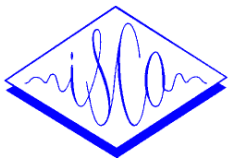

3rd International Conference on
Spoken Language Processing (ICSLP 94)
Yokohama, Japan
September 18-22, 1994

TIME SYNCHRONOUS HEURISTIC SEARCH IN A STOCHASTIC SEGMENT BASED RECOGNIZER

*Nick Cremelie and Jean-Pierre Martens**

ELIS, University of Gent, St.-Pietersnieuwstraat 41, B-9000 Gent (Belgium)

ABSTRACT

A single pass heuristic search method to be included in a stochastic segment based recognizer is presented. Thanks to a novel and efficient implementation of Nilsson's A/A* graph search algorithm, and thanks to the introduction of an appropriate heuristic function, the presented algorithm significantly outperforms the standard Viterbi beam search. Moreover, it was possible to conceive a time-synchronous search (no prior knowledge of the endpoint needed), and to restrict the amount of storage required. As such, the algorithm is extremely suitable for real-time implementation.

1 INTRODUCTION

Heuristic search is an attractive method for solving various search problems. It is based upon the *best-first* principle: at any time the search is guided towards the direction that seems most likely to result in the best solution. In order to decide on which direction to follow, a heuristic search relies on some prior knowledge about the search problem.

Continuous speech recognition is commonly modeled as a search for the best path in a trellis network, and a full search can be performed by means of the Viterbi algorithm. However, as the vocabulary of the recognizer grows, a full search may become too expensive in terms of computational requirements. A reduction of the computational load can be obtained by retaining at each time index only a limited number of paths, falling within a beam around the best path found so far. This strategy is called the Viterbi beam search. The heuristic search method presented in this paper will prove to yield a substantial reduction of the computational complexity with respect to the classical beam search.

Some form of heuristic search has been integrated in various recognition systems [6, 7], often in a second pass (see e.g. [7]: the path scores are gathered in a forward Viterbi pass, and used in a backward heuristic pass in order to obtain the N best sentence hypotheses). The search method presented here aims at finding the best sentence hypothesis in a single forward pass.

The outline of this paper is as follows. First we briefly review the heuristic search algorithm introduced by Nilsson [3], and then we present a new and efficient implementation of this algorithm. In section 3 we derive an appropriate heuristic function for our segment based recognizer [1, 2]. In section 4 we show how the search can be organized in a time synchronous manner. Experimental results are summarized and discussed in section 5.

*Senior Research Associate of the National Fund for Scientific Research

2 ALGORITHM AND IMPLEMENTATION

Search problems can formally be defined as follows: given a set A of nodes, a set T of possible transitions between nodes (each with an associated score), a set $S \subset A$ of source nodes and a set $G \subset A$ of goal nodes, find the path of successive transitions, starting in S and ending in G, yielding the largest possible accumulated score (this is the sum of the scores of all transitions along the path).

An uninformed search method usually has to expand all nodes in the course of the search process ("expanding a node" means examining all transitions leaving that node; the nodes reachable from that node are called "successor" nodes). A heuristic or informed search method on the other hand uses some knowledge about the search problem to estimate which direction is most likely to result in the best solution. Roughly speaking a heuristic search starts at the source nodes, expands nodes in some strict order imposed by the prior knowledge, and stops when a goal node is reached.

A practical algorithm was introduced by Nilsson [3]. In this algorithm a so-called *evaluation function* $f(n)$ is used to order the nodes. This function estimates the score of the best path from source to goal passing through node n. Usually $f(n)$ is the sum of some $g(n)$ being the score of the best path from S to n, and some $h(n)$ (called the *heuristic function*) being an estimate of the best score to get from n to G. Moreover, if $h(n)$ is an upper bound of the exact best path score from n to G, one can show that the heuristic search is admissible, i.e. it is guaranteed to find the best path (A* search). Nilsson's algorithm uses two lists of nodes, an OPEN list containing visited nodes still waiting for further expansion, and a CLOSED list containing the expanded nodes. The algorithm runs as follows:

1. Put the source nodes on OPEN.
2. **If** OPEN is empty **then** stop the search (no solution!)
   else select from OPEN the node n with the largest $f(n)$, remove it from OPEN and put it on CLOSED.
3. **If** n is a goal node **then** stop the search (solution found!).
4. For all successor nodes n_i of n, compute $f(n_i)$, and:
   If n_i is not on OPEN, nor on CLOSED
   then put n_i on OPEN and store the path to n_i
   else if the new $f(n_i)$ > the old one **then** replace the old path to n_i by the new one and put n_i on OPEN.
5. goto 2.

Obviously, several list searches are required at each execution of the main loop. Since the computational cost of searching in an

unordered list is of the order p (p being the number of entries in the list) [4], the above algorithm has a computational complexity of the order p^2. By sorting the lists, this complexity can be made proportional to $p \log p$, but this is still impractical if the number of visited nodes is large.

We propose a more efficient implementation which is not based on lists. Suppose that the search space consists of a finite set of N nodes, each having a unique index between 1 and N (this index is easy to compute from the coordinates in the trellis). It is then possible to set up an array of N entries, with entry n containing all characteristics of node n: $f(n)$, $g(n)$, a backpointer to the predecessor node, and a flag to indicate whether the node is open, closed or new (= not yet visited). In this way, information is retrieved after a single indexing operation in an array.

To determine the open node having the largest score $f(n)$, a K-ary maximum tree is introduced. The nodes of the search space constitute the terminals of the tree (figure 1) and each non-terminal a holds a pointer to the best open node (terminal) reachable from a. Consequently, the root of the tree points to the best open node in the entire search space.

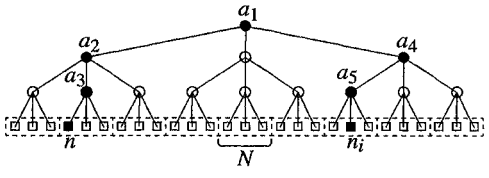

Figure 1: *K-ary maximum tree; the terminals □ represent the search space nodes.*

It is useful to define the non-terminals on the branches connecting a terminal n with the root as the "ancestors" of n. The "children" of a non-terminal a are the K non-terminals or terminals on the branches leaving a.

Initializing the tree is straightforward: all nodes except the source nodes are new, the pointers in the source nodes' ancestors are set appropriately, and those in the remaining non-terminals are set to point to some dummy node with an extremely low score. Whenever a node changes in some way, only small portions of the tree need an update. One can distinguish two cases:

Case 1: node n is selected for expansion and becomes closed. Since n was the open node with the largest $f(n)$, all ancestors of n (a_1, a_2, a_3 in the case of figure 1) were referring to n and have to be updated as n becomes closed. The update process starts at the nearest ancestor of n (a_3 in figure 1): the maximum score of the open nodes reachable from a_3 is determined, and the pointer in a_3 is adjusted to point to the node yielding this maximum. This requires $K + \alpha K$ tests, with $\alpha \leq 1$ (for each node: test on open status, test on score if open). Updating the remaining ancestors of n (a_2, a_1 in figure 1) requires $K - 1$ tests for each ancestor: only the maximum score of the terminals pointed to by its children is to be determined. Consequently, the total number of tests for this case is $(K + \alpha K) + (K - 1)(\log_K N - 1) = T_1$.

Case 2: The value of f in a successor n_i of n is modified. In this case, node n_i becomes or remains open. The new $f(n_i)$ must be compared to the score f in the nodes referred to by the ancestors of n_i, starting at the nearest ancestor of n_i and going up in the tree as long as $f(n_i)$ improves the maximum. The number of tests for this update is $\beta \log_K N = T_2$ ($\beta \leq 1$).

Case 1 occurs once per node expansion, while case 2 can apply to all successors of the expanded node. If the expected number of applications of case 2 is called γ, then the total number of tests per expansion is $T_1(K) + \gamma T_2(K) = T(K)$. Note that α and β can only be estimated a posteriori, whereas γ is problem-dependent. However, assuming the worst case values for α, β and γ (an upper bound on γ follows from the problem description), one can determine an upper bound on the total number of tests per expansion. It is clear that this number does *not* depend on the number of expansions already made. Consequently the computational complexity of the proposed implementation is of the order p.

An adequate implementation for the K-ary tree is one using separate arrays for the different levels of the tree. Furthermore, K should be chosen as to minimize $T(K)$. A more profound study of $T(K)$ learns that N and α have little impact on the location of this minimum. The parameters β and particularly γ are more important. Fortunately $T(K)$ does not increase too quickly if K exceeds its optimum value. Hence even a rather inaccurate estimate of α, β and γ can yield a near optimal K. Notice that a larger K also results in a lower storage requirement for the tree.

3 THE HEURISTIC FUNCTION

Speech recognition can be described as a search in a two-dimensional trellis (figure 2). One direction exhibits discrete times representing possible segment boundaries, the other direction represents the states of a statistical automaton. The states and most of the transitions emerge from the phoneme and word models. The transitions between words emerge from the task (language) model. By repeating the models (i.e. the states *and* the transitions between states) at all discrete times, one obtains a trellis network defining the search space. Each transition in the network has an associated score composed of log probabilities. As our system is adopting a Stochastic Segment Model (SSM) approach [5, 1, 2], different transitions spanning different time intervals (e.g. transitions t_{i2}, t_{i3} on figure 2) are allowed at any node. The word initial states at the beginning of the utterance are considered as the source nodes, and the word final states at the end of the utterance are the goal nodes.

The main problem now is to find a suitable heuristic function $h(n)$, predicting the score of the best path from any node $n = (i,j)$ to a goal node. We have used a heuristic function which is based on properties of the initial segmentation algorithm that produced the boundaries b_i:

$$h(n) = h(i,j) = N_p(i) \cdot S_p + N_w(i) \cdot S_w$$

with $N_p(i)$ = estimated number of phonetic segments on the best path from b_i to b_I, S_p = average score per phonetic segment along the best path, $N_w(i)$ = estimated number of inter-word transitions on the best path from b_i to b_I, and S_w = average inter-word transition score (emerging from a language model).

This function takes into account the scores of the intra-word as well as the inter-word transitions. Of course the different terms and factors still need to be specified.

An estimate $N_w(i)$ can be derived from $N_p(i)$: if the expected number of phonetic segments per word is q, then $N_p(i)/q$ is a

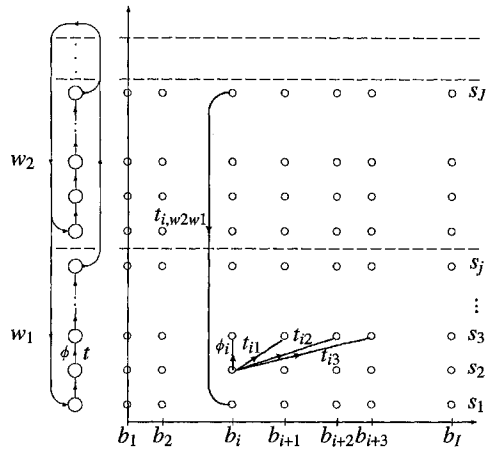

Figure 2: *Trellis network for speech recognition. Two word models, w_1 and w_2, are shown; $b_1, \ldots, b_I$ are the possible segment boundaries, $s_1, \ldots, s_J$ the states.*

possible estimate $N_w(i)$. The accuracy of this estimate will obviously depend on the vocabulary and on the language model. The value of q can be calculated from the segmentation of training utterances.

It seems reasonable to take $N_p(i)$ proportional to $(I - i)$, the number of boundaries following b_i. During the training of our recognizer, it was established that the number of phonetic segments was close to 70% of the number of boundaries . Consequently, $0.7(I - i)$ was considered a good estimate of $N_p(i)$.

If the language model (grammar) has a perplexity P_{LM}, the score S_w is given by $\log(1/P_{LM})$. The average (acoustic) score per segment S_p, will slightly depend on P_{LM}, as the acoustical scores contribute more importantly to the total score when P_{LM} is large. It is however not obvious how to anticipate this effect. Therefore we have estimated S_p by processing a set of training utterances (of course using a full search and the same language model that will be used afterwards). Clearly one may expect that for an independent test set the optimal value of S_p will be somewhat smaller. Taking all this into account, we obtain for $h(i,j)$:

$$h(i,j) = 0.7(I - i)\left(S_p + \frac{1}{q}\log(\frac{1}{P_{LM}})\right) \triangleq S_i \cdot (I - i)$$

The quantity S_i can be interpreted as an average score per initial segment, incorporating the contributions of both the intra-word and the inter-word transitions.

Note that the calculation of $h(n)$ is trivial. It therefore suffices to store $g(n)$ (rather than $g(n)$ *and* $f(n)$) for each node n. As suggested in [3], we have also tested $g(n) + \lambda \cdot h(n)$ as an evaluation function ($\lambda > 0$).

4 TIME SYNCHRONOUS HEURISTIC SEARCH

In its present form the above search method still has a few drawbacks: it does not proceed time synchronously, and it requires prior knowledge of the endpoint of the utterance (note the presence of I in the expression of our heuristic function). This implies that the search phase can only start when the whole utterance is available. The result may be an intolerable delay at the output of the recognizer.

Thanks to the specific choice of our heuristic function, we can easily deal with these problems. One learns from the description of the algorithm that the evaluation function $f(n)$ is used only for comparing nodes. In fact, the absolute value of $f(n)$ is not important, as long as the order imposed by $f(n)$ is maintained. Consequently, $f(n) = g(n) + h(n)$ can be replaced by $f'(n) = g(n) + h'(n)$, with $h'(n) = h(n) + C$ (C being an arbitrary constant), without this affecting the search. Since

$$h(i,j) = S_i \cdot (I - i) = S_i \cdot (-i) + S_i \cdot I \triangleq h'(i,j) + C,$$

$h'(i,j)$ can be used instead of $h(i,j)$. It is similar to $h(i,j)$, except that it only relies on the starting point of the speech. This being established, we can derive a time synchronous implementation of the search. The last available segment boundary is considered as a temporary endpoint. At the moment that the next node expansion would reach a node beyond this endpoint, the search process is suspended until the next segment of speech is available. The boundary of this segment is then taken as a new endpoint, and the search is continued until another new segment is needed. This procedure is repeated until the endpoint of the utterance is reached. At that point the search is continued as usual, until a goal node is reached.

One could argue that the presented strategy may require large amounts of storage. However, as the search does not seem to make giant leaps back in time, node expansions take place in an area close to the moving endpoint. It is therefore feasible to discard the nodes on the segment boundaries falling outside a fixed length window moving along with the temporary endpoint. The appropriate window length L clearly depends on the recognition task. In our case, a window of 25 segments (about three to four words) appeared to be sufficient. The storage requirements are obviously limited in this way.

In the context of real time recognition, one must further assume that only a limited number N_{epf} of node expansions can be performed in each time frame. It is therefore possible that a new speech segment is available before the search has reached the temporary endpoint. Experiments have shown that it is still acceptable in this case to move the window to the new boundary. With no limits on the number of node expansions ($N_{epf} = \infty$), it appears that at some instants (typically around difficult word transitions) a high number of expansions is necessary before the window can be moved to the next position, while at other instants only a few expansions are needed. With a realistic value of N_{epf}, some expansions may be postponed, but if the window length is appropriate the search is able to "catch up" in time, and the recognition performance remains unaffected.

5 EXPERIMENTAL RESULTS

Our system was tested on a speaker independent continuous speech recognition task, with a vocabulary of 413 Dutch words. The test set consisted of 130 different sentences (10 different speakers, 13 sentences per speaker). Two different cases were considered. In one case, a bigram language model was used. In the other case, an artificial language model was constructed by randomly adding word pairs to the bigram model in order to increase the perplexity.

	word errors	expanded nodes	CPU-time in sec. per sec. speech
Viterbi Search	2.96%	100%	1.19
Beam Search			
$\Delta = \log(200000)$	2.96%	4.40%	0.22
$\Delta = \log(50000)$	3.70%	3.08%	0.21
$\Delta = \log(20000)$	4.44%	2.38%	0.19
Heuristic Search			
$\lambda = 1.000, L = 25$	2.96%	1.88%	0.14
$\lambda = 0.650, L = 25$	2.96%	0.53%	0.07
$\lambda = 0.325, L = 25$	4.81%	0.31%	0.05
$\lambda = 0.225, L = 25$	7.41%	0.30%	0.05
$\lambda = 0.650, L = I$	2.96%	0.67%	0.09
$\lambda = 0.325, L = I$	2.84%	0.31%	0.07
$\lambda = 0.225, L = I$	4.81%	0.30%	0.07

Table 1: *Results with the bigram language model (775 word pairs, test set perplexity = 4.1)*

	word errors	expanded nodes	CPU-time in sec. per sec. speech
Viterbi Search	20.62%	100%	2.48
Beam Search			
$\Delta = \log(200000)$	21.23%	14.20%	0.44
$\Delta = \log(20000)$	21.73%	6.91%	0.32
$\Delta = \log(10000)$	23.33%	5.39%	0.27
Heuristic Search			
$\lambda = 1.000, L = 25$	20.86%	6.25%	0.46
$\lambda = 0.750, L = 25$	20.37%	3.21%	0.26
$\lambda = 0.625, L = 25$	22.84%	2.14%	0.18

Table 2: *Results with the artificial language model (7000 word pairs, test set perplexity = 26.5)*

Three different search methods were evaluated: the standard Viterbi search, a Viterbi search with threshold pruning (beam search), and the heuristic search method presented in this paper. The maximum score on a particular boundary, diminished by some amount Δ, was considered as the pruning threshold for the beam search on that boundary. The factor S_i was computed as indicated before. We found $S_i = 0.92$ with the bigram model, and $S_i = 0.71$ with the artificial language model.

The results are summarized in tables 1 and 2. Displayed are the total word error rate (deletions + insertions + substitutions), the percentage of nodes being expanded, and the average CPU-time per second of speech required for the search (measured on an IBM RS/6000 workstation). Apparently the heuristic search outperforms the Viterbi beam search: it can attain the same recognition accuracy with less expanded nodes and less computations.

The calculated values of S_i did not result in an admissible heuristic function. For both tasks there were about 15% best path errors with these values, but the errors did not affect the recognition results. Apparently, there are many near-optimal paths yielding the same word string and the heuristic method is able to find such a path. This is not always the case for a beam search.

By applying a $\lambda < 1$, the efficiency of the search can be improved substantially. However, with a restricted window length L, the recognition accuracy drops if λ becomes too small (see table 1). In this case the search proceeds too fast (*depth-first* character), and too much information gets lost. With a window that can hold the entire utterance ($L = I$), smaller values of λ are still acceptable, but they do not significantly speed up the search.

The very low number of node expansions — compared to a beam search with the same recognition accuracy — indicates that the heuristic function, although simple, provides sufficient knowledge about the search. The success of this function undoubtedly emerges from the fact that it is based on the properties of a good speaker-independent initial segmentation algorithm [1]. Note that it was shown in [8] that a similar heuristic function, but based on the number of frames left to be recognized, was not at all capable of yielding an efficient search.

Compared to beam search, the heuristic search has a higher computational cost per node expansion, but a lower fixed cost (i.e. the cost if only the minimal number of nodes is expanded). Heuristic search therefore becomes efficient as soon as the number of node expansions drops below a certain threshold. Thanks to our new implementation, this threshold can be quite high.

Although a reasonable limit on N_{epf} was imposed (200 for the tests of table 1, and 500 for those of table 2), the window was moved prematurely at only 2.6% of the boundaries. This had no effect on the path found by the search (as compared to $N_{epf} = \infty$). Additional tests showed that the number of premature movements can be raised to about 15% without jeopardizing the results.

6 CONCLUSION

By integrating a heuristic search in our segment based recognizer, we were able to speed up the linguistic processing by a factor 2 to 4 with respect to a Viterbi beam search. This was possible thanks to the introduction of a new implementation of the search, and a suitable heuristic function. An interesting property of the heuristic search is that it can be organized in a time-synchronous way.

REFERENCES

[1] J.P. Martens, A. Vorstermans, N. Cremelie (1993), "A new Dynamic Programming/ Multi-Layer Perceptron Hybrid for continuous speech recognition," in *Proceedings of EUROSPEECH-93*, pp.1937-1940.

[2] J.P. Martens (1994), "Neural Networks as Probability Estimators in a Segment-based Speech Recognizer," in *Modern Modes of Man-Machine Communication*, edts. Howat and Kacic (ISBN 86-435-0073-9).

[3] N.J. Nilsson (1980), *Principles of Artificial Intelligence*, Palo Alto, California: Tioga.

[4] D. Knuth (1973), *The Art of Computer Programming, Volume 3 / Sorting and Searching*, Reading, Massachusetts: Addison-Wesley.

[5] M. Ostendorf, S. Roucos, "A Stochastic Segment Model for Phoneme-based Continuous Speech Recognition," IEEE Trans. ASSP-37, pp.1857-1869.

[6] D.B. Paul (1992), "An Efficient A* Stack Decoder Algorithm for Continuous Speech Recognition with a Stochastic Language Model," in *Proceedings of ICASSP-92*, pp.I25-I28.

[7] S. Austin, R. Schwartz, P. Placeway (1991), "The Forward-Backward Search Algorithm," in *Proceedings of ICASSP-91*, pp.697-700.

[8] H. Ney (1992), "A comparative Study of Two Search Strategies for Connected Word Recognition: Dynamic Programming and Heuristic Search," in *IEEE Trans. PAMI-14, No. 5*, pp.586-595.