

EWAVES: AN EFFICIENT DECODING ALGORITHM FOR LEXICAL TREE BASED SPEECH RECOGNITION

Patrick Nguyen, Luca Rigazio, Jean-Claude Junqua

Panasonic Technologies Inc. / Speech Technology Laboratory
3888 State Street, Suite 202, Santa Barbara, CA 93105, U.S.A.
email: {nguyen, rigazio, jcj}@stl.research.panasonic.com

ABSTRACT

We present an optimized implementation of the Viterbi algorithm suitable for small to large vocabulary, and isolated or continuous speech recognition. The Viterbi algorithm is certainly the most popular dynamic programming algorithm used in speech recognition. In this paper we propose a new algorithm that outperforms the Viterbi algorithm in term of complexity and of memory requirements. It is based on the assumption of strictly left to right models and explores the lexical tree in an optimal way, such that book-keeping computation is minimized. The tree is encoded such that children of a node are placed contiguously and in increasing order of memory heap so that the proposed algorithm also optimizes cache usage. Even though the algorithm is asymptotically two times faster than the conventional Viterbi algorithm, in our experiments we measured an improvement of at least three.

1. OVERVIEW

The core of a speech recognition system lies in the search algorithm. Its speed will be the prominent factor that determines the overall speed of the recognition system. It is widely understood that the organization of the search space for the dynamic programming alignment is a key factor. However, it is commonplace to keep linked lists or hash tables to maintain the list of *active hypotheses*, or list of so-called *tokens*. Storing and consulting items from these data structures is called *book-keeping* of active hypotheses and in most systems the processor spends a non-negligible time performing this task. Typically, at the end of each frame, the list is sorted in decreasing order of score. Finding a hypothesis is usually solved through the use of a hash table or a linear search [1].

In this paper we propose an algorithm that reduces the book-keeping to a minimum: merging two sorted linked lists. Insertion of a hypothesis as well as lookup have a cost $O(1)$ (constant cost).

Our algorithm can be used for isolated word recognition or as a first pass fast match for continuous speech recognition. It could be extended to cross-word modeling.

The algorithm traverses the list of active nodes in a way that maximizes speed. For that purpose, we rely on the topology of our HMM models.

2. ACTIVE ENVELOPE

In this section we introduce the preliminaries to understand the algorithm, called *active envelope* or *ewaves* search. We start with lexical trees and the so-called dynamic programming equations. Then, we show how to manipulate the equations to obtain a faster implementation.

2.1. Lexical tree

Organization of possible words in a lexical tree [4], or pronunciation prefix tree is the first and perhaps most effective reorganization of the search space. The larger the vocabulary, the larger gain we can achieve. The fan out of the search space at the beginning of a word or the explosion in the number of hypotheses, if entries of the vocabulary are listed linearly, is equal to the number of words. On the other hand, if words beginning with the same prefix are hypothesized only once, the fan out is drastically reduced. It can be at most equal to the number of phonemes in the case of context independent models, and to the number of left states if we use context-dependent acoustic units. Globally a lexical tree implies a two to three fold reduction in the representation of the lexical items.

We build a static lexical tree based on states. That is, the prefix information is compared at the state-level. This allows us to take advantage of the tying that results from the decision trees. As we will see later, it also minimizes the overhead of book-keeping and traversal of hypotheses. However, for cross-word context dependent modeling we expand the tree dynamically to contain memory usage.

2.2. The Viterbi step

The Viterbi algorithm traverses the lexical tree in a time-synchronous fashion and applies the *dynamic programming equations* at each active node. Let us consider the state hypotheses. Let k be a node of the lexical tree, and $a_{j,k}$ the log probability of transition from node j to node k . Let $s_k(t)$ be its score at time t , and φ one of its parents, that is, a node such that $a_{\varphi k} \neq -\infty$. We further define the acoustic match, $d_k(t)$, to be the emission log probability of the current observation. Hence if the transition from a node φ to the node k is chosen at time t , then its score will be $s_k(t) = s_\varphi(t-1) + a_{\varphi,k} + d_k(t)$.

We state the dynamic programming equation for a node k :

$$s_k(t) = \max_{\varphi} \{s_{\varphi(t-1)} + a_{\varphi,k}\} + d_k(t) \quad (1)$$

We are confronted with a simple problem: given a node, we must have a list of its predecessors and their corresponding active hypotheses (if any) to apply the DP equations. We call it inheritance since each node must *inherit* its score from its parents. This is unnatural and also expensive in the framework of lexical trees. If we look at the equations in the other direction, then our design becomes clearer: for each node, we transmit or *bequeath* from the parent node to its children. However, merging hypotheses becomes an issue. If we keep active hypotheses in an array $\alpha(t)$ and the next $\alpha(t+1)$, whenever we activate a child of a node in $\alpha(t)$, we need to check whether it is already in $\alpha(t+1)$, which is expensive. Obviously, the book-keeping of these arrays is somewhat cumbersome. We refer to this method as the *array-swapping method*, because we apply $\alpha(t) \leftarrow \alpha(t+1)$ once all of $\alpha(t)$ were processed. In the next section we explain how to order $\alpha(\cdot)$ properly so as to lookup an hypothesis in $\alpha(t+1)$ at cost $O(1)$, and how to replace $\alpha(t)$ immediately after a node was processed, to avoid the use of another array.

2.3. Active envelope

We consider left-to-right models with no skip. Each state can loop unto itself and activate the state that is immediately to its right. The algorithm can be extended to models with skip states but with a penalty of $O(H)$, where H is the number of active hypotheses at any given time. Our special topology implies that $a_{j,k} = -\infty \forall k, j : j \neq k, j \neq k^*$ where k^* is the parent (ie state on the left) of node k .

Clearly, any given node in the lexical tree can only be activated by its only parent. Define the depth of the node in the lexical tree as the number of states on its left, and a column of the lexical tree as the set of nodes of the same depth. For each column, define and fix an arbitrary order relation on the nodes, such that, if n is a node with parent n^* , and similarly k and k^* , $k^* < n^*$ implies $k < n$. Since all nodes of a given depth in the lexical tree can be processed in an almost arbitrary order, we choose the traversing sequence that maximizes the performance of the memory cache, that is, in increasing order of the heap. Thus we choose the convenient pointer comparison as the order relation. Let the list of active nodes or *active envelope* for that level be the set of nodes with non trivial score. We traverse the active envelope in the order as shown in figure 1. We named it Z-traversal because we traverse nodes in an increasing order within a column, but process columns in reverse order.

For best performance the implementation uses a single linked list. We insert nodes in the active envelope in increasing order, and thus by induction no additional step to sort the envelope is required. Nodes in the active envelope are traversed exactly twice: one for activation (see next section), and one for insertion purposes only. Also, accessing children hypotheses is combined with insertion. The total cost for traversal, insertion and lookup and is linear ($O(H)$).

We proceed as follows

1. Start from the deepest active list in the lexical tree.

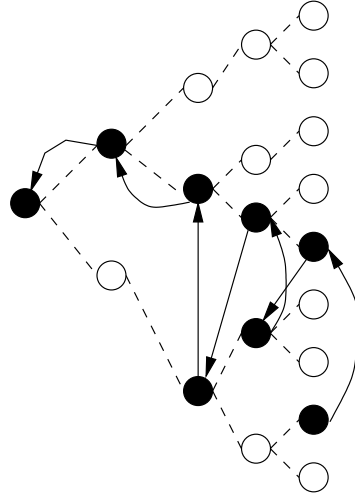


Figure 1: Z-traversal of the active envelope

2. Let n be the smallest ranked node in the active list of the children column.
3. Traverse the active list in increasing order.
4. For each child c of the current node k , if $n < c$, then increment n until that condition is false.
5. if $n = c$, then apply the DP equations; increment n .
6. if $n > c$, then simply link c before n .
7. Decrement the depth and process the parent column.

It is easy to show that the algorithm merges two sorted lists, namely, the existing sorted list with the list of activated children. To complete the algorithm we must show why this is sufficient to retain only *one* list at each level. We reduce the memory requirements by one half over the array-swapping method. It should be obvious by now nonetheless that we do not need to access hypotheses in the active list arbitrarily but only during the insertion process.

2.4. Traversing a hypothesis

Define the loop and incoming probabilities as $l_k = a_{k,k}$ and $i_k = a_{k^*,k}$. The score $s_k(\cdot)$ at time $t+1$ can be computed with

$$s_k(t+1) = \max \{s_k(t) + l_k, s_{k^*}(t) + i_k\} + d_k(t) \quad (2)$$

Note that we use t and $t+1$ instead of t and $t-1$ to denote a *forward* recursion instead of a *backwards* recursion: the ultimate goal is to compute score based on knowledge of children only (i.e. from k^* and not k) to avoid use of back-pointers (i.e. knowledge of father).

Now let us define the *topological score* $r_k(t) = s_k(t) - d_k(t)$ and the *partial topological score* $\tilde{r}(t) = s_k(t) + l$. The operations diagram is shown on figure 2.

It is more convenient to apply the acoustic match at the *beginning* of the iteration. Note that $\tilde{r}(t) = r(t)$ when k^* does

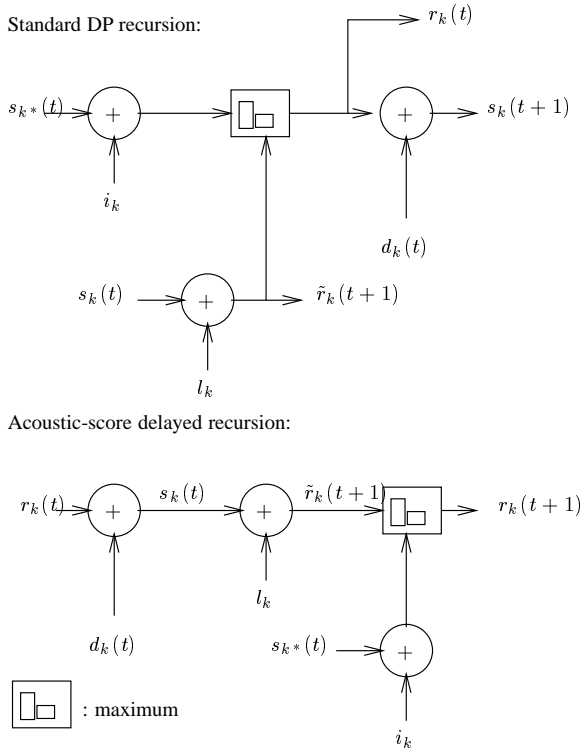


Figure 2: Block diagram of the DP equations

not belong to the active list. The cell k computes its own topological score and acoustic scores at each frame. We call this property *self-activation*. Each cell activates itself and then, for all of its children, if they have activated themselves already, we just need to bequeath our score to every children. Thus when traversing a cell in the active envelope, we perform the following operations:

- Compute score $s_k \leftarrow r_k + d_k$ (acoustic match)
- Bequeathal: for each child c , $r_c \leftarrow \max\{s_k + i_c, r_c\}$. The score field of the child is assumed to hold the partial score \tilde{r}
- Self-activation: $r_k \leftarrow \tilde{r}_k = r_k + l_k$. The score field now holds the partial topological score. If no score inheritance takes place then this is also the topological score for $t + 1$.

Bequeathal and self-activation can be inverted if we keep s_k and the next active node in variables: we can discard the node from the memory cache right after self-activation.

It is important that during the bequeathal process, a node has a direct access to its children. This is ensured by construction of our active envelope.

A small difference occurs when we apply the beam. Standard Viterbi algorithms compute and apply the beam on $s_k(t)$. We beam based on the topological scores, $r_k(t)$. This can be seen as a mini topological lookahead. HMM transition probabilities have however seldom proven of much influence in the past. Other than the application of the beam heuristic, how-

ever, the algorithm is equivalent to the standard Viterbi algorithm. Figure 3 illustrates the array swapping implementation: the beam is applied after summing the $d_k(\cdot)$.

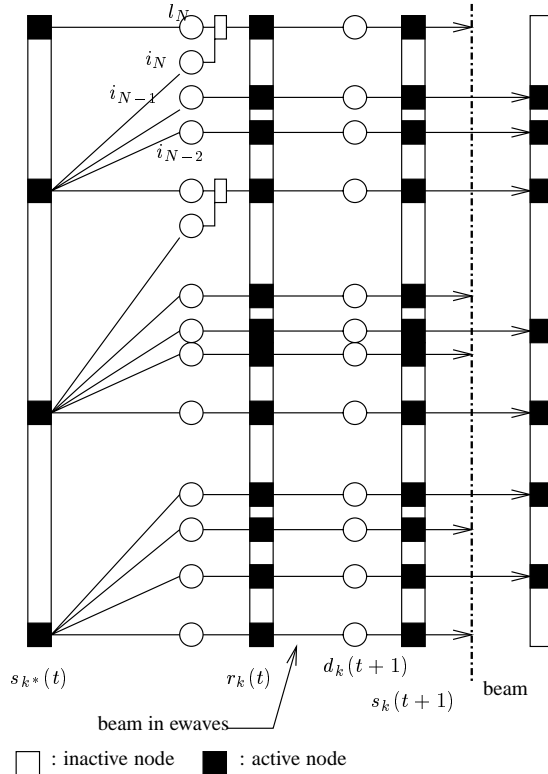


Figure 3: Array-swapping Viterbi with beam

2.5. Extension to continuous speech recognition

In continuous speech recognition, the processor must spend time on computation of the acoustic match, the search algorithm itself, and language modeling. Due to the late application of language model penalties, the search space must be split. We can no longer store the hypotheses embedded in the lexical tree. If word-internal context-dependent models are used, however, we need only one instance of the static lexical tree. Furthermore, unigram language models (LM) can be pre-factored. They are useful for unigram or bigram language model lookahead. In addition, a vast number of nodes in the lexical tree will share the same LM lookahead score. For example, a typical lexical tree for Wall Street Journal (WSJ), 20k words, non-verbalized pronunciation, with about 2k mixtures, will expand into a tree comprising about 200k nodes. Each node corresponds to a state in an HMM. The LM trees, on the other hand, have only 39k nodes. It is typical to factorize the small LM lookahead trees in the forward direction using DP alignment, because it allows for partial, on-demand factorization. We factorize the full tree backwards, starting from non-backoff bigrams.

Our representation is especially adequate for a tree fast

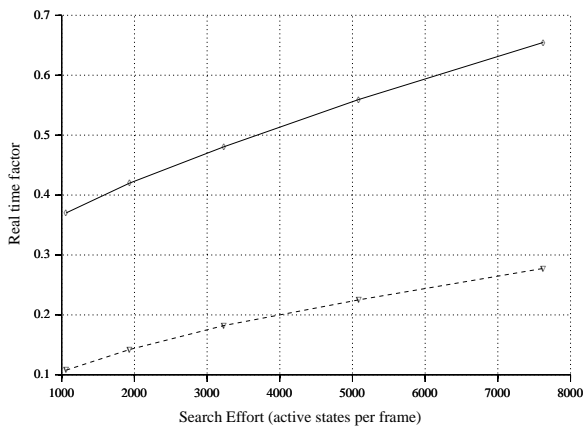


Figure 4: Overall speed of recognition with respect to the search effort

match, since the algorithm is very efficient at processing a large number of state hypotheses.

3. EXPERIMENTAL RESULTS

As far as recognition results are concerned, our algorithm diverges from the standard Viterbi only in the place where the beam is applied. Therefore, recognition performance are only marginally different. In our experiments, results are almost exactly the same. It is clear, however, that a faster search makes room for a wider beam or more complex algorithmic improvements.

We present benchmarks for the Voice PhoneBook database [7]. The database consists of isolated words spoken over a telephone channel. We used 9 static PLP coefficients augmented with their time derivative, and cepstral filtering. Our recognition system uses decision-tree clustered context-dependent models, with a total of 7131 mixture distributions which comprise a total of 19332 gaussian distributions. The Hidden Markov Models (HMM) are left-to-right with 3 emitting states. Our phoneme set has 55 items. There are about 1800 words in the decoding lexicon. We used phonological rules to generate multiple transcriptions. There were about 44k transcriptions in total. The static state-based lexical tree has 551435 nodes. The recognition accuracy is 91%.

Figure 4 compares the active envelope search with conventional Viterbi. On this task we improved speed by a factor of three. We measured the real-time factor (xRT) on an Intel Pentium III, 500 MHz machine running Linux. The figure shows the overall speed of recognition, including the frontend parameterization, acoustic match score computations, and the search itself. The baseline Viterbi algorithm uses a model-based lexical tree. The more efficient, state-based lexical tree used in the ewaves search reduces the amount of gaussian distance computations by 20%. The ewaves algorithm tends to be more profitable when used in conjunction with multiple transcriptions since it provides a more efficient way to explore the search space. On the other hand, a larger number of gaussian distributions may reduce the overall speed-up provided by the

search algorithm. It is clear that the overall gain will be dependent on the application.

4. CONCLUSION

Our method operates on three points

- Delay acoustic match scoring.
- Traverse from the deepest level in the lexical tree so that we can lookup and insert into the active envelope at a minimal cost.
- Replace inheritance by bequeathal (forward recursion instead of backwards recursion) so that we need only one active list.

Although not strictly required by the algorithm, a statically encoded lexical tree improves the performance even further.

In this paper, we have described an implementation of a lexical tree based Viterbi algorithm. We utilize the assumption of left-to-right topology and reorder the search space to minimize the book-keeping of hypotheses. Our new decoder performs three times faster than our older implementation on the Voice Phonebook database. With some modifications, the approach can be used with continuous speech recognizers.

5. REFERENCES

- [1] K. Demuynck, J. Duchateau, D. Van Compernelle, and P. Wambacq. An efficient search space representation for large vocabulary continuous speech recognition. *Speech Communication*, 30(1):37–54, January 2000.
- [2] Neeraj Deshmukh, Aravind Ganapathiraju, and Joseph Picone. Hierarchical Search for Large Vocabulary Conversational Speech Recognition. *IEEE Signal Processing Magazine*, 6(5):84–107, September 1999.
- [3] Hermann Ney and Stefan Ortmanns. Dynamic Programming Search for Continuous Speech Recognition. *IEEE Signal Processing Magazine*, 6(5):63–83, September 1999.
- [4] J. Odell. *The Use of Context in Large Vocabulary Speech Recognition*. PhD thesis, Cambridge University, 1995.
- [5] S. Ortmanns, A. Eiden, H. Ney, and N. Coenen. Language-model look-ahead for large vocabulary speech recognition. In *Proceedings of the Fourth European Conference on Speech Communication and Technology*, pages 2095–2098, Philadelphia, PA, October 1996.
- [6] S. Ortmanns, A. Eiden, H. Ney, and N. Coenen. Look-ahead techniques for fast beam search. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1783–1786, Munich, Germany, April 1997.
- [7] J. Pitrelli, C. Fong, S.H. Wong, J. R. Spitz, and H. C. Lueng. Phonebook: A phonetically-rich isolated-word telephone-speech database. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1767–1770, 1995.
- [8] Mosur K. Ravishankar. *Efficient Algorithms for Speech Recognition*. PhD thesis, Carnegie Mellon University, 1996.