Algorithms for an Optimal A* Search and Linearizing the Search in the Stack Decoder.*

Douglas B. Paul

Lincoln Laboratory, MIT
Lexington, Ma. 02173

Abstract

The stack decoder is an attractive algorithm for controlling the acoustic and language model matching in a continuous speech recognizer. It implements a best-first tree search of the language to find the best match to both the language model and the observed speech. This paper describes a method for performing the optimal A* search which guarantees to find the most likely path (recognized sentence) while extending the minimum number of stack entries. A tree search, however, is exponential in the number of words. A second algorithm is presented which linearizes the search at the cost of approximating some of the path likelihoods.

Introduction

Speech recognition may be treated as a tree network search problem. As one proceeds from the root toward the leaves, the branches leaving each junction represent the set of words which may be appended to the current partial sentence. Each of the branches leaving a junction has a probability and each word has a likelihood of being produced by the observed acoustic data. The recognition problem is to identify the most likely path (word sequence) from the root (beginning) to a leaf (end) taking into account the junction probabilities (the stochastic language model) and the optimum acoustic match (including time alignment) given that path.

This paper is concerned with the network search problem and therefore correct recognition is defined as outputting the most likely sentence given the language model, the acoustic models, and the observed acoustic data. If the most likely sentence is not the one spoken, that is a modeling error—not a search error. This paper will assume for simplicity that an isolated sentence is the object to be recognized. (The stack decoder can be extended to recognize continuous sentences.)

The stack decoder [4], as used in speech, is an implementation of a best-first tree search. The basic operation of a sentence decoder is as follows [1, 2]:

1. Initialize the stack with a null theory.
2. Pop the best (highest score) theory off the stack.
3. if(end-of-sentence) output sentence and terminate.
4. Perform acoustic and language-model fast matches to obtain a short list of candidate word extensions of the theory.
5. For each word on the candidate list:
   (a) Perform acoustic and language-model detailed matches and add the log-likelihoods to the theory log-likelihood.
      i. if(not end-of-sentence) insert into stack.
      ii. if(end-of-sentence) insert into stack with end-of-sentence flag = TRUE.

   (note: end-of-sentence may be optional)
6. Go to 2.

The fast matches [2] are computationally cheap methods for reducing the number of word extensions which must be checked by the more accurate, but computationally expensive detailed matches.[1] (The fast matches may also be considered a predictive component for the detailed matches.) Top-N mode is achieved by delaying termination until N sentences have been output.

The stack itself is just a sorted list which supports the operations: pop the best entry, insert new entries according to their scores, and (in some implementations) discard the worst entry. The following must be contained in each stack entry: the stack score used to order the entries, the word history (path or theory identification), an output log-likelihood distribution, and an end-of-sentence flag. Since the time of exiting the word cannot be uniquely determined during a forward-decoder pass, the output log-likelihood as a function of time must be contained in each entry. This distribution is the input to the next word model. The end-of-sentence flag identifies the theories which are candidates to end the sentence.

This exposition will assume discrete observation hidden Markov model (HMM) word models [9, 10] with the

*This work was sponsored by the Defense Advanced Research Projects Agency.

[1] The following discussion concerns the basic stack decoder and therefore it will be assumed that the correct word will always be on the fast match list. This can be guaranteed by the scheme outlined in reference [2].

observation log-pdfs identified as the $B = b_{j,k}$ matrix, where j identifies the arc (transition between states) and k identifies the observation symbol. (This can be trivially extended to continuous observation, mixture, and tied-mixture HMM systems.) However, it should apply to any stochastic word model which outputs a log-likelihood. Similarly, a stochastic language model which outputs a partial sentence log-likelihood is assumed. An accept-reject language model will also work—its output is either zero or minus infinity.

The A* Stack Criterion

A key issue in the stack decoder is the scoring criterion. (All scores used here are log-likelihoods or log-probabilities.) If one uses the raw log-likelihoods as the stack score, a *uniform search* [7] will result. This search will result in a prohibitive amount of computation and a very large stack for any large-vocabulary high-perplexity speech recognition task because the score decreases rapidly with path length and thus short paths will be "carried along". A better scoring criterion is the *A* criterion* [7]. The A* criterion is the difference between the actual log-likelihood of reaching a point in time on a path and an upper bound on the log-likelihood of any path reaching that point in time:

$$\Lambda_i(t) = L_i(t) - \mathrm{ub}L(t) \tag{1}$$

where $\Lambda_i(t)$ is the scoring function, $L_i(t)$ is the output log-likelihood, t denotes time, i denotes the path (tree branch or left sentence fragment) and $\mathrm{ub}L(t)$ is an upper bound on $L_i(t)$. The stack entry score is

$$StSc_i = \max_t \Lambda_i(t). \tag{2}$$

If $\mathrm{ub}L(t) \geq \mathrm{lub}L(t)$, where $\mathrm{lub}L(t)$ is the least upper bound on L, the stack search is guaranteed to be *admissible*, i.e., the first output sentence will be the correct (highest log-likelihood) one [7] and, in addition, the following sentences in top-N mode will be output in log-likelihood order. In general, the closer $\mathrm{ub}L(t)$ is to $\mathrm{lub}L(t)$, the less computation. If $\mathrm{ub}L(t) = \mathrm{lub}L(t)$, the search is guaranteed to be *optimal* [7], i.e., a minimum number of stack entries will have to be popped and extended. If $\mathrm{ub}L(t)$ becomes less than $\mathrm{lub}L(t)$, longer paths will be favored excessively and the first output sentence may not have the highest log-likelihood, i.e., a search error may occur. (Note that $\mathrm{ub}L(t)$ is constant for any t and therefore does not affect the relative scores of the paths at any fixed time—it only affects the comparison of paths of differing lengths and the resultant order of path expansion.)

The basic problem is obtaining a good estimate of $\mathrm{ub}L(t)$ in a time-asynchronous decoder. (Note that $\mathrm{lub}L_{state}(t)$ over the states is easily computed in a time-synchronous decoder and that $\Lambda_{state}(t)$ is the value compared to the pruning threshold in a beam search [6].)

One estimate of $\mathrm{ub}L(t)$ is

$$\mathrm{ub}L(t) = -\alpha t. \tag{3}$$

where α is some constant greater than zero. This approach attempts to cancel out the average log-likelihood per time step. If α is too large, it will underestimate the bound and risk recognition errors. If α is small the search will be admissible, but will require an excessive amount of computation. (In fact, $\alpha = 0$ is the uniform search mentioned above.) An intermediate value of α will achieve a balance between the two extremes and produce the winner with reduced computation. Unfortunately no single value of α is optimum for all sentences or all parts of a single sentence. Thus a conservative value is generally chosen. One way of altering the tradeoff is to run in top-N mode and sort the output sentences by score. If α is slightly high, the sentences may be output out of order, but the sort provides a recovery mechanism. This scheme may also require additional computation to produce the additional output sentences.

The criterion of equation 3 can be improved by normalizing the observation probabilities by the A* criterion:

$$\mathrm{ub}L(t) = \sum_{\tau=1}^{t} \max_j b_{j,o_\tau} - \alpha t \tag{4}$$

where o_t is the observation at time t. This helps, but basic problems of equation 3 still remain. Both of these corrections can be precomputed by modifying the B matrix:

$$\overline{B_{j,k}} = B_{j,k} - \max_j b_{j,k} + \alpha. \tag{5}$$

This stack criterion allows estimation of the most-probable partial-path exit time. $\Lambda_i(t)$ now exhibits a peak whose location is the estimate of the exit time of the word. (The stack decoder only implements the forward decoder—finding the exact most-probable exit time requires information from the decode of the remainder of the sentence.) Therefore the estimated exit time is:

$$t_{max,i} = \operatorname*{argmax}_t \Lambda_i(t). \tag{6}$$

The Optimal A* Stack Decoder

The upper bounds of equations 3 and 4 are fixed approximations to the least upper bound and therefore force a tradeoff of computation and probability of search error. It is, however, possible to compute the *exact* least upper bound and so perform an optimal A* search. The primary difficulty is that only the "lub so far" can be computed, i.e., only the upper bound of the currently computed paths can be obtained. This creates two difficulties:

1. Since the estimate of the lub ($\hat{\mathrm{lub}}$) is changing, the stack order may change as a result of an update of $\hat{\mathrm{lub}}L(t)$.
2. A degeneracy in determining the best path may occur since the current bound may equal $L_i(t)$ for more than one i (path).

Problem 1 is easily cured by reevaluating score $StSc$ every time $\mathrm{lub}L(t)$ is updated and reorganizing the stack.

This is easily accomplished if the stack is stored as a heap[2]. The reorganization is accomplished by scanning the heap in addressing order and, at each entry, reevaluating the score, and if the score is greater than that of its parent, successively swapping the entry with its parent until the next parent has a higher score than the current entry. Once the first sentence is output, $\hat{\text{lub}}L(t)$ will be stable.

Problem 2 occurs because two or more theories may dominate different parts of the current upper bound. Thus all of these theories will have a score of zero. If the longest theory is extended, its descendents will also dominate the bound and will in turn be extended. This will, of course, result in search errors because the shorter theories will never have a chance to be extended. The cure is to extend the shortest theory. One could choose the shorter theory in the case of a tie or a simpler way of doing this is to use a slightly modified criterion:

$$\Lambda_i(t) = L_i(t) - \hat{\text{lub}}L(t) - \epsilon t \qquad (7)$$

where $\hat{\text{lub}}L(t)$ is the least upper bound so far and ϵ is a very small number greater than zero. (The value of ϵ need only be large enough to avoid being lost in the system quantization error and therefore the loss of optimality of the search criterion can be ignored.) The ϵt term serves as a tie-breaker in favor of the shorter theory in a manner which is compatible with equation 2. Note that this criterion completely accounts for all factors: the language model, the reduction of the log-likelihood as paths grow, any B matrix normalization (equation 5), and any effects due to the restrictions on the HMM state sequences in the word models. (In fact, this criterion makes $\Lambda_i(t)$ invariant to any fixed normalizations such as those of equation 5—a fact that will allow both the A* search and the above definition of t_{max} be used in the search linearizing algorithm described below.)

Reorganizing the stack immediately preceding each pop if the least upper bound has been updated, adding a miniscule length dependent penalty, and using the maximum of the normalized log-likelihood as the stack score for each theory results in a computationally-optimal admissible implementation of the stack decoder. Furthermore, it guarantees that when the stack decoder is used in top-N mode, the sentences will come out in decreasing log-likelihood order.

What is the advantage of the A* stack decoder over the time-synchronous beam search? Both, after all, end up using a least upper bound to control the search. The two lub's are different—the time synchronous search generally computes its lub over all states and the pruning will be affected by the location in the network in which the language-model log-likelihoods are added (at the beginning of the word, at the end, or spread out along the word model). In contrast, the stack decoder only computes its lub at the end of the words and all places for adding the language-model log-likelihoods are equivalent. The stack decoder lub can also be better (but never worse) than the time-synchronous lub because it is computed only for the word ends. (On the other hand, a time-synchronous decoder can prune parts of words—the stack decoder, as described here, treats words as indivisible units.) Finally, the effective pruning threshold for the A* stack is continuously adaptive—it only extends theories which have a chance of winning while the pruning threshold for a time-synchronous decoder is determined by the worst case (which may be rare). An unbounded stack has been assumed—stack size will be discussed later. (Note that typically only a relatively small number of theories are actually popped from the stack and extended—the majority are abandoned when decoder terminates.)

[2] A heap is a row-wise linearly-addressed binary-tree data structure, whose tree representation resembles a pyramid. The score of a parent is greater than or equal to the scores of its children and each parent is located in the row above its children. A parent always has a lower address than its child. The highest scoring entry resides at the root of the tree (top of the pyramid) which is stored in the first location of the array [5].

Linearizing the Search

The basic stack decoder is exponential in the length of the input because it is a tree search with the consequence that identical subtrees must be searched as many times as there are distinct histories. (This section will initially deal with the acoustic matching part of the decode and will therefore assume no language model—adding language models will be discussed once the acoustic issues have been described.) There is, however, a method for combining identical subtrees which depends only on the last word of the theory to be extended and t_{max} (equation 6). This limits the maximum number of theories to VT where V is the number of vocabulary words and T is the length of the input. Thus the search will be $O(T)$ or linear in the length of the input. This method involves a minor approximation for the lower likelihood of the two joined paths.

Given that:

1. the last word of the history for each of the two theories is the same
2. and the t_{max}'s of equation 6 are the same

the lower likelihood theory can be pruned because it, assuming the approximation to be correct, can never beat the higher likelihood path. (Entensions to top-N will be given later.) This is true because probability as a function of time of transitioning between the the left word (last word of the history for the theory) and the next word is only a very weak function of words preceding the left word. (Any two words having the same acoustic model are considered equal here—so if, for instance, stress is being modeled, then the stressed and unstressed word models are considered different.) The assumption that the weak function is not a function of the words preceding the left word is the approximation. Obviously, this approximation is more accurate for longer left words or can be made more accurate if left word groups are used as the matching context.

This algorithm is easily implemented in the stack decoder. For any new theory about to be inserted onto the stack (comparisons for a match are made by the above two criteria):

1. If the new theory matches a theory on a list of past theories popped from the stack, discard it. It, by definition, has a lower likelihood than the theory on the list.

2. If the new theory matches a theory currently on the stack, save the more likely theory, discard the less likely theory and, if necessary, reorganize the stack.

This algorithm saves both stack space and computation. (It is the analog of a path join in a time-synchronous decoder.)

To extend this theory to a top-N decoder, instead of discarding the less likely theory, attach a link from the more likely theory to the less likely theory, record the difference in the likelihoods, and store the less likely theory off the stack. Whenever an end-of-sentence theory is popped from the stack, follow the links and:

1. apply the likelihood differences to reconstruct the likelihoods,

2. reconstruct the word sequences, and

3. place the word sequences on a secondary heap ordered by their likelihoods.

Once reconstructed theories have been placed on the secondary heap, the pop operation must check both the stack and the secondary heap for the most likely theory to pop. This algorithm will be exact for theories which stay on the stack, but the likelihoods will be approximate for the reconstructed theories from the secondary heap.

This algorithm can also be extended to include language models. Unigram and bigram language models [1] are trivial—the unigram model is not a function of the left context and the bigram model is a function of the same word which was used in determining a match. Thus both can be inserted directly into the linearizing algorithm without modification. A trigram language model [1] would insert directly if the left matching context was two words rather than one word.

Language models with left contexts too long to be efficiently incorporated directly into the linearizing algorithm could be handled by by caching the acoustic matches according to the above path joining criterion. Whenever a cache match is found, use the cached next word output likelihoods to avoid recomputation. (Note that the acoustic detailed match must use only the word list from the acoustic fast match. Otherwise the cached set of word extensions will be limited by the language-model context in effect at the time of caching. The language-model fast match can then be applied after the acoustic detailed match.) This will make the acoustic computation linear, but will not reduce the language-model computation. (Some language-model algorithms may also be able to use similar mechanisms to avoid repeating computations.) This approach extends trivially to top-N mode without the use of the links or the secondary heap.

A final method of using language models with a long left-context dependency is to simply operate the linearized stack decoder with a bigram, unigram or no language model, and output the top-N sentences to a decoupled language-model analyzer. This is the decoupled mode of reference [8].

Limiting the Stack Size

One difficulty of the stack decoder is the stack size. The A* search algorithm reduces the stack size, but it can still grow exponentially. The first linearizing algorithm places an upper bound of VT (or V^2T for the trigram language model) on the stack size, but the secondary heap can grow exponentially. To minimize these problems, a pruning threshold can be applied to the stack insertion operations and the secondary heap generation. Any theory whose $StSc$ from equation 2 is below a threshold is discarded. (Since $\hat{\text{lub}}L(t)$ can only increase as the decoder operates, no discarded theory would be accepted by a later value of $StSc$. Note also that the stack reorganizing operation can also discard any entries that fall below the threshold according the the new values of $StSc$.) While this pruning threshold may seem similar to the time-synchronous decoder pruning threshold, a conservative value only increases the stack size, but not the computation in the basic stack decoder. A conservative value would also increase the link tracing computation and the secondary heap size, but the computation is minor compared to the basic stack computation.

A second method of limiting the stack size is, a-priori choosing some reasonable size and when it is about to overflow, discard the worst theory. This, in effect, dynamically chooses a pruning threshold. (It can be viewed as a fail-soft stack overflow.) The standard heap does not support efficient location of the worst theory. (Locating the worst theory in a full heap requires searching the bottom row or about half the heap.) It is possible to modify a heap from its usual pyramid shape to a diamond shape by attaching an upside down pyramid to the bottom of the first pyramid. This structure would have the best theory at the top and the worst theory at the bottom. This would complicate the stack operation somewhat, but it would probably be as efficient as most other data structures.

Conclusion

Two algorithms have been presented for accelerating the operation of a stack decoder. The first is a method for computing the true least upper bound so that an optimal admissible A* search can be performed. The second is a set of methods for linearizing the computation required by a stack decoder. The A* search has been

implemented in a continuous speech recognizer simulator and has demonstrated a significant speedup. The linearizing algorithm has been partially implemented in the simulator and has also shown significant computational savings.

Addendum

Jim Baker conjectured that the optimal A* search as described above might not be admissible when a language model is used due to an interaction between the acoustic and the language model likelihoods [3] which can prevent $\hat{\text{lub}}L(t)$ from becoming the true lub. Such a loss of admissibility can result in the sentences being output out of likelihood order. This was tested by running the simulator in top-N mode with and without a language model. The test used 600 sentences from the DARPA Resource Management task and the word-pair language model. No recognition errors (the most likely sentence was not the first output) were observed for the no language model case and two errors were observed in the word-pair language model case. This verifies Baker's conjecture, but suggests that the problem may be relatively rare. It also offers empirical evidence that the no language model case is admissible.

References

[1] L. R. Bahl, F. Jelinek, and R. L. Mercer, "A Maximum Likelihood Approach to Continuous Speech Recognition," IEEE Trans. Pattern Analysis and Machine Intelligence, PAMI-5, March 1983.

[2] L. Bahl, P. S. Gopalakrishnam, D. Kanevsky, D. Nahamoo, "Matrix Fast Match: A Fast Method for Identifying a Short List of Candidate Words for Decoding," ICASSP 89, Glasgow, May 1989.

[3] J. K. Baker, personal communication, 25 June 1990.

[4] F. Jelinek, "A Fast Sequential Decoding Algorithm Using a Stack," IBM J. Res. Develop., vol. 13, November 1969.

[5] D. E. Knuth, "The Art of Computer Programming: Sorting and Searching,", Vol. 3., Addison-Wesly, Menlo Park, California, 1973.

[6] B. T. Lowerre, "The HARPY Speech Recognition System," PhD thesis, Computer Science Department, Carnegie Mellon University, April 1976.

[7] N. J. Nilsson, "Problem-Solving Methods of Artificial Intelligence," McGraw-Hill, New York, 1971.

[8] D. B. Paul, "A CSR-NL Interface Specification," Proceedings October, 1989 DARPA Speech and Natural Language Workshop, Morgan Kaufmann Publishers, October, 1989.

[9] D. B. Paul, "Speech Recognition using Hidden Markov Models," Lincoln Laboratory Journal, Vol. 3, no. 1, Spring 1990.

[10] A. B. Poritz, "Hidden Markov Models: A Guided Tour," Proc. ICASSP 88, April 1988.