Exact Alpha-Beta Computation in Logarithmic Space with Application to MAP Word Graph Construction

Geoffrey Zweig and Mukund Padmanabhan
IBM T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598
{gzweig,mukund@watson.ibm.com}

Abstract

The classical dynamic programming recursions for the forwards-backwards and Viterbi HMM algorithms are linear in the number of time frames being processed. Adapting the method of [8] to the context of speech recognition, this paper uses a recursive divide-and-conquer algorithm to reduce the space requirement to logarithmic in the number of frames. With this procedure, it is possible to do exact computations for observation sequences of essentially arbitrary length. The procedure works by manipulating a stack of alpha vectors, and by using sparse vectors, the space savings can be combined with those of traditional pruning techniques. We apply this technique to MAP lattice construction, and present the first results in the literature for that technique. We find that it is an effective way of creating word lattices, and that doing the exact computations enabled by the log-space technique results in lower word error rates than space saving via traditional pruning.

1 Introduction

Hidden Markov models provide the cornerstone of most speech recognition technology, and are underpinned by two fundamental algorithms: the computation of posterior state occupancy probabilities, and the computation of the best (Viterbi) path through an HMM graph [6]. These algorithms are useful because they elegantly use dynamic programming to reduce the computation time from exponential in the number of time frames to linear. However, the amount of space that is required is still linear in the number of frames (N), and linear in the number of states (S); i.e. it is $O(SN)$. Therefore, in the past, it has been infeasible to exactly compute best paths or posterior state occupancy probabilities for many HMM graphs of relevance to LVCSR tasks. For example, if one expresses a trigram language model as an FSM graph [7] and expands it to the phonetic state level, it can easily comprise millions of states. For an utterance several thousand frames long, finding the best path with straight dynamic programming is infeasible because of the gigabytes of space required.

The conventional method for overcoming the space problem is to perform pruning: instead of considering all the states in the graph at each time frame, a set of live states is maintained. Only their successors are computed for the following time frame, and only the best of these are added to the set of live states at that time. Although this has been successfully used in many applications, it has two drawbacks: first, it is inexact; and secondly, the maintenance, propagation, and pruning of the sets of live states adds complexity and indirection to the dynamic programming code.

It has recently been observed [8] that it is in fact possible to organize a large class of computations whose space requirements are normally linear in time so that instead they are logarithmic in time. In this paper, we demonstrate the use of the log-space algorithm in speech recognition, and use it both to do exact alpha-beta computation on a larger scale than has been reported in the past, and to do approximate computations in less space than has previously been required.

The specific task to which we apply the method is MAP word-lattice generation [1]. This technique, which has not to our knowledge been previously tested, uses posterior state occupancy probabilities to compute posterior word occupancy probabilities, and from these it constructs a word lattice. In our experiments, we find that the MAP lattice technique is effective and robust. We further find that doing the exact computation produced lower error rates than pruning, and that the conventional method of computing alpha-beta quantities required an exorbitant amount of space, even with pruning.

The remainder of the paper is organized as follows. In section 2 we present the log-space algorithm and discuss its space and time requirements. In section 3, we review the MAP lattice generation technique, and some simple modifications for producing small, accu-

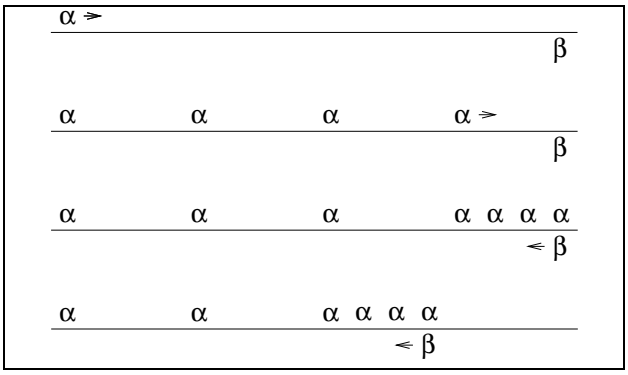

Figure 1: An example of alpha-beta computation in less than linear space. There are N time frames. Initially, the alphas and betas of the first and last frames are know. Then the alphas are computed for subsequent time frames, but only stored at intervals of $\sqrt{N}$. For each of these intervals starting with the rightmost, the alphas are computed and stored for the interior of the interval, and the beta vector is receded in time and combined with the stored alphas. Space is reclaimed after each interval is processed. The maximum amount of memory required is shown in the third line of this figure, and is $2S\sqrt{N}$.

rate lattices. Experimental results are presented in section 4, followed by a conclusion in section 5.

2 LogSpace Algorithm

Recall that for an HMM with states $\{\mathcal{Q}\}$, transition matrix a_{ij}, observation distributions $b_j(x)$, and observation sequence $\mathbf{O}$, the quantity $\alpha_t(i)$ is defined as $P(o_1, \ldots, o_t, q_t = i)$, and the quantity $\beta_t(i)$ is defined as $P(o_{t+1}, \ldots, o_T | q_t = i)$. The posterior probability of being in state i at time t is $\frac{\alpha_t(i)\beta_t(i)}{\sum_j \alpha_t(j)\beta_t(j)}$. Simple recursions allow the alpha and beta quantities to be computed:

$$\alpha_{t+1}(j) = \sum_i \alpha_i(t) a_{ij} b_j(o_{t+1})$$

$$\beta_t(i) = \sum_j a_{ij} b_j(t+1) \beta_j(t+1)$$

In the classical dynamic programming procedure, the alpha vectors are computed and stored for each frame in the utterance, starting from a base case at $t = 0$ and moving forwards in time. The beta vectors are then computed for each frame starting from a base case at $t = N$ and moving backwards in time. Since a beta vector can be immediately combined with the corresponding alpha vector as soon as it is computed (for example, to extract posterior occupancy probabilities), there is no need to store the beta vectors.

```
process(l, α, r, β)
    if r − l < threshold
        advance αs to r and store
        recede βs to l − 1 and combine with αs
        reclaim space used to store αs
        return βs at l − 1
    else
        break into k parts with boundaries left[i], right[i]
        advance αs to r storing at boundaries left[i]
        for s=k downto 1
            β = process(left[s], α[s], right[s], β)
    end
end
```

Figure 2: Pseudocode for logarithmic-space alpha-beta computation. On entry, the αs for every state are known at l, and the βs are known at r. The procedure returns the betas at $l - 1$. The example of Fig. 1 results when $k = \sqrt{N}$ and $threshold = \sqrt{N}$.

The key insight to the log-space algorithm is that by storing alpha vectors only at strategic intervals, and recomputing on demand those alpha vectors that are not stored, the space required can be drastically reduced at the price of a relatively small amount of additional computation. A simple example of this is presented in Figure 1. In this case, the N frames are broken into $\sqrt{N}$ intervals of $\sqrt{N}$ frames each, to achieve a memory usage of $O(S\sqrt{N})$.

By generalizing this strategy to one in which blocks of frames are recursively broken into k sub-blocks, a space usage of $O(kS \log_k N)$ is achieved, and a runtime of $O(SN \log_k N)$. The general algorithm is presented in Figure 2. It is important to stress that this generalizes the example of Figure 1 by its recursive nature: rather than terminating after just one k-way split, the procedure recurses until a threshold number of frames are present in a block, in the extreme case just 1. Note that the alpha and beta vectors may themselves be sparse representations of only those states with some appreciable probability, and therefore the algorithm can be combined with standard pruning techniques in a straightforward way. The space requirements are minimized when $k = e$ i.e. $k = 2.71\ldots$; in practice this means 3-way splits.

3 MAP Lattice Algorithm

The basic concept of a MAP word lattice was presented in [1], and is exteremly simple. The procedure uses word-internal acoustic context, and a bigram language model, and has two main steps: first it generates a list of likely words and their approximate time spans; and secondly it connects overlapping words to form a lattice. We now describe each step in detail.

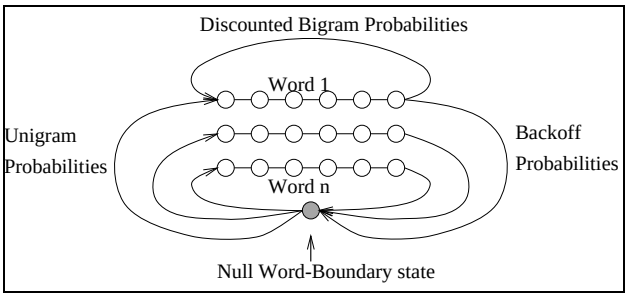

Figure 3: HMM structure used to generate MAP lattices. This HMM uses word internal acoustic context and the inter-word transition arcs encode a Kneser-Ney bigram language model. The circles within words represent phonetic states, and self-loops are omitted.

3.1 Word Trace Generation

The construction of a MAP lattice is based on the assumption that utterances are produced by an HMM with a structure as shown in Figure 3. Each pronunciation variant in the vocabulary appears as a linear sequence of phones, and the structure of this model permits the use of word-internal context dependent phones. In our work, we used a bigram language model with modified Kneser-Ney smoothing [2, 3]; this factors naturally as shown in Figure 3. There is an arc from the end of each word to a null word-boundary state, and this arc has a transition probability equal to the back-off probability for the word. From the word-boundary state, there is an arc to the beginning of each word, labeled with the unigram probability. For word pairs for which there is a direct bigram probability, we introduce an arc from the end of the first word to the beginning of the second, and this arc has a transition probability equal to the discounted bigram probability.

The MAP lattice is constructed by computing the posterior state occupancy probabilities for each state at each time, and then computing posterior word occupancy probabilities $P_t(W)$ by summing over all the states interior to each word. That is, if $\mathcal{W}_i$ is the set of states in word W_i, we compute $\sum_{s \in \mathcal{W}_i} \frac{\alpha_t(s)\beta_t(s)}{P(\mathbf{O})}$ at each time frame. We then keep track of the N likeliest words at each frame, and output these as a first step in the processing.

Note that a word will be on the list of likeliest words for a period of time, and then fall off that list. Thus the output of the first step is essentially a set of word traces, as illustrated in Figure 4. The horizontal axis is time, and the vertical axis ranges over all the word pronunciations.

3.2 Word Trace Connection

The next step is to connect the word traces into a lattice. Many connection schemes are possible, but we have found the following simple strategy to be quite effective. It requires that one more quantity be computed as the word traces are generated: the temporal midpoint of each trace as computed from the first moment of its posterior probability: $\frac{\sum_{t=start}^{t=end} tP_t(W)}{\sum_{t=start}^{t=end} t}$. To construct an actual lattice, we add a connection from the end of one word to the beginning of another if the two overlap, and the midpoint of the second is to the right of the midpoint of the first. This is illustrated at the bottom of Figure 4.

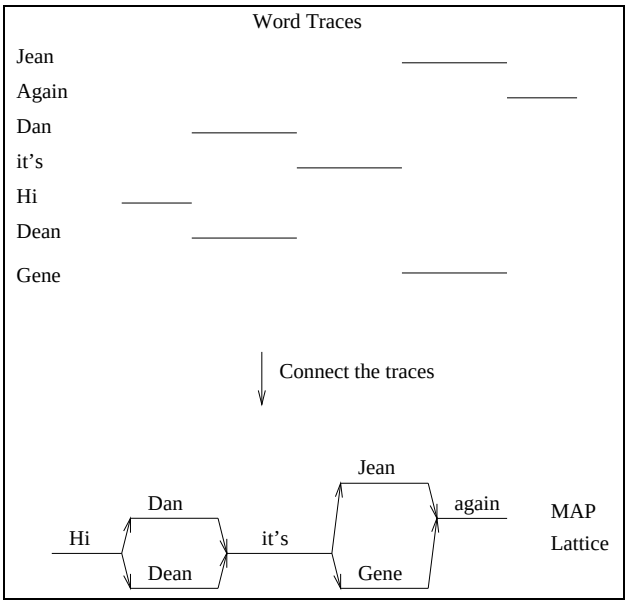

Figure 4: Word traces produced by the MAP lattice HMM, and their connection into a word lattice. In reality, since the N-best words at each frame are output, a vertical line should intersect a constant number of word traces; for visual simplicity, we have simplified the picture.

4 Experiments

The results presented here address two issues: first, what is the baseline accuracy that can be expected from a MAP lattice system, and secondly, what benefit can be expected from using the log-space algorithm to compute quantities exactly, as opposed to computing them with conventional pruning techniques. To address these issues, we experimented with a voicemail transcription task [5]. This is a large vocabulary continuous speech telephony task, and the HMM graph we used had about 16,000 word pronunciations and 300,000 states.

The experimental results are presented in Table 1, which shows the memory used in storing alpha and beta vectors, lattice size, and lattice word-error-rate for the basic MAP lattice strategy, and with two dif-

Algorithm	RAM (MB)	Link Density	WER
LogSpace	70.6	4960	6.76 %
FS-2.5k	84.7	4020	14.5
FS-5k	169	4540	11.0
FS-10k	339	4810	8.70
FS-20k	678	4910	7.23
BEAM-60	70.6	3620	12.2
BEAM-70	184	4420	9.09
BEAM-80	429	4740	8.35
BEAM-90	887	4910	7.06

Table 1: Lattice oracle word error rates with exact computation (LogSpace) and two kinds of pruning. For FS and BEAM, the pruning threshold is progressively less severe proceeding down the table.

ferent kinds of alpha-beta pruning. The first kind (FS) keeps a fixed number of states alive on the forward pass, and only computes the betas for those states. The second kind (BEAM) keeps all the states within a threshold of the maximum on the forward pass. The log-space technique produced the most accurate lattices with the least memory.

In Table 1, the word error rate reported is the oracle error rate for the lattice; i.e. the error rate of the single path through the lattice that has the smallest edit distance from the reference script. The link density is the ratio of the number of links between words in the lattice to the number of words in the reference script. The log-space algorithm was run with 3-way splits, and the recursion terminated for blocks of nine frames or fewer. The lattices were constructed from the traces of the 100 likeliest words at each time frame.

4.1 Lattice Pruning

For some applications, for example expanding lattices to trigram context, it is desirable to have lattices that are smaller than those produced by the basic MAP lattice technique. To do this, we developed a simple technique for reducing the lattice size, with relatively little affect on the error rate. To prune the lattices, we recalculate the alphas and betas and compute the posterior probability of transitioning along the arcs that connect word-ends. That is, if i is the last state in one word occurrence and j is the first state in a successor, we compute $P(q_t = i, q_{t+1} = j|\mathbf{O}) = \frac{\alpha_t(\mathbf{i})\beta_{t+1}(\mathbf{j})\mathbf{a_{ij}}\mathbf{b_j}(\mathbf{o_{t+1}})}{\mathbf{P(O)}}$. This is the posterior probability of being in state i at time t and in state j at time $t+1$, and transitioning between the words at an intermediate time. For each link between words, we sum this quantity from $t = 0$ to $t = N$ to get the total probability that the two words occurred sequentially; we then discard the links with the lowest posteriors. As in [4], we have found that over 95% of the links can be removed without a major loss of accuracy. In our pruned lattices the average indegree is reduced from over 70 to a little under 4, and the error rate is about 11%.

5 Conclusion

The log-space recursion is a simple way of enabling exact computations for HMMs with very long observation streams. Alternatively, it can be combined with traditional pruning to achieve an exponential decrease the overall memory requirements.

We have tested the log-space technique by constructing MAP lattices, both with exact computations, and with pruning, and find that more accurate results are produced in less space with the log-space technique. Achieving comparable results with traditional pruning required a prohibitive amount of space.

References

[1] F. Jelinek. *Statistical Methods for Speech Recognition.* The MIT Press, 1997.

[2] R. Kneser and H. Ney. Improved Backing-off for n-gram Language Modeling. *Proc. of ICASSP'95.* 1995.

[3] S.F. Chen and J. Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. Center for Research in Computing Technology, Harvard University, 1998.

[4] L. Mangu and E. Brill. Lattice Compression in the Consensual Post-Processing Framework. *Proc. of SCI/ISAS*, Orlando, Florida, 1999.

[5] M. Padmanabhan, G. Saon, S. Basu, J. Huang and G. Zweig. Recent improvements in voicemail transcription. *Proc. of EUROSPEECH'99*, Budapest, Hungary, 1999.

[6] L. Rabiner and B.H. Juang. *Fundamentals of Speech Recognition.* Prentice Hall, 1993.

[7] M. Mohri and M. Riley. Integrated Context-Dependent Networks in Very Large Vocabulary Speech Recognition. *Proc. of EUROSPEECH'99*, Budapest, Hungary, 1999.

[8] J. Binder, K. Murphy and S. Russell. Space-Efficient Inference in Dynamic Probabilistic Networks. *Proc. 15th Int'l. Joint Conf. on AI*, 1997.

[9] S. Ortmanns and H. Ney. A Word Graph Algorithm for Large Vocabulary Continuous Speech Recognition. *Computer Speech and Language*, (11) 1997.

[10] N. Deshmukh, A. Ganapathiraju, and J. Picone. Hierarchical Search for Large-Vocabulary Conversational Speech Recognition. *IEEE Signal Processing Magazine*, V.16 n.5, 1999.