9: A* generation and scoring of a new partial string hypothesis in the context of speech recognition

8.3 A* algorithm to speed up the search for the optimal path

So far we have stressed the ability of the A* algorithm to produce the true N-best string candidates efficiently (assuming that the forward scores are admissible, i.e. that no pruning is used in the forward phase, and that the grammatical constraints of the forward phase are either the same as or looser than those used in the backward phase.) However, the fact that the search allows one to use in the forward phase a looser set of constraints suggests an application of A* that goes beyond finding the N-best candidates: speeding up the overall search by using a simplified grammar in the forward phase, followed by an A* search using the full grammar in the backward phase. This application has been investigated by Huang et al. [13] [14]. It is not clear, however, how much more efficient this procedure is. The simpler the forward constraints, the faster the forward search, but the slower the backward A* search. When the forward constraints are much weaker than the backward constraints (the latter being the "final" set of constraints for the target task), the forward score estimates are much smaller than the true forward scores. The A* admissibility condition is satisfied, but the search now has to evaluate more candidates that at first seem promising but turn out to have high costs. However, there may be a combination of forward and backward constraints that balances the forward and backward search costs in a manner that yields a lower overall search cost than a full search, or even than a beam search (up to certain beam sizes). In addition, as mentioned above, the stack used during the A* search can be limited to a preset maximum size. In general, this removes the guarantee of optimality, but may effectively speed

up the search, while preserving good recognition accuracy. We investigate these questions in the next section.

9 Experiments

In the preceding, we described 1) a fixed-width beam pruning method for time-synchronous dynamic programming in the forward phase, and 2) an A* based method that allows both a) generation of N-best candidates and b) a possibly faster overall search by using a simple grammar in the forward time-synchronous phase, followed by an A* search using the full target grammar. We also mentioned that it might be desirable to prune the stack used during A* search, in order to reduce computational time.

We aimed to compare the recognition accuracy and computational cost of 1) and 2b). Specifically, given a finite state machine that represents the target task (e.g. isolated word recognition for a given vocabulary), and a smaller finite state machine that represents a simpler set of constraints (i.e., one that overgenerates the target strings), we compared:

Approach I: Forward - beam search through target FSM; Backward - A* search through target FSM

Same target task finite state machine backwards and forwards, but various degrees of pruning in the forward search (the histogram pruning method of Ney et al. [35]) (followed by A* search in the backward phase to generate the N-best candidates).

Cost. The bulk of the computation occurs in the forward pass; the backwards A* search merely finds the N-best string candidates, which for a small N, does not significantly affect the overall search time. Using histogram beam pruning, the cost is directly proportional to the length of the utterance and the beam width.

Accuracy. As beam pruning is used, the forward scores are not guaranteed to be the true forward scores; there is no guarantee of A* admissibility in the N-best search. The accuracy is linked to the beam width, but the precise relationship is unknown.

Approach II: Forward - full search through simplified FSM; Backward - A* search through target FSM

No pruning but use of a simplified FSM (e.g. unconstrained phoneme grammar or unconstrained syllable grammar) in the forward pass, followed by a backward A* search using a finite state machine corresponding to the target task. Partial path hypotheses will be expanded backwards one phoneme at a time; only phoneme successors that are allowed by the target finite state machine will be considered.

Cost. The forward phase is now a very small component of the overall search time. It is the backward phase that essentially determines the overall search cost. Each partial path expansion involves a backward DP matching (alternatively, "phoneme verification") of a (legal) successor phoneme. These verifications are computationally expensive. The overall cost of the backward search is difficult to estimate, as it depends on the relation between the forward estimates (based on the simple forward grammars) and the true scores according

to the backward (target) grammatical constraints. Limiting the stack size offers a way of controlling the search cost.

Accuracy. If the stack size is not limited, the forward scores are admissible, and the A* algorithm used in the backward phase is optimal; i.e., it will find the best string candidate given the utterance and the model parameters. (Of course, the best candidate is not necessarily the correct one.) If the stack size is limited, the guarantee of optimality is lost.

9.1 Database

The database we used to investigate these two approached for speeding up the search is a set of utterances recorded at ATR HIP Laboratories for the purpose of designing a speaker independent telephone-based isolated word recognition system, that would recognize the names of HIP researchers and forward telephone calls to them. We used 570 utterances for training, 114 for testing, from about 40 speakers, recorded at 8 kHz over the telephone. The utterances were transcribed using a set of 25 phonemes. Speech frames were represented as 16 FFT Bark scale coefficients, calculated every 5 ms with a window size of 20 ms.

9.2 Finite state machine design

The *new_fsmify* program described above (in the ATR HIP environment, /usr/hearing/recog/bin/new_fs was used to generate two left-right tree-structured finite state machines for vocabularies of 63 names and 311 names, respectively. These are the two target tasks. In addition, finite state machines were generated for a much smaller set of constraints: 1) an unconstrained connected phoneme grammar, 2) an unconstrained syllable (mora) grammar (the latter is illustrated in Figure 9.2). In order to speed up the backward A* search after a forward search with the simple FSMs, *right-left* tree-structured FSMs for the target tasks were generated. These are used to direct the A* search through only the lexically meaningful paths of the unconstrained phoneme/syllable FSM.

9.3 Recognizer design

As no labels were available, a segmental k-means procedure (in the ATR HIP environment, /usr/hearing/recog/bin/kmeans_seg) was used to automatically segment the utterances, and output a set of sub-phonetic states with accompanying mean vectors for each phoneme. These were used to initalize the PBMEC model (with the (diagonal) weighting matrix initialized at the identity matrix). MCE/GPD training was performed using a full beam (i.e. no pruning) for 10 epochs and different values of the loss function parameter $Q1$ [32], for the 63 names recognition task. On testing data, again with a full beam, the best set of reference vectors produced a correct recognition rate of 95.6 percent.

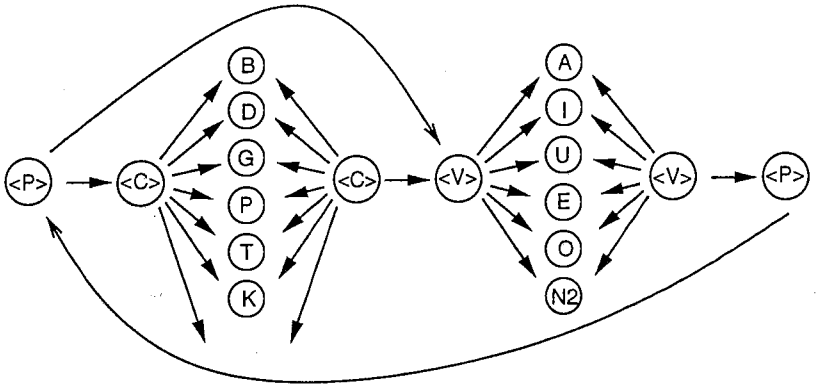

Figure 10: FSM representing an unconstrained connected syllable grammar

9.4 Results

Approach I, performance vs. beam width

Figure 11 contrasts test data performance for the two name recognition tasks as a function of beam size, expressed here as a fraction of the total number of states in the finite state machine. The computational cost of the search is directly linked to the beam size, which, through the histogram pruning method, is nearly constant throughout the search.

Approach II, performance vs. stack size

Ideally, Approach II would be fast enough that it would not require limiting the stack size. Nonetheless, we investigated the impact on recognition accuracy of limiting the stack size.

Figures 12 and 13 show test data performance for the two name recognition tasks as a function of stack size. The size of the vocabulary used (63 or 311) is the largest that the stack will ever grow, corresponding to the total possible number of paths through the finite state machine.

Approach II, number of phoneme verifications vs. stack size

The limit on stack size does not directly reflect the computational cost of the search. Rather, it is the number of phoneme verifications during the backward phase that will determine computation time. Every time a partial path is extended backward by one phoneme, a new phoneme-synchronous backward-DP procedure must be carried out over the whole length (or a large segment) of the utterance. (We refer to this match as a "phoneme verification.") Thus, the more extensions required by the A* algorithm, the slower the search. We first show the average number of phoneme verifications per utterance required by the A* search

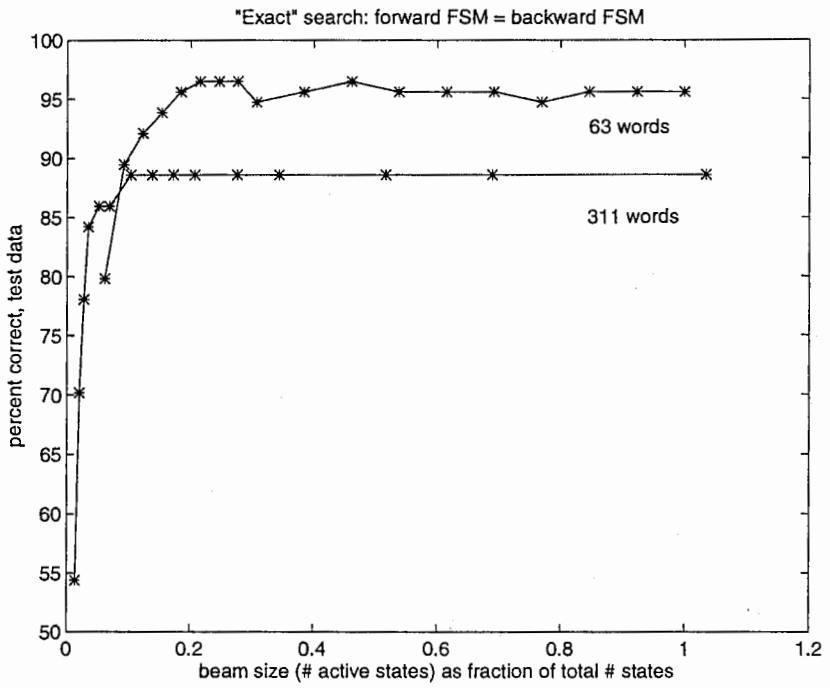

Figure 11: Performance vs. beam size (as fraction of the maximum number of states) for 63 names task and 311 names task.

as a function of the limit on stack size, for both target tasks, in figures 14 and 15. Here the unconstrained syllable grammar was used in the forward phase.

Approach II, performance vs. number of phoneme verifications.

Finally, we show plots of performance vs. the number of phoneme verifications, indirectly controlled by limiting the size of the stack, in figures 16 and 17. Again, the unconstrained syllable grammar was used in the forward phase.

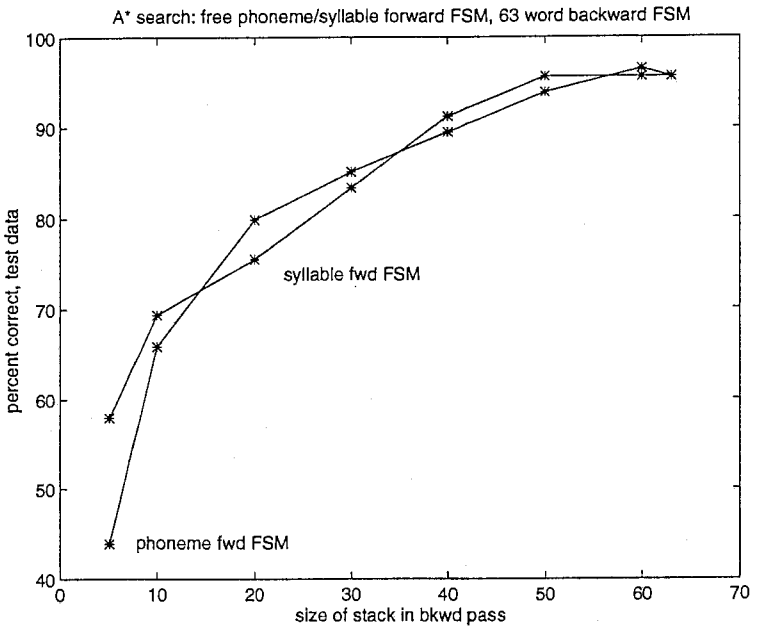

Figure 12: Performance vs. stack size for free phoneme/syllable forward search followed by full lexical backward search, 63 names task

10 Discussion

Use of beam search in forward phase(Approach I)

First, a general comment about pruning in the forward, time-synchronous phase: Ney's histogram pruning method works well and is easy to implement. In general, pruning time-synchronously is probably more flexible that maintaining a beam in a phoneme synchronous search, as pruning is effectively performed at a finer grain. As a consequence, the overhead of pruning is higher for time-synchronous approaches than more phoneme or word synchronous approaches.

More specifically, one sees from Figure 11 that even when the beam is just a small fraction of the overall number of states in the target FSM, recognition accuracy is quite high. This is particularly true for the larger task, where a beam of 20 percent of the total number of states is sufficient to ensure a recognition accuracy that is as high as when performing a full search.

Use of simplified FSM in forward phase (Approach II)

In an isolated word recognition scheme, for a vocabularly of 300 words, approach II requires an average of about 1200 phoneme verifications for each utterance, or nearly half of the maximum, to achieve optimal search accuracy. For the smaller task, nearly 0.7 of the maximum number of phoneme verifications is required. In both cases, a much larger fraction of the maximum number of verifications is required than for beam search in order to obtain similar recognition performance. In this sense, the beam search is much more efficient.

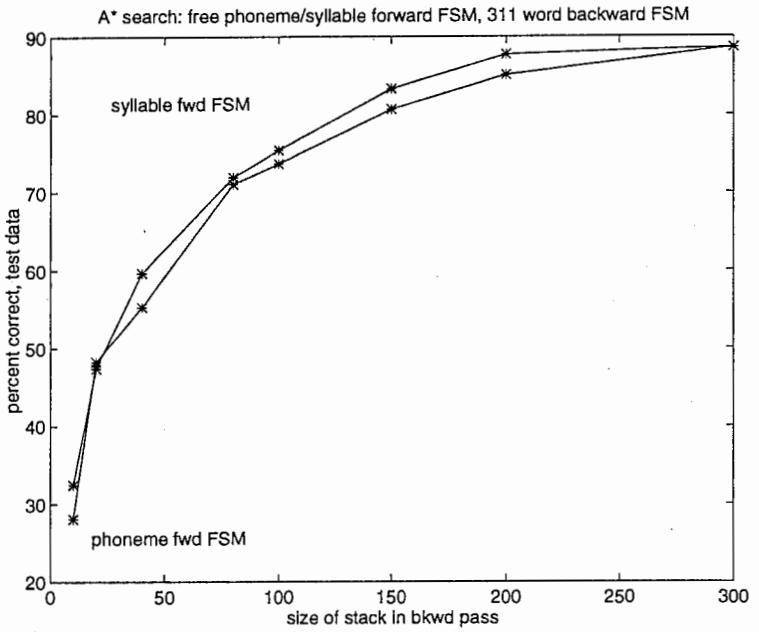

Figure 13: Performance vs. stack size for free phoneme/syllable forward search followed by full lexical backward search, 311 names task

On the other hand, the overhead of performing backward phoneme synchronous DP matches is less than the overhead of performing time synchronous DP matches for phoneme states in the context of beam pruning. Time synchronous beam search performs its pruning at a more fine grained method than a phoneme synchronous search. It typically involves frequent checks on whether a state is active, whether its predecessor is active, and so on. In contrast, the pruning in phoneme synchronous search is at a higher level, which decides (here, through use of the A* stack) whether or not to match a phoneme model over the entire length of the utterance, or some large segment thereof. Once the decision has been made, the DP match for that phoneme proceeds with no local decisions on whether to match a particular grid point. Thus, there is less overhead. (Conceivably one could perform time-synchronous beam search in the backward direction also.) In this light, the notions of beam size compared to maximum number of states, or number of verifications compared to the maximum, cannot be compared directly. We should also consider pure CPU time, and for instance, show plots of CPU time versus performance.

Figures 14 and 15 suggest that the A* search spends a considerable amount of time investigating paths that turn out not to be very good. Clearly, this is linked to the discrepancy between the forward score estimates, based on simplified FSMs, and the true path scores based on the target FSMs. Here, the differences between free phoneme/syllable forward FSM and backward lexical FSM are quite large. Providing a syllable grammar instead of a phoneme grammar helps a little.

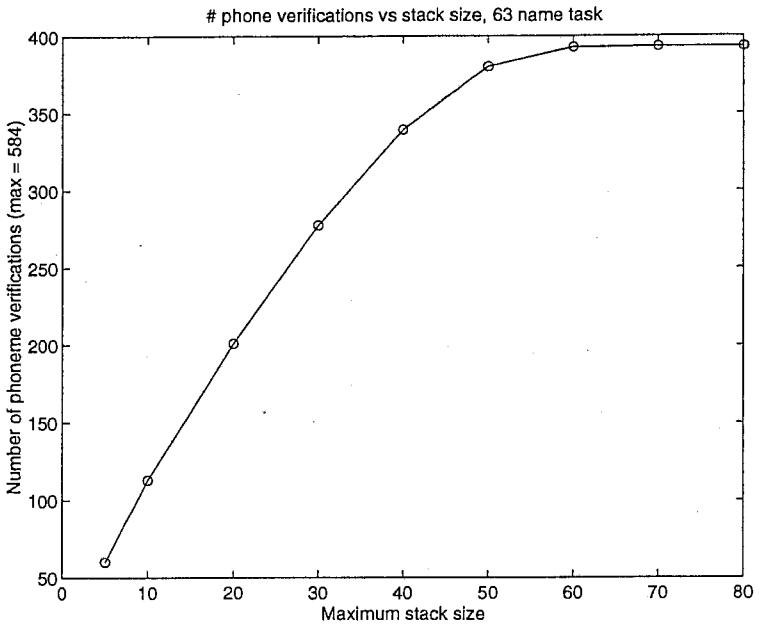

Figure 14: Average number of phoneme verifications per utterance vs. maximum stack size, 63 names task (the maximum number of phoneme verifications for this task is 584.)

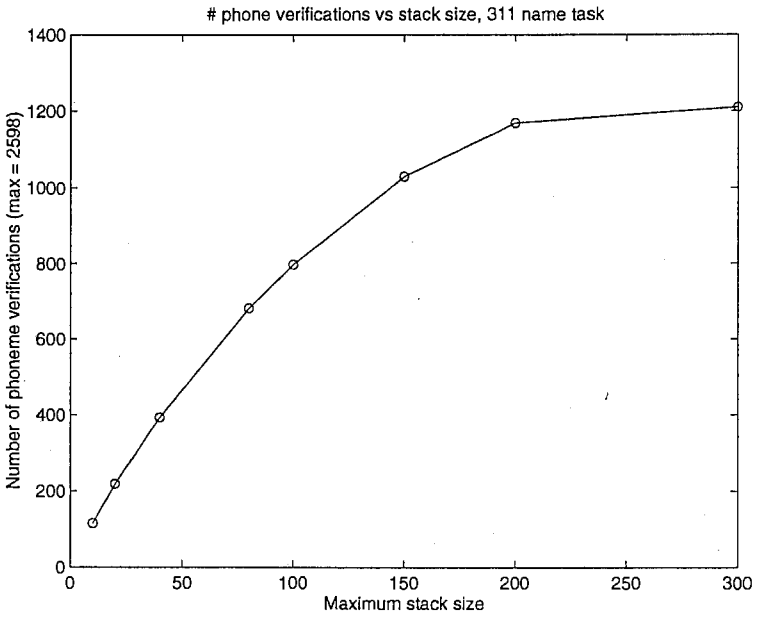

Figure 15: Average number of phoneme verifications per utterance vs. maximum stack size, 311 names task (the maximum number of phoneme verifications for this task is 2598.)

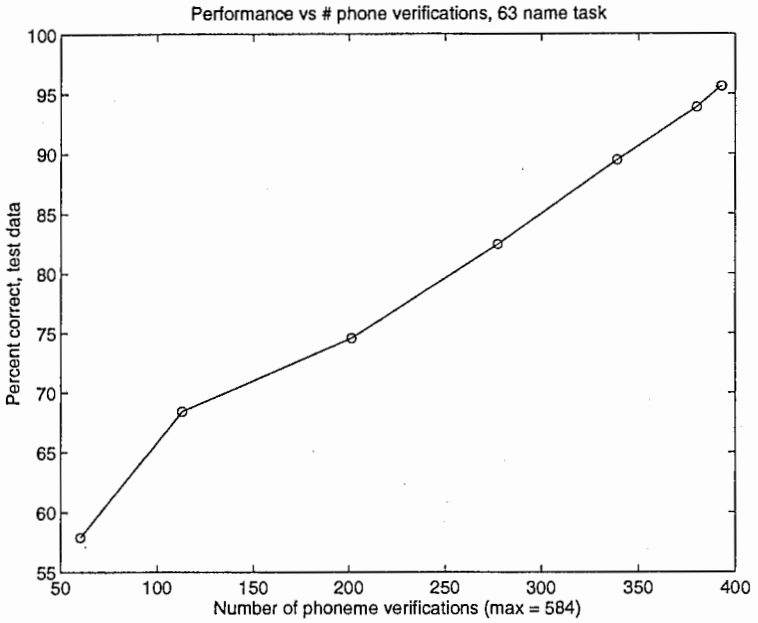

Figure 16: Performance vs. average number of phoneme verifications per utterance, 63 names task

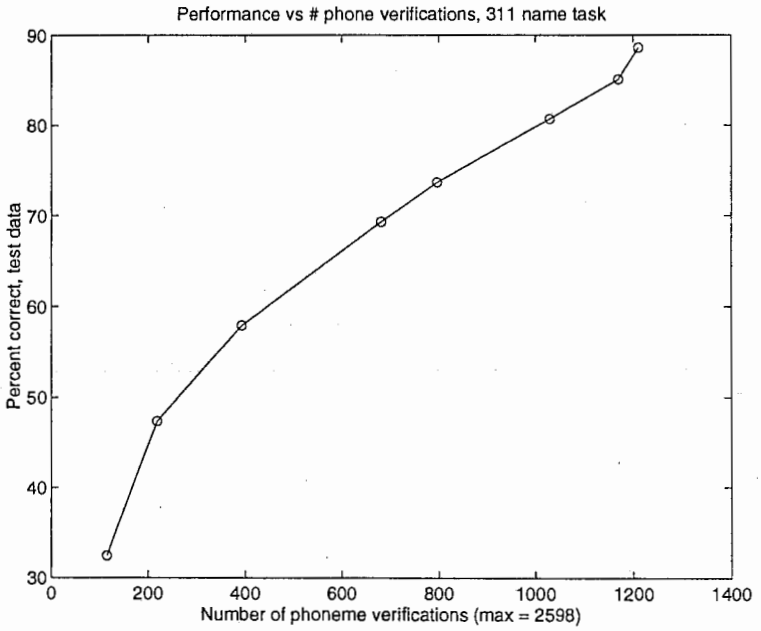

Figure 17: Performance vs. average number of phoneme verifications per utterance, 311 names task

Improving the A* forward estimates, reducing the number of verifications

In order to reduce the number of A* path expansions, two methods come to mind. The first is to score partial hypotheses in a more sophisticated manner, described above, which uses a ranked list of scores for the new partial hypothesis, each score in the list corresponding to a successor of the new hypothesis. This allows the stack ranking to be based on scores that in effect look ahead one level of expansion without performing that expansion. This method will clearly reduce the number expansions. However, it only affords one level of look ahead. The sucessor-based scores are themselves limited by the accuracy of the forward estimates.

This brings us to the other way in which the number of path expansions may be reduced: making the forward esimates more accurate. This could be done by the method described in [13], where rather than a grammar of connected phonemes or syllables for the forward phase, a grammar of connected *sub-words* is used. The sub-words are larger chunks of the words in the lexicon than are phonemes or syllables. The constraints of a sub-word grammar are thus closer to the constraints of the target grammar. Therefore the forward scores will be closer estimates of the true path scores according to the target grammar. This will make the A* search more efficient. However, the forward phase will now be more costly.

We have not investigated either of these methods.

Dispensing with backward search?

Both approach I and approach II assume the use of A* search in a backward phase. One could question this assumption; an all forward pass method (with simple backtracking through pointers, and/or production of the "immortal paths" as needed) may be more suited to pipelining a recognition system. The forward-only A* method proposed by Noda and Sagayama [36] may offer a way to apply A* to a more easily pipelined system.

11 Conclusion

From the simple and somewhat rough comparison presented here, it seems that time-synchronous beam search through the target finite state machine is a simpler, faster and more robust method than the more involved combination of simple forward time synchronous search followed by detailed, A* based, phoneme synchronous backward search. The former method can apparently provide high recognition accuracy even with a beam that is just a small fraction of the overall number of states in the target finite state machine. This is not the case for the latter approach, at least for the tasks we examined. In this light, though the *pbmec* program provides the option for using a different grammar in the backward search than used in the forward search, our current focus in developing the HIP telephone-based recognition system is on the simpler approach where the same finite state machine, that for the target task, is used in both forward and backward phases, and histogram-based beam pruning is used to speed up the forward search when necessary.

The comparison presented here was quite useful to us in deciding what direction to adopt regarding the search component of our recognizer. However, a more careful examination of 1) the properties of time synchronous beam search vs. phoneme synchronous beam search, 2) the relation between forward and backward A* estimates, and 3) experimental evaluation on additional tasks, is necessary before pronouncing the final word on the relative merits of the two approaches investigated here.

The main purpose of this report was to describe the use of DP search in the context of the ATR HIP *pbmec* recognizer. Our particular use of DP is quite standard and close to the state of the art. What could be improved is the efficiency of our computer implementation of the search components, which still has considerable overhead, for example in maintaining a time-synchronous beam of active states, or managing the A* stack. Hopefully, we have made the use of DP search in a speech recognizer more comprehensible to the researcher not versed in the intricacies of this rapidly expanding domain.

References

[1] Amari, S. (1967). *A Theory of Adaptive Pattern Classifiers.* IEEE Transactions, EC-16, No. 3, pp. 299-307.

[2] Austin, S., Schwartz, R. and Placeway, P. (1991). *The Forward-Backward Search Algorithm.* Proceedings of the IEEE, ICASSP-91, pp. 697-700.

[3] Bates, M., Bobrow, R., Fung, P., Ingria, R., Kubala, F., Makhoul, J., Nguyen, L., Schwartz, R. and Stallard, D. (1993). *The BBN/HARC Spoken Language Understanding System.* Proceedings of the IEEE, ICASSP-93. pp. II-111-114.

[4] Bottou, L. (1991). *Une Approche theorique de l'Apprentissage Connexioniste; Applications a la reconnaissance de la Parole.* PhD Thesis, Ch. 3, Universite de Paris Sud, Centre D'Orsay. (In French).

[5] Chang, P.-C. and Juang, B.-H. (1991). *Discriminative Training of Dynamic Programming Based Speech Recognizers.* Proceedings of the IEEE ICASSP-91, pp. 549-552.

[6] Chang, P.-C. and Juang, B.-H. (1992). *Discriminative Template Training for Dynamic Programming Speech Recognition.* Proceedings of the IEEE, ICASSP-92, pp. I:493-6.

[7] Chang, P.-C. and Juang, B.-H. (1993). *Discriminative Training of Dynamic Programming Based Speech Recognizers.* IEEE Transactions on Speech and Audio Processing, Vol. 1, No. 2, April 1993, pp. 135-143.

[8] Chen, J.-K., Soong, F. and Lee, L.-S. (1994). *Large Vocabulary Word Recognition Based on Tree Trellis Search.* Proceedings of the IEEE, ICASSP-94, pp. II:137-140.

[9] Chou, W., Juang, B.-H. and Lee, C.-H. (1992). *Segmental GPD Training of HMM Based Speech Recognizer.* Proceedings of the IEEE, ICASSP-92, pp. I:473-476.

[10] Fu, K.-S. (1968). *Sequential Methods in Pattern Recognition and Machine Learning.* Academic Press.

[11] Haffner, P., Franzini, M. and Waibel, A. (1991). *Integrating Time Alignment and Neural Networks for High Performance Continuous Speech Recognition.* Proceedings of the IEEE, ICASSP-91, pp. 105-108.

[12] Hetherington, I.L., Phillips, M.S., Glass J.R. and Zue, V.W. (1993). *A* Word Network Search for Continuous Speech Recognition.* Proceedings of the 3rd European Conference on Speech Communication and Technology, September 21-23, Berlin, Germany.

[13] Huang, E.-F., Wang, H.-C. and Soong, F. (1994). *A Fast Algorithm for Large Vocabulary Keyword Spotting Application.* IEEE Transactions on Speech and Audio Processing, Vol. 2, No. 3, July 1994, pp. 449-452.

[14] Huang, E.-F., Soong, F. and Wang, H.-C. (1994). *The Use of Tree-Trellis Search Large-Vocabulary Mandarin Polysyllabic Word Speech Recognition.* Computer Speech and Language, volume 8. pp. 39-50.

[15] Iwamida, H., Katagiri, S., McDermott, E. (1991). *Speaker-Independent Large Vocabulary Word Recognition Using an LVQ/HMM Hybrid Algorithm.* Proceedings of the IEEE, ICASSP-91, pp. 553-556.

[16] Iwamida, H., Katagiri, S. and McDermott, E. (1991). *Large Vocabulary Word Recognition Using LVQ and HMM Phoneme Concatenated Training.* Proceedings of the Acoustical Society of Japan, Fall Meeting, pp. 99-100.

[17] Juang, B.-H. and Katagiri, S. (1992). *Discriminative Training* ASJ Special Issue, Vol. 13, No. 6, November 1992. pp. 333-339.

[18] Juang, B.-H. and Katagiri, S. (1992). *Discriminative Learning for Minimum Error Classification.* IEEE Transactions on Signal Processing, vol. 40, No. 12, December 1992, pp. 3043-3053.

[19] Katagiri, S., Lee, C.-H. and Juang, B.-H. (1990). *A Generalized Probabilistic Descent Method.* Proceedings of the of Acoustical Society of Japan, Fall Meeting, pp. 141-142.

[20] Katagiri, S., Lee, C.-H. and Juang, B.-H. (1991). *Discriminative Multilayer Feedforward Networks.* Proceedings of the 1991 IEEE Workshop on Neural Networks for Signal Processing, August 1991, pp 11-20.

[21] Katagiri, S., Biem, A., and Juang, B.-H. (1993). *Discriminative Feature Extraction.* in "Artificial Neural Networks for Speech and Vision," Chapter 18. Ed. R. Mammone, Chapman & Hall. pp. 278-293.

[22] Kohonen, T. (1990). *The Self-Organizing Map.* Proceedings of the of the IEEE, 78, No. 9. pp. 1464-1480.

[23] Kohonen, T., Barna, G. and Chrisley, R. (1988). *Statistical Pattern Recognition with Neural Networks: Benchmarking studies.* Proceedings of the of IEEE, ICNN, vol. I, July 1988, pp. 61-68.

[24] Komori, T. and Katagiri, S. (1991). *A New Discriminative Training Algorithm for Dynamic Time Warping-Based Speech Recognition.* IEICE, Tech. Report SP91-10, June 1991. pp. 33-40.

[25] Komori, T. and Katagiri, S. (1992). *Application of GPD Method to Dynamic Time Warping-based Speech Recognition.* Proceedings of the IEEE, ICASSP-92. pp. I:497-500.

[26] McDermott, E. and Katagiri, S. (1989). *Shift-Invariant, Multi-Category Phoneme Recognition using Kohonen's LVQ2.* Proceedings of the IEEE, ICASSP-89. pp. 81-84.

[27] McDermott, E. (1990). *LVQ3 for Phoneme Recognition.* Proceedings of the Acoustical Society of Japan, Spring Meeting, pp. 151-152.

[28] McDermott, E. and Katagiri, S. (1991). *Shift-Tolerant LVQ for Phoneme Recognition.* IEEE Transactions on Signal Processing, 39, No. 6, June 1991, pp. 1398-1411.

[29] McDermott, E. and Katagiri, S. (1991). *Discriminative Training for Various Speech Units.* Technical Meeting of IEICE, SP 91-12, June 1991. pp. 47-54.

[30] McDermott, E. & Katagiri, S. (1992). *Prototype-Based Discriminative Training for Various Speech Units.* Proceedings of the IEEE, ICASSP-92. pp. I:417-420.

[31] McDermott, E. and Katagiri, S. (1994). *Prototype Based Minimum Error Training for Speech Recognition.* Journal of Applied Intelligence, Kluwer Academic Publishers, volume 4, pp. 245-256.

[32] McDermott, E. and Katagiri, S. (1994). *Prototype Based Discriminative Training for Various Speech Units.* Computer Speech and Language, volume 8, pp. 351-368.

[33] Nilsson, N. (1982). *Principles of Artificial Intelligence.* Tioga Publishing Company.

[34] Ney, H. (1984). *The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition.* IEEE Transactions on Acoustics, Speech and Signal Processing, vol. ASSP-32, pp. 263-271.

[35] Steinbiss, V., Tran, B.-H. and Ney, H. (1994). *Improvements in Beam Search.* Proceedings of the International Conference on Spoken Language Processing, S36-5.1, pp. 2143-2146.

[36] Noda, Y., and Sagayama, S. (1994). *Forward Heuristic Functions Applied to Beam Search in HMM-LR Speech Recognition.* Proceedings of Acoustical Society of Japan (fall meeting), pp. 139-140.

[37] Rainton, D. and Sagayama, S. (1992). *Minimum Error Classification Training of HMMs -Implementational Details and Experimental Results.* IEICE, Technical Report SP91 XX.

[38] Rainton, D. and Sagayama, S. (1992). *Appropriate Error Criterion Selection for Continuous Speech HMM Minimum Error Training.* Proceedings of the International Conference on Spoken Language Processing.

[39] Sakoe, H. and Chiba, S. (1978). *A Dynamic Programming Algorithm Optimization for Spoken Word Recognition.* IEEE Transactions on Acoustics, Speech and Signal Processing, vol. ASSP-26, Feb. 1978, pp. 43-49.

[40] Sakoe, H. (1979). *Two-level DP-matching - A Dynamic Programming-Based Pattern Matching Algorithm for Connected Word Recognition.* IEEE Transactions on Acoustics, Speech and Signal Processing, vol. ASSP-27, pp. 588-595.

[41] Sawai, H. (1991). *TDNN-LR Continuous Speech Recognition System Using Adaptive Incremental Training.* Proceedings of the IEEE, ICASSP-91. pp. 53-56.

[42] Schwartz, R. and Austin, S. (1991). *A Comparison of Several Approximate Algorithms for Finding Multiple (N-BEST) Sentence Hypotheses.* Proceedings of the IEEE, ICASSP-91, pp. 701-704.

[43] Schwartz, R., Austin, S., Kubala, F., Makhoul, J., Nguyen, L., Placeway, P., Zavaliagkos, G. (1992). *New Uses for the N-Best Sentence Hypotheses within the BYBLOS Speech Recognition System.* Proceedings of the IEEE, ICASSP-92. pp. 53-56.

[44] Soong, F. and 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, ICASSP-91, pp. 705-708.

[45] Strom, N. (1994). *Optimising the Lexical Representation to Improve A* Lexical Search.* HKT Technical Report, STL-QPSR 2-3/1994.

[46] Su, K.-Y. and Lee, C.-H. (1991). *Robustness and Discrimination Oriented Speech Recognition Using Weighted HMM and Subspace Projection Approaches.* Proceedings of the IEEE, ICASSP-91. pp. 541-544.

[47] Waibel, A., Sawai, H. and Shikano, K. (1989). *Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Neural Networks.* Proceedings of the IEEE, ICASSP-89. pp. 112-115.

[48] Zue, V., Glass, J., Goodine, D., Leung, H., Phillips, M., Polifroni, J. and Seneff, S. (1991). *Integration of Speech Recognition and Natural Language Processing in the MIT Voyager System.* Proceedings of the IEEE, ICASSP-91, pp. 713-716.

[49] Yamaguchi, K. (1992). *Continuous Mixture HMM-LR using the A* Algorithm for Continuous Speech Recognition.* Proceedings of the International Conference on Spoken Language Processing, pp. 301-304.