CHAPTER 13

Large Vocabulary Search Algorithms

Chapter 12 discussed the basic search techniques for speech recognition. However, the search complexity for large-vocabulary speech recognition with high-order language models is still difficult to handle. In this chapter we describe efficient search techniques in the context of time-synchronous Viterbi beam search, which becomes the choice for most speech recognition systems because it is very efficient. We use Microsoft Whisper as our case study to illustrate the effectiveness of various search techniques. Most of the techniques discussed here can also be applied to stack decoding.

With the help of beam search, it is unnecessary to explore the entire search space or the entire trellis. Instead, only the promising search state-space needs to be explored. Please keep in mind the distinction between the implicit search graph specified by the grammar network and the explicit partial search graph that is actually constructed by the Viterbi beam search algorithm.

In this chapter we first introduce the most critical search organization for large-vocabulary speech recognition—tree lexicons. Tree lexicons significantly reduce potential search space, although they introduce many practical problems. In particular, we need to

address problems such as reentrant lexical trees, factored language model probabilities, subtree optimization, and subtree polymorphism.

Various other efficient techniques also are introduced. Most of these techniques aim for clever pruning with the hope of sparing the correct paths. For more effective pruning, different layers of beams are usually used. While fast match techniques described in Chapter 12 are typically required for stack decoding, similar concepts and techniques can be applied to Viterbi beam search. In practice, the look-ahead strategy is equally effective for Viterbi beam search.

Although it is always desirable to use all the knowledge sources (KSs) in the search algorithm, some are difficult to integrate into the left-to-right time-synchronous search framework. One alternative strategy is to first produce an ordered list of sentence hypotheses (a.k.a. *n-best list*), or a lattice of word hypotheses (a.k.a. *word lattice*) using relatively inexpensive KSs. More expensive KSs can be used to rescore the *n*-best list or the word lattice to obtain the refined result. Such a multipass strategy has been explored in many large-vocabulary speech recognition systems. Various algorithms to generate sufficient *n*-best lists or the word lattices are described in the section on multipass search strategies.

Most of the techniques described in this chapter rely on nonadmissible heuristics. Thus, it is critical to derive a framework to evaluate different search strategies and pruning parameters.

13.1. EFFICIENT MANIPULATION OF TREE LEXICON

The lexicon entry is the most critical component for large-vocabulary speech recognition, since the search space grows linearly along with increased linear vocabulary. Thus an efficient framework for handling large vocabulary undoubtedly becomes the most critical issue for efficient search performance.

13.1.1. Lexical Tree

The search space for *n*-gram discussed in Chapter 12 is organized based on a straightforward linear lexicon; i.e., each word is represented as a linear sequence of phonemes, independent of other words. For example, the phonetic similarity between the words *task* and *tasks* is not leveraged. In a large-vocabulary system, many words may share the same beginning phonemes. A tree structure is a natural representation for a large-vocabulary lexicon, as many phonemes can be shared to eliminate redundant acoustic evaluations. The lexical tree-based search is thus essential for building a real-time[1] large-vocabulary speech recognizer.

Figure 13.1 shows an example of such a lexical tree, where common beginning phonemes are shared. Each leaf corresponds to a word in the vocabulary. Please note that an

[1] The term *real-time* means the decoding process takes no longer than the duration of the speech. Since the decoding process can take place as soon as the speech starts, such a real-time decoder can provide real instantaneous responses after speakers finish talking.

extra null arc is used to form the leaf node for each word. This null arc has the following two functions:

1. When the pronunciation transcription of a word is a prefix of other ones, the null arc can function as one branch to end the word.
2. When there are homophones in the lexicon, the null arcs can function as linguistic branches to represent different words such as *two* and *to*.

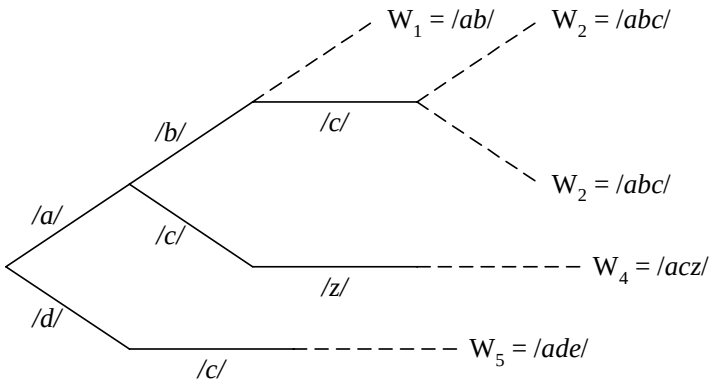

Figure 13.1 An example of a lexical tree, where each branch corresponds to a shared phoneme and the leaf corresponds to a word.

The advantage of using such a lexical tree representation is obvious: it can effectively reduce the state search space of the trellis. Ney et al. [32] reported that a lexical tree representation of a 12,306-word lexicon with only 43,000 phoneme arcs had a saving of factor of 2.5 over the linear lexicon with 100,800 phoneme arcs. Lexical trees are also referred to as *prefix trees*, since they are efficient representations of lexicons with sharing among lexical entries that have a common prefix. Table 13.1 shows the distribution of phoneme arcs for this 12,306-word lexical tree. As one can see, even in the fourth level the number of phoneme arcs is only about one-third of the total number of words in the lexicon.

Table 13.1 Distribution of the tree phoneme arcs and active tree phoneme arc for a 12,306-word lexicon using a lexical tree representation [32].

Level	1	2	3	4	5	6	≥ 7
Phoneme arcs	28	331	1511	3116	4380	4950	29.200
Average active arcs	23	233	485	470	329	178	206

The saving by using a lexical tree is substantial, because it not only results in considerable memory saving for representing state search space but also saves tremendous time by searching far fewer potential paths. Ney et al. [32] report that a tree organization of the lexicon reduces the total search effort by a factor of 7 over the linear lexicon organization. This is because the lion's share of hypotheses during a typical large-vocabulary search is on the

first and second phonemes of a word. Haeb-Umbach et al. [23] report that for a 12,306-word dictation task, 79% and 16% of the state hypotheses are in the first and second phonemes, when analyzing the distribution of the state hypotheses over the state position within a word. Obviously, the effect is caused by the ambiguities at the word boundaries. The lexical tree representation reduces that effort by evaluating common phonetic prefixes only once. Table 13.1 also shows the average number of active phoneme arcs in the layers of the lexical tree [32]. Based on this table, you can expect that the overall search cost is far less than the size of the vocabulary. This is the key reason why lexical tree search is widely used for large-vocabulary continuous speech recognition systems.

The lexical tree search requires a sophisticated implementation because of a fundamental deficiency—*a branch in a lexical tree representation does not correspond to a single word with the exception of branches ending in a leaf*. This deficiency translates to the fact that a unique word identity is not determined until a leaf of the tree is reached. This means that any decision about the word identity needs to be delayed until the leaf node is reached, which results in the following complexities.

- Unlike a linear lexicon, where the language model score can be applied when starting the acoustic search of a new word, the lexical tree representation has to delay the application of the language model probability until the leaf is reached. This may result in an increased search effort, because the pruning needs to be done on a less reliable measure, unless a factored language model is used as discussed in Section 13.1.3
- Because of the delay of language model contribution by one word, we need to keep a separate copy of an entire lexical tree for each unique language model history.

13.1.2. Multiple Copies of Pronunciation Trees

A simple lexical tree is sufficient if no language model or a unigram is used. This is because the decision at time *t* depends on the current word only. However, for higher-order *n*-gram models, the linguistic state cannot be determined locally. A tree copy is required for each language model state. For bigrams, a tree copy is required for each predecessor word. This may seem to be astonishing, because the potential search space is increased by the vocabulary size. Fortunately, experimental results show only a small number of tree copies are required, because efficient pruning can eliminate most of the unneeded ones. Ney el al [32] report that the search effort using bigrams is increased by only a factor of 2 over the unigram case. In general, when more detailed (better) acoustic and/or language models are used, the effect of a potentially increased search space is often compensated by a more focused beam search from the use of more accurate models. In other words, although the static search space might increase significantly by using more accurate models, the dynamic search space can be under control (sometimes even smaller), thanks to improved evaluation functions.

To deal with tree copies [19, 23, 37], you can create redundant subtrees. When copies of lexical trees are used to disambiguate active linguistic contexts, many of the active state

hypotheses correspond to the same redundant unigram state due to the postponed application of language models. To apply the language model sooner, and to eliminate redundant unigram state computations, a successor tree, T_i, can be created for each linguistic context *i*. T_i encodes the nonzero n-grams of the linguistic context *i* as an isomorphic subgraph of the unigram tree, T_0. Figure 13.2 shows the organization of such successor trees and unigram tree for bigram search. For each word *w* a successor tree, T_w is created with the set of successor words that have nonzero bigram probabilities. Suppose *u* is a successor of *w*; the bigram probability $P(u \mid w)$ is attached to the transition connecting the leaf corresponding to *u* in the successor tree T_w, with the root of the successor tree T_u. The unigram tree is a full-size lexical tree and is shared by all words as the back-off lexical tree. Each leaf of the unigram tree corresponds to one of |*V*| words in the vocabulary and is linked to the root of its bigram successor tree (T_u) by an arc with the corresponding unigram probability $P(u)$. The backoff weight, $\alpha(u)$, of predecessor *u* is attached to the arc which links the root of successor tree T_u to the root of the unigram tree.

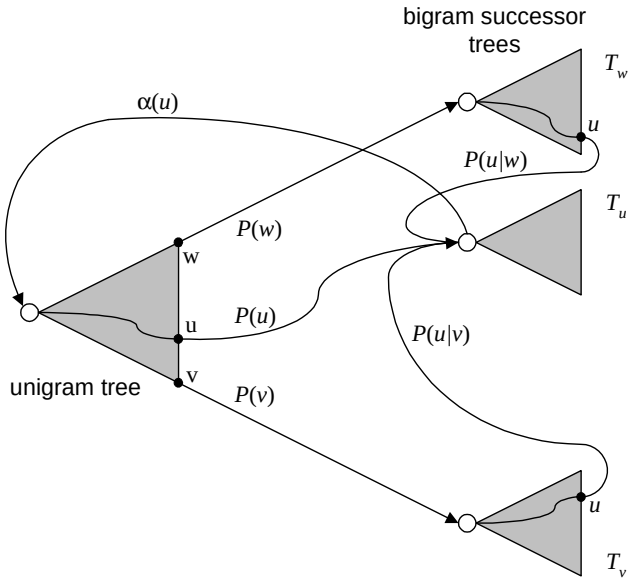

Figure 13.2 Successor trees and unigram trees for bigram search [13].

A careful search organization is required to avoid computational overhead and to guarantee a linear time complexity for exploring state hypotheses. In the following sections we describe techniques to achieve efficient lexical tree recognizers. These techniques include factorization of language model probabilities, tree optimization, and exploiting subtree dominance.

13.1.3. Factored Language Probabilities

As mentioned in Section 13.1.2, search is more efficient if a detailed knowledge source can be applied at an early stage. The idea of *factoring* the language model probabilities across the tree is one such example [4, 19]. When more than one word shares a phoneme arc, the upper bound of their probability can be associated to that arc.[2] The factorization can be applied to both the full lexical tree (unigram) and successor trees (bigram or other higher-order language models).

An unfactored tree only has language model probabilities attached to the leaf nodes, and all the internal nodes have probability 1.0. The procedure for factoring the probabilities across the tree computes the maximum of each node n in the tree according to Eq. (13.1). The tree can then be factored according to Eq. (13.2) so when you traverse the tree you can multiply $F^*(n)$ along the path to get the needed language probability.

$$P^*(n) = \max_{x \in child(n)} P(x) \tag{13.1}$$

$$F^*(n) = \frac{P^*(n)}{P^*(parent(n))} \tag{13.2}$$

An illustration of the factored probabilities is shown in Table 13.2. Using this lexicon, we create the tree depicted in Figure 13.3(a). In this figure the unlabeled internal nodes have a probability of 1.0. We distribute the probabilities according to Eq. (13.1) in Figure 13.3(b), which is factored according to Eq. (13.2), resulting in Figure 13.3(c).

Table 13.2 Sample probabilities P(w_j) and their pseudoword pronunciations [4].

w_j	Pronunciation	P(w_j)
w_0	/a b c/	0.1
w_1	/a b c/	0.4
w_2	/a c z/	0.3
w_3	/d e/	0.2

Using the upper bounds in the factoring algorithm is not an approximation, since the correct language model probabilities are calculated by the product of values traversed along each path from the root to the leaves. However, you should note that the probabilities of all the branches of a node do not sum to one. This can solved by replacing the upper-bound (max) function in Eq. (13.1) with the sum.

$$P^*(n) = \sum_{x \in child(n)} P(x) \tag{13.3}$$

[2] The choice of upper bound is because it is an admissible estimate of the path no matter which word will be chosen later.

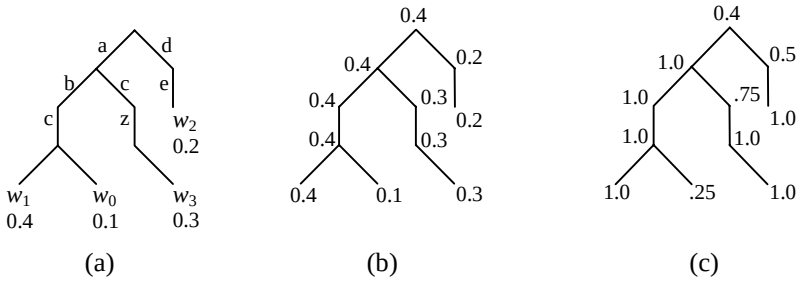

Figure 13.3 (a) Unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree $F^*(n)$ [4].

To guarantee that all the branches sum to one, Eq. (13.2) should also be replaced by the following equation:

$$F^*(n) = \frac{P^*(n)}{\sum_{x \in child(parent(n))} P^*(x)} \tag{13.4}$$

A new illustration of the distribution of LM probabilities by using sum instead of upper bound is shown in Figure 13.4. Experimental results have shown that the factoring method with either sum or upper bound has comparable search performance.

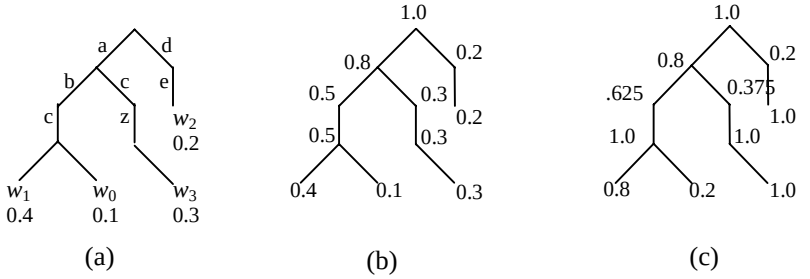

Figure 13.4 Using sum instead of upper bound when factoring tree, the corresponding (a) unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree with computed $F^*(n)$ [4].

One interesting observation is that the language model score can be regarded as a heuristic function to estimate the linguistic expectation of the current word to be searched. In a linear representation of the pronunciation lexicon, application of the linguistic expectation was straightforward, since each state is associated with a unique word. Therefore, given the context defined by the hypothesis under consideration, the expectation for the first phone of

word w_i is just $P(w_i \mid w_1^{i-1})$. After the first phone, the expectation for the rest of the phones becomes 1.0, since there is only one possible phone sequence when searching the word w_i. However, for the tree lexicon, it is necessary to compute $E(p_j \mid p_1^{j-1}, w_i^{i-1})$, the expectation of phone p_j given the phonetic prefix p_1^{j-1} and the linguistic context w_1^{i-1}. Let $\phi(j, w_k)$ denote the phonetic prefix of length j for w_k. Based on Eqs. (13.1) and (13.2), we can compute the expectation as:

$$E(p_j \mid p_1^{j-1}, w_1^{i-1}) = \frac{P(w_c \mid w_1^{i-1})}{P(w_p \mid w_1^{i-1})} \tag{13.5}$$

where $c = \arg\max_k (w_k \mid w_1^{i-1}, \phi(j, w_k) = p_1^j)$ and $p = \arg\max_k (w_k \mid w_1^{i-1}, \phi(j-1, w_k) = p_1^{j-1})$. Based on Eq. (13.5), an arbitrary *n*-gram model or even a stochastic context-free grammar can be factored accordingly.

13.1.3.1. Efficient Memory Organization of Factored Lexical Trees

A major drawback to the use of successor trees is the large memory overhead required to store the additional information that encodes the structure of the tree and the factored linguistic probabilities. For example, the 5.02 million bigrams in the 1994 NABN (North American Business News) model require 18.2 million nodes. Given a compact binary tree representation that uses 4 bytes of memory per node, 72.8 million bytes are required to store the predecessor-dependent lexical trees. Furthermore, this tree representation is not as amenable to data compression techniques as the linear bigram representation.

ALGORITHM 13.1 ENCODING THE LEXICAL SUCCESSOR TREES (LST)

For each linguistic context
1. Distribute the probabilities according to Eq. (13.1).
2. Factor the probabilities according to Eq. (13.2).
3. Perform a depth-first traversal of the LST and encode each leaf record,
 (a) the depth of the most recently visited node that is not a direct ancestor,
 (b) the probability,
 (c) the word identity.

The factored probability of successor trees can be encoded as efficiently as the *n*-gram model based on Algorithm 13.1, i.e., one *n*-gram record results in one constant-sized record. Step 3 is illustrated in Figure 13.5(b), where the heavy line ends at the most recently visited node that is not a direct ancestor. The encoding result is shown in Table 13.3.

Clearly the new data structure meets the requirements set forth, and, in fact, it only requires additional $\log(n)$ bits per record (n is the depth of the tree). These bits encode the common prefix length for each word. Naturally this requires some modification to the de-

coding procedure. In particular the decoder must scan a portion of the *n*-gram successor list in order to determine which tree nodes should be activated. Depending on the structure of the tree (which is determined by the acoustic model, the lexicon, and language model) the tree structure can be interpreted at runtime or cached for rapid access if memory is available.

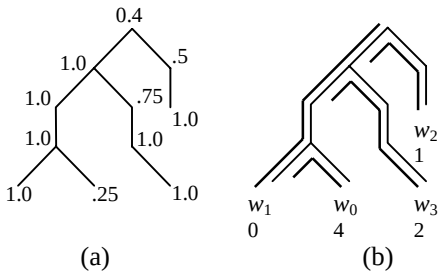

Figure 13.5 (a) Factored tree; (b) tree with common prefix-length annotation.

Table 13.3 Encoded successor lexical tree; each record corresponds to one augmented factored *n*-gram.

w_j	Depth	$F^*(w_j)$
w_1	0	0.4
w_0	4	0.25
w_3	2	0.75
w_2	1	0.5

13.1.4. Optimization of Lexical Trees

We now investigate ways to handle the huge search network formed by the multiple copies of lexical trees in different linguistic contexts. The factorization of lexical trees actually makes it easier to search. First, after the factorization of the language model, the intertree transitions shown in Figure 13.2 no longer have the language model scores attached because they are already applied completely before leaving the leaves. Moreover, as illustrated in Figure 13.3, many transitions toward the end of a single-word path now have an associated transition probability that is equal to 1. This observation implies that there could be many duplicated subtrees in the network. Those duplicated subtrees can then be merged to save both space and computation by eliminating redundant (unnecessary) state evaluation. Unlike pruning, this saving is based on the dynamic programming principle, without introducing any potential error.

13.1.4.1. Optimization of Finite State Network

One way to compress the lexical tree network is to use a similar algorithm for optimizing the number of states in a deterministic finite state automaton. The optimization algorithm is based on the *indistinguishable* property of states in a finite state automaton. Suppose that s_1 and s_2 are the initial states for automata T_1 and T_2, then s_1 and s_2 are said to be *indistinguishable* if the languages accepted by automata T_1 and T_2 are exactly the same. If we consider our lexical tree network as a finite state automaton, the symbol emitted from the transition arc includes not only the phoneme identity, but also the factorized language model probability.

The general set-partitioning algorithm [1] can be used for the reduction of finite state automata. The algorithm starts with an initial partition of the automaton states and iteratively refines the partition so that two states s_1 and s_2 are put in the same block B_i if and only if $f(s_1)$ and $f(s_2)$ are both in the same block B_j. For our purpose, $f(s_1)$ and $f(s_2)$ can be defined as the destination state given a phone symbol (in the factored trees, the pair <*phone*, *LM-probability*> can be used). Each time a block is partitioned, the smaller subblock is used for further partitioning. The algorithm stops when all the states that transit to some state in a particular block with arcs labeled with the same symbol are in the same block. When the algorithm halts, each block of the resulting partition is composed of *indistinguishable* states, and those states within each block can then be merged. The algorithm is guaranteed to find the automaton with the minimum number of states. The algorithm has a time complexity of $O(MN \log N)$, where M is the maximum number of branching (fan-out) factors in the lexical tree and N is the number of states in the original tree network.

Although the above algorithm can give optimal finite state networks in terms of number of states, such an optimized network may be difficult to maintain, because the original lexical tree structure could be destroyed and it may be troublesome to add any new word into the tree network [1].

13.1.4.2. Subtree Isomorphism

The finite state optimization algorithm described above does not take advantage of the tree structure of the finite state network, though it generates a network with a minimum number of states. Since our finite state network is a network of trees, the indistinguishablility property is actually the same as the definition of subtree isomorphism. Two subtrees are said to be *isomorphic* to each other if they can be made equivalent by permuting the successors. It should be straightforward to prove that two states are indistinguishable, if and only if their subtrees are isomorphic.

There are efficient algorithms [1] to detect whether two subtrees are isomorphic. For all possible pairs of states *u* and *v*, if the subtrees starting at *u* and *v*, $ST(u)$ and $ST(v)$, are isomorphic, *v* is merged into *u* and $ST(v)$ can be eliminated. Note that only internal nodes

need to be considered for subtree isomorphism check. The time complexity for this algorithm is $O(N^2)$ [1].

13.1.4.3. Sharing Tails

A *linear tail* in a lexical tree is defined as a subpath ending in a leaf and going through states with a unique successor. It is often referred as a *single-word subpath*. It can be proved that such a linear tail has unit probability attached to its arcs according to Eqs. (13.1) and (13.2). This is because LM probability factorization *pushes forward* the LM probability attached to the last arc of the linear tail, leaving arcs with unit probability. Since all the tails corresponding to the same word *w* in different successor trees are linked to the root of successor tree T_w,[3] the subtree starting from the first state of each linear tail is isomorphic to the subtree starting from one of the states forming the longest linear tail of *w*. A simple algorithm to take advantage of this share-tail topology can be employed to reduce the lexical tree network.

Figure 13.6 and Figure 13.7 show a lexical tree network before and after shared-tail optimization. For each word, only the longest linear tail is kept. All other tails can be removed by linking them to an appropriate state in the longest tail, as shown in Figure 13.7.

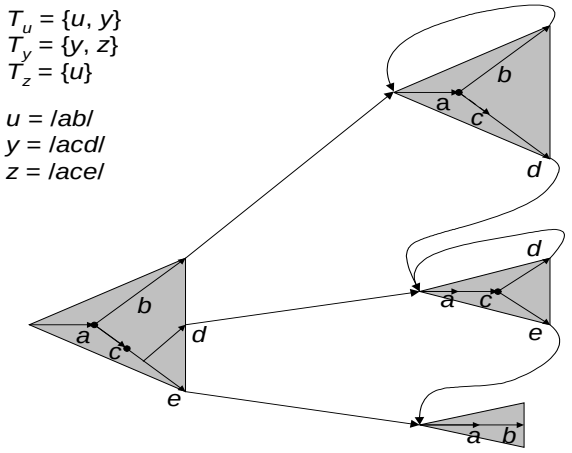

Figure 13.6 An example of a lexical tree network without shared-tail optimization [12]. The vocabulary includes three words, *u*, *v*, and *z*. T_u, T_v, and T_z are the successor trees for *u*, *v*, and *z*, respectively [13].

Shared-tail optimization is not global optimization, because it considers only some special topology optimization. However, there are some advantages associated with shared-tail optimization. First, in practice, duplicated linear tails account for most of the redundancy in lexical tree networks [12]. Moreover, shared-tail optimization has a nice property of maintaining the basic lexical tree structure for the optimized tree network.

[3] We assume bigram is used in the discussion of "sharing tails."

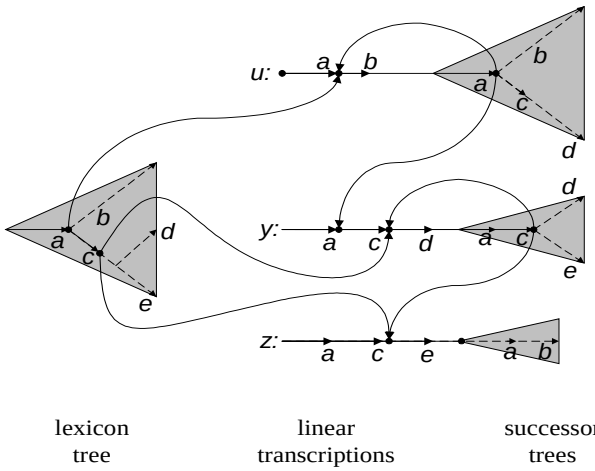

Figure 13.7 The lexical tree network in Figure 13.6 after shared-tail optimization [12].

13.1.5. Exploiting Subtree Polymorphism

The techniques of optimizing the network of successor lexical trees can only eliminate identical subtrees in the network. However, there are still many subtrees that have the same nodes and topology but with different language model scores attached to the arcs. The acoustic evaluation for those subtrees is unnecessarily duplicated. In this section we exploit *subtree dominance* for additional saving.

A subtree instance is *dominated* when the best outcome in that subtree is not better than the worst outcome in another instance of that subtree. The evaluation becomes redundant for the dominated subtree instance. Subtree isomorphism and shared-tail are cases of subtree dominance, but they require prearrangement of the lexical tree network as described in the previous section.

If we need to implement lexical tree search dynamically, the network optimization algorithms are not suitable. Although subtree dominance can be computed using minimax search [35] during runtime, this requires that information regarding subtree isomorphism be available for all corresponding pairs of states for each successor tree T_w. Unfortunately it is not practical in terms of either computation or space.

In place of computing strict subtree dominance, a *polymorphic* linguistic context assignment to reduce redundancy is employed by estimating subtree dominance based on local information and ignoring the subgraph isomorphism problem. Polymorphic context assignment involves keeping a single copy of the lexical tree and allowing each state to assume the linguistic context of the most promising history. The advantage of this approach is that it employs maximum sharing of data structures and information, so each node in the tree is evaluated at most once. However, the use of local knowledge to determine the dominant context could introduce significant errors because of premature pruning. Whisper [4] reports

a 65.7% increase in error rate when only the dominant context is kept, based on local knowledge.

ALGORITHM 13.2 HANDLING MULTIPLE LINGUISTIC CONTEXTS IN LEXICAL TREE

1. $d = Cost(s_n, c) + (-\log P(s_m \mid s_n, c))$

2. if $InHeap(s_m, c)$ **then**

(a) If $d < Cost(s_m, c)$ **then**

(i) $Cost(s_m, c) = d$

(ii) $StateInfo(s_m, c) = StateInfo(s_n, c)$

3. elseif $d < \varepsilon$ **then**

(a) $Add(s_m, c)$; $StateInfo(s_m, c) = StateInfo(s_n, c)$

(b) $Cost(s_m, c) = d$

4. else

(a) $w = WorstContext(s_m)$

(b) if $d < Cost(s_m, w)$ **then**

(i) $Delete(s_m, w)$

(ii) $Add(s_m, c)$; $StateInfo(s_m, c) = StateInfo(s_n, c)$

(iii) $Cost(s_m, c) = d$

To recover the errors created by using local linguistic information to estimate subtree dominance, you need to delay the decision regarding which linguistic context is most promising. This can be done by keeping a heap of contexts at each node in the tree. The heap maintains all contexts (linguistic paths) whose probabilities are within a constant threshold ε, comprised of the best global path plus the best path whose probability is less than ε but better than the global pruning threshold β. The effect of the ε-heap is that more contexts are retained for high-probability states in the lexical tree. The pseudocode fragment in Algorithm 13.2 [3] illustrates a transition from state s_n in context *c* to state s_m. The terminology used in Algorithm 13.2 is listed as follows:

- $(-\log P(s_m \mid s_n, c))$ is the cost associated with applying acoustic model matching and language model probability of state s_m given state s_n in context *c*.
- $InHeap(s_m, c)$ is true if context *c* is in the heap corresponding to state s_m.
- $Cost(s_m, c)$ is the score for context *c* in state s_m.
- $StateInfo(s_m, c)$ is the auxiliary state information associated with context *c* in state s_m.

- $Add(s_m, c)$ adds context *c* to the state s_m heap.
- $Delete(s_m, c)$ deletes context *c* from state s_m heap.
- $WorstContext(s_m)$ retrieves the worst context from the heap of state s_m.

When higher-order *n-gram* is used for lexical tree search, the potential heap size for lexical tree nodes (some also refer to *prefix nodes*) could be unmanageable. With decent acoustic models and efficient pruning, as illustrated in Algorithm 13.2, the average heap size for active nodes in the lexical tree is actually very modest. For example, Whisper's average heap size for active nodes in the 20,000-word WSJ lexical tree decoder is only about 1.6 [3].

13.1.6. Context-Dependent Units and Inter-Word Triphones

So far, we have implicitly assumed that context-independent models are used in the lexical tree search. When context-dependent phonetic or subphonetic models, as discussed in Chapter 9, are used for better acoustic models, the construction and use of a lexical tree becomes more complicated.

Since senones represent both subphonetic and context-dependent acoustic models, this presents additional difficulty for use in lexical trees. Let's assume that a three-state context-dependent HMM is formed from three senones, one for each state. Each senone is context-dependent and can be shared by different allophones. If we use allophones as the units for lexical tree, the sharing may be poor and fan-out unmanageable. Fortunately, each HMM is uniquely identified by the sequence of senones used to form the HMM. In this way, different context-dependent allophones that share the same *senone sequence* can be treated as the same. This is especially important for lexical tree search, since it reduces the order of the fan-out in the tree.

Interword triphones that require significant fan-ins for the first phone of a word and fan-outs for the last phones usually present an implementation challenge for large-vocabulary speech recognition. A common approach is to delay full interword modeling until a subsequent rescoring phase.[4] Given a sufficiently rich lattice or word graph, this is a reasonable approach, because the static state space in the successive search has been reduced significantly. However, as pointed out in Section 13.1.2, the size of the dynamic state space can remain under control when detailed models are used to allow effective pruning. In addition, a multipass search requires an augmented set of acoustic models to effectively model the biphone contexts used at word boundaries for the first pass. Therefore, it might be desirable to use genuine interword acoustic models in the single-pass search.

Instead of expanding all the fan-ins and fan-outs for inter-word context-dependent phone units in the lexical tree, three *metaunits* are created.

1. The first metaunit, which has a known right context corresponding to the second phone in the word, but uses open left context for the first phone of a word (sometimes referred to as the *word-initial unit*). In this way, the fan-in

[4] Multipass search strategy is described in Section 13.3.5.

is represented as a subgraph shared by all words with the same initial left-context-dependent phone.

2. Another metaunit, which has a known left context corresponding to the second-to-last phone of the word, but uses open right context for the last phone of a word (sometimes referred to as the *word-final unit*). Again, the fan-out is represented as a subgraph shared by all words with the same final right-context-dependent phone.
3. The third metaunit, which has both open left and right contexts, and is used for single-phone word unit.

By using these metaunits we can keep the states for the lexical trees under control, because the fan-in and fan-out are now represented as a single node.

During recognition, different left or right contexts within the same metaunit are handled using Algorithm 13.2, where the different acoustic contexts are treated similarly as different linguistic contexts. The open left-context metaunit (fan-ins) can be dealt with in a straightforward way using Algorithm 13.2, because the left context is always known (the last phone of the previous word) when it is initiated. On the other hand, the open right-context metaunit (fan-out) needs to explore all possible right contexts because the next word is not known yet. To reduce unnecessary computation, fast match algorithms (described in 13.2.3) can be used to provide both expected acoustic and language scores for different context-dependent units to result in early pruning of unpromising contexts.

13.2. OTHER EFFICIENT SEARCH TECHNIQUES

Tree structured lexicon represents an efficient framework of manipulation of search space. In this section we present some additional implementation techniques, which can be used to further improve the efficiency of search algorithms. Most of these techniques can be applied to both Viterbi beam search and stack decoding. They are essential ingredients for a practical large-vocabulary continuous speech recognizer.

13.2.1. Using Entire HMM as a State in Search

The state in state-search space based on HMM-trellis computation is by definition a Markov state. Phonetic HMM models are the basic unit in most speech recognizers. Even though subphonetic HMMs, like senones, might be used for such a system, the search is often based on phonetic HMMs.

Treating the entire phonetic HMM as a state in state-search has many advantages. The first obvious advantage is that the number of states the search program needs to deal with is smaller. Note that using the entire phonetic HMM does not in effect reduce the number of states in the search. The entire search space is unchanged. All the states within a phonetic HMM are now bundled together. This means that all of them are either kept in the beam, if the phonetic HMM is regarded as promising, or all of them are pruned away. For any given

time, the minimum cost among all the states within the phonetic HMM is used as the cost for the phonetic HMM. For pruning purposes, this cost score is used to determine the promising degree of this phonetic HMM, i.e., the fate of all the states within this phonetic HMM. Although this does not actually reduce the beam beyond normal pruning, it has the effect of processing fewer candidates in the beam. In programming, this means less checking and bookkeeping, so some computation savings can be expected.

You might wonder if this organization might be ineffective for beam search, since it forces you to keep or prune all the states within a phonetic HMM. In theory it is possible that only one or two states in the phonetic HMM need to be kept, while other states can be pruned due to high cost score. However, this is in reality very rare, since a phone is a small unit and all the states within a phonetic HMM should be relatively promising when the search is near the acoustic region corresponding to the phone.

During the trellis computation, all the phonetic HMM states need to advance one time step when processing one input vector. By performing HMM computation for all states together, the new organization can reduce memory accesses and improve cache locality, since the output and transition probabilities are held in common by all states. Combining this organization strategy with lexical tree search further enhances the efficiency. In lexical tree search, each hypothesis in the beam is associated with a particular node in the lexical tree. These hypotheses are linked together in the heap structure described in Algorithm 13.2 for the purposes of efficient evaluation and heuristic pruning. Since the node corresponds to a phonetic HMM, the HMM evaluation is guaranteed to execute once for each hypothesis sharing this node.

In summary, treating the entire phonetic HMM as a state in state-search space allows you to explore the effective data structure for better sharing and improved memory locality.

13.2.2. Different Layers of Beams

Because of the complexity of search, it often requires pruning of various levels of search to make search feasible. Most systems thus employ different pruning thresholds to control what states participate. The most frequently used thresholds are listed below:

- τ_s controls what states (either phone states or senone states) to retain. This is the most fundamental beam threshold.
- τ_p controls whether the next phone is extended. Although this might not be necessary for both stack decoding and linear Viterbi beam search, it is crucial for lexical tree search, because pruning unpromising phonetic prefixes in the lexical trees could improve search efficiency significantly.
- τ_h controls whether hypotheses are extended for the next word. Since the branching factor for word boundaries is very large, we need this threshold to limit search to only the promising ones.
- τ_c controls where a linguistic context is created in a lexical tree search using higher-order language models. This is also known as ε-heap in Algorithm 13.2.

Pruning can introduce search errors if a state is pruned that would have been on the globally best path. The principle applied here is that the more constraints you have available, the more aggressively you decide whether this path will participate in the globally best path. In this case, at the state level, you have the least constraints. At the phonetic level there are more, and there are most at the word level. In general the number of word hypotheses tends to drop significantly at word boundaries. Different thresholds for different levels allow the search designer to fine-tune those thresholds for their tasks to achieve best search performance without significant increase in error rates.

13.2.3. Fast Match

As described in Chapter 12, fast match is a crucial part of stack decoding, which mainly reducew the number of possible word expansions for each path. Similarly, fast match can be applied to the most expensive part—extending the phone HMM fan-outs within or between lexical trees. Fast match is a method for rapidly deriving a list of candidates that constrain successive search phases in which a computationally expensive *detailed match* is performed. In this sense, fast match can be regarded as an additional pruning threshold to meet before a new word/phone can be started.

Fast match is typically characterized by the approximations that are made in the acoustic/language models to reduce computation. The factorization of language model scores among tree branches in lexical trees described in Section 13.1.3 can be viewed as fast match using a language model. The factorized method is also an admissible estimate of the language model scores for the future word. In this section we focus on acoustic model fast match.

13.2.3.1. Look-Ahead Strategy

Fast match, when applied in time-synchronous search, is also called *look-ahead* strategy. since it basically searches ahead of the time-synchronous search by a few frames to determine which words or phones are likely to extend. Typically the look-ahead frames are fixed, and the fast match is also done in time-synchronous fashion with another specialized beam for efficient pruning. You can also use simplified models, like the one-state HMMs or context-independent models [4, 32]. Some systems [21, 22] have tried to simplify the level of details in the input feature vectors by aggregating information from several frames into one. A straightforward way for compressing the feature stream is to skip every other frame of speech for fast match. This allows a longer-range look-ahead, while keeping computation under control. The approach of simplifying the input feature stream instead of simplifying the acoustic models can reuse the fast match results for detailed match.

Whisper [4] uses phoneme look-ahead fast match in lexical tree search, in which pruning is applied based on the estimation of the score of possible phone fan-outs that may follow a given phone. A context-independent phone-net is searched synchronously with the search process but offset N frames into the future. In practice, significant savings can be obtained in search efforts without increase in error rates.

The performance of word and phoneme look-ahead clearly depends on the length of the look-ahead frames. In general, the larger the look-ahead window, the longer is the computation and the shorter the word/phone Λ list. Empirically, the window is a few tens of milliseconds for phone look-ahead and a few hundreds of milliseconds for word look-ahead.

13.2.3.2. The Rich-Get-Richer Strategy

For systems employing continuous-density HMMs, tens of mixtures of Gaussians are often used for the output probability distribution for each state. The computation of the mixtures is one of the bottlenecks when many context-dependent models are used. For example, Whisper uses about 120,000 Gaussians. In addition to using various beam pruning thresholds in the search, there could be significant savings if we have a strategy to limit the number of Gaussians to be computed.

The *Rich-Get-Richer* (RGR) strategy enables us to focus on most promising paths and treat them with detailed acoustic evaluations and relaxed path-pruning thresholds. On the contrary, the less promising paths are extended with less expensive acoustic evaluations and less forgiving path-pruning thresholds. In this way, locally optimal candidates continue to receive the maximum attention while less optimal candidates are retained but evaluated using less precise (computationally expensive) acoustic and/or linguistic models. The RGR strategy gives us finer control in the creation of new paths that has potential to grow exponentially.

RGR is used to control the level of acoustic details in the search. The goal is to reduce the number of context-dependent senone probability (Gaussian) computations required. The context-dependent senones associated with a phone instance p would be evaluated according to the following condition:

$$\begin{aligned} & Min\left[ci(p)\right] * \alpha + LookAhead\left[ci(p)\right] < \text{ threshold} \\ & \text{where } Min\left[ci(p)\right] = \min_{s}\left\{\cos t(s) \mid s \in ci\_phone(p)\right\} \\ & \text{and } \quad LookAhead\left[ci(p)\right] = \text{look-ahead estimate of } ci(p) \end{aligned} \tag{13.6}$$

These conditions state that the context-dependent senones associated with p should be evaluated if there exists a state s corresponding to p, whose cost score in linear combination with a look-ahead cost score corresponding to p falls within a threshold. In the event that p does not fall within the threshold, the senone scores corresponding to p are estimated using the context-independent senones corresponding to p. This means the context-dependent senones are evaluated only if the corresponding context-independent senones and the look-ahead start showing promise. RGR strategy should save significant senone computation for clearly unpromising paths. Whisper [26] reports that 80% of senone computation can be avoided without introducing significant errors for a 20,000-word WSJ dictation task.

13.3. *N*-BEST AND MULTIPASS SEARCH STRATEGIES

Ideally, a search algorithm should consider all possible hypotheses based on a unified probabilistic framework that integrates all *knowledge sources* (KSs).[5] These KSs, such as acoustic models, language models, and lexical pronunciation models, can be integrated in an HMM state search framework. It is desirable to use the most detailed models, such as context-dependent models, interword context-dependent models, and high-order *n*-grams, in the search as early as possible. When the explored search space becomes unmanageable, due to the increasing size of vocabulary or highly sophisticated KSs, search might be infeasible to implement.

As we develop more powerful techniques, the complexity of models tends to increase dramatically. For example, language understanding models in Chapter 17 require long-distance relationships. In addition, many of these techniques are not operating in a left-to-right manner. A possible alternative is to perform a multipass search and apply several KSs at different stages, in the proper order to constrain the search progressively. In the initial pass, the most discriminant and computationally affordable KSs are used to reduce the number of hypotheses. In subsequent passes, progressively reduced sets of hypotheses are examined, and more powerful and expensive KSs are then used until the optimal solution is found.

The early passes of multipass search can be considered fast match that eliminates those unlikely hypotheses. Multipass search is, in general, not admissible because the optimal word sequence could be wrongly pruned prematurely, due to the fact that not all KSs are used in the earlier passes. However, for complicated tasks, the benefits of computation complexity reduction usually outweigh the nonadmissibility. In practice, multipass search strategy using progressive KSs could generate better results than a search algorithm forced to use less powerful models due to computation and memory constraints.

The most straightforward multipass search strategy is the so-called *n*-best search paradigm. The idea is to use affordable KSs to first produce a list of *n* most probable word sequences in a reasonable time. Then these *n* hypotheses are rescored using more detailed models to obtain the most likely word sequence. The idea of the *n*-best list can be further extended to create a more compact hypotheses representation—namely word lattice or graph. A word lattice is a more efficient way to represent alternative hypotheses. *N*-best or lattice search is used for many large-vocabulary continuous speech recognition systems [20, 30, 44].

In this section we describe the representation of the *n*-best list and word lattice. Several algorithms to generate such an *n*-best-list or word lattice are discussed.

13.3.1. *N*-Best Lists and Word Lattices

Table 13.4 shows an example *n*-best (10-best) list generated for a North American Business (NAB) sentence. *N*-best search framework is effective only for *n* of the order of tens or hun-

[5] In the field of artificial intelligence, the process of performing search through an integrated network of various knowledge sources is called *constraint satisfaction.*

dreds. If the short *n*-best list that is generated by using less optimal models does not include the correct word sequence, the successive rescoring phases have no chance to generate the correct answer. Moreover, in a typical *n*-best list like the one shown in Table 13.4, many of the different word sequences are just one-word variations of each other. This is not surprising, since similar word sequences should achieve similar scores. In general, the number of *n*-best hypotheses might grow exponentially with the length of the utterance. Word lattices and word graphs are thus introduced to replace *n*-best list with a more compact representation of alternative hypotheses.

Table 13.4 An example 10-best list for a North American Business sentence.

1. I will tell you would I think in my office
2. I will tell you what I think in my office
3. I will tell you when I think in my office
4. I would sell you would I think in my office
5. I would sell you what I think in my office
6. I would sell you when I think in my office
7. I will tell you would I think in my office
8. I will tell you why I think in my office
9. I will tell you what I think on my office
10. I Wilson you I think on my office

Word lattices are composed by word hypotheses. Each word hypothesis is associated with a score and an explicit time interval. Figure 13.8 shows an example of a word lattice corresponding to the *n*-best list example in Table 13.4. It is clear that a word lattice is more efficient representation. For example, suppose the spoken utterance contains 10 words and there are 2 different word hypotheses for each word position. The *n*-best list would need to have $2^{10} = 1024$ different sentences to include all the possible permutations, whereas the word lattice requires only 20 different word hypotheses.

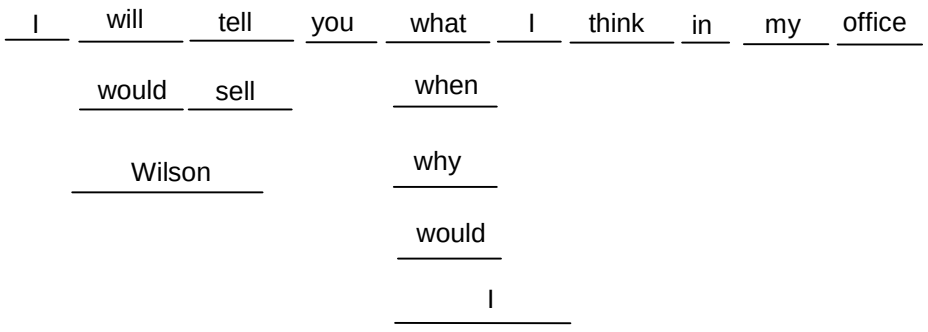

Figure 13.8 A word lattice example. Each word has an explicit time interval associated with it.

Word graphs, on the other hand, resemble finite state automata, in which arcs are labeled with words. Temporal constraints between words are implicitly embedded in the topology. Figure 13.9 shows a word graph corresponding to the *n*-best list example in Figure 13.12. Word graphs in general have an explicit specification of word connections that don't allow overlaps or gaps along the time axis. Nonetheless, word lattices and graphs are similar, and we often use these terms interchangeably.[6] Since an *n*-best list can be treated as a simple word lattice, word lattices are a more general representation of alternative hypotheses. *N*-best lists or word lattices are generally evaluated on the following two parameters:

- *Density:* In the *n*-best case, it is measured by how many alternative word sequences are kept in the *n*-best list. In the word lattice case, it is measured by the number of word hypotheses or word arcs per uttered word. Obviously, we want the density to be as small as possible for successive rescoring modules, provided the correct word sequence is included in the *n*-best list or word lattice.
- *The lower bound word error rate:* It is the lowest word error rate for any word sequence in the *n*-best list or the word lattice.

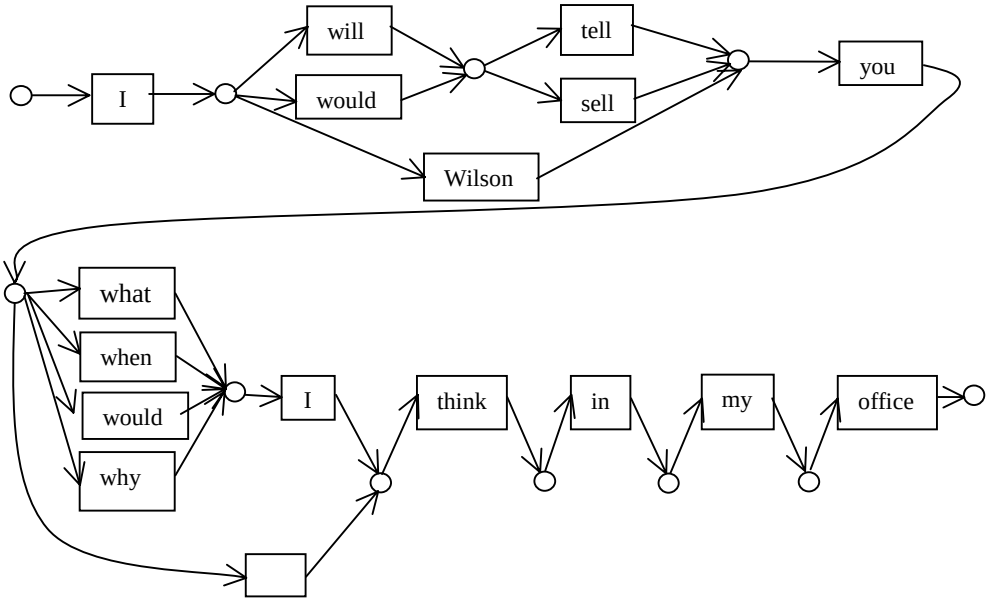

Figure 13.9 A word graph example for the *n*-best list in Table 13.4. Temporal constraints are implicit in the topology.

Rescoring with highly similar *n*-best alternatives duplicates computation on common parts. The compact representation of word lattices allows both data structure and computation sharing of the common parts among similar alternative hypotheses, so it is generally computationally less expensive to rescore the word lattice.

[6] We use the term *word lattice* exclusively for both representations.

Figure 13.10 illustrates the general *n*-best/lattice search framework. Those KSs providing most constraints, at a lesser cost, are used first to generate the *n*-best list or word lattice. The *n*-best list or word lattice is then passed to the rescoring module, which uses the remaining KSs to select the optimal path. You should note that the *n*-best and word-lattice generator sometimes involve several phases of search mechanisms to generate the *n*-best list or word lattice. Therefore, the whole search framework in Figure 13.10 could involve several (> 2) phases of search mechanism.

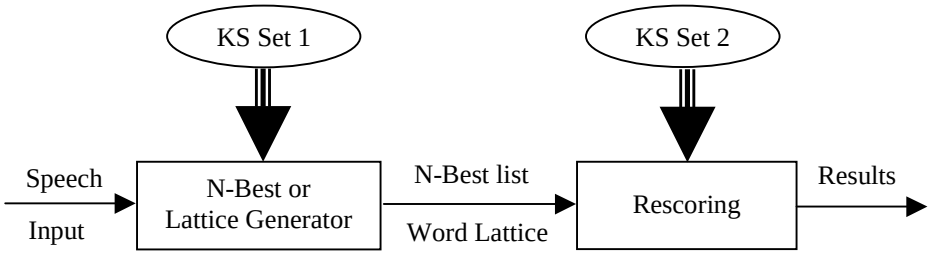

Figure 13.10 *N*-best/lattice search framework. The most discriminant and inexpensive knowledge sources (KSs 1) are used first to generate the *n*-best/lattice. The remaining knowledge sources (KSs 2, usually expensive to apply) are used in the rescoring phase to pick up the optimal solution [40].

Does the compact *n*-best or word-lattice representation impose constraints on the complexity of the acoustic and language models applied during successive rescoring modules? The word lattice can be expanded for higher-order language models and detailed context-dependent models, like inter-word triphone models. For example, to use higher-order language models for word lattice entails copying each word in the appropriate context of preceding words (in the trigram case, the two immediately preceding words). To use interword triphone models entails replacing the triphones for the beginning and ending phone of each word with appropriate interword triphones. The expanded lattice can then be used with detailed acoustic and language models. For example, Murveit et al. [30] report this can achieve trigram search without exploring the enormous trigram search space.

13.3.2. The Exact *N*-best Algorithm

Stack decoding is the choice of generating *n*-best candidates because of its best-first principle. We can keep it generating results until it finds *n* complete paths; these *n* complete sentences form the *n*-best list. However, this algorithm usually cannot generate the *n* best candidates efficiently. The efficient *n*-best algorithm for time-synchronous Viterbi search was first introduced by Schwartz and Chow [39]. It is a simple extension of time-synchronous Viterbi search. The fundamental idea is to maintain separate records for paths with distinct histories. The history is defined as the whole word sequence up to the current time *t* and word *w*. This exact *n*-best algorithm is also called *sentence-dependent n*-best algorithm. When two or more paths come to the same state at the same time, paths having the same history are merged and their probabilities are summed together; otherwise only the *n* best

paths are retained for each state. As commonly used in speech recognition, a typical HMM state has 2 or 3 predecessor states within the word HMM. Thus, for each time frame and each state, the *n*-best search algorithm needs to compare and merge 2 or 3 sets of *n* paths into *n* new paths. At the end of the search, the *n* paths in the final state of the trellis are simply re-ordered to obtain the *n* best word sequences.

This straightforward *n*-best algorithm can be proved to be admissible[7] in normal circumstances [40]. The complexity of the algorithm is proportional to $O(n)$, where *n* is the number of paths kept at each state. This is often too slow for practical systems.

13.3.3. Word-Dependent *N*-Best and Word-Lattice Algorithm

Since many of the different entries in the *n*-best list are just one-word variations of each other, as shown in Table 13.4, one efficient algorithm can be derived from the normal 1-best Viterbi algorithm to generate the *n* best hypotheses. The algorithm runs just like the normal time-synchronous Viterbi algorithm for all within-word transitions. However for each time frame *t*, and each word-ending state, the algorithm stores all the different words that can end at current time *t* and their corresponding scores in a *traceback* list. At the same time, the score of the best hypothesis at each grammar state is passed forward, as in the normal time-synchronous Viterbi search. This obviously requires almost no extra computation above the normal time-synchronous Viterbi search. At the end of search, you can simply search through the stored traceback list to get all the permutations of word sequences with their corresponding scores. If you use a simple threshold, the traceback can be implemented very efficiently to only uncover the word sequences with accumulated cost scores below the threshold. This algorithm is often referred as *traceback*-based *n*-best algorithm [29, 42] because of the use of the traceback list in the algorithm.

However, there is a serious problem associated with this algorithm. It could easily miss some low-cost hypotheses. Figure 13.11 illustrates an example in which word w_k can be preceded by two different words w_i and w_j in different time frames. Assuming path w_i-w_k has a lower cost than path w_j-w_k when both paths meet during the trellis search of w_k, the path w_j-w_k will be pruned away. During traceback for finding the *n* best word sequences, there is only one best starting time for word w_k, determined by the best boundary between the best preceding word w_i and it. Even though path w_j-w_k might have a very low cost (let's say only marginally higher than that of w_i-w_k), it could be completely overlooked, since the path has a different starting time for word w_k.

[7] Although one can show in the worst case, when paths with different histories have near identical scores for each state, the search actually needs to keep all paths ($> N$) in order to guarantee absolute admissibility. Under this worst case, the admissible algorithm is clearly exponential in the number of words for the utterance, since all permutations of word sequences for the whole sentence need to be kept.

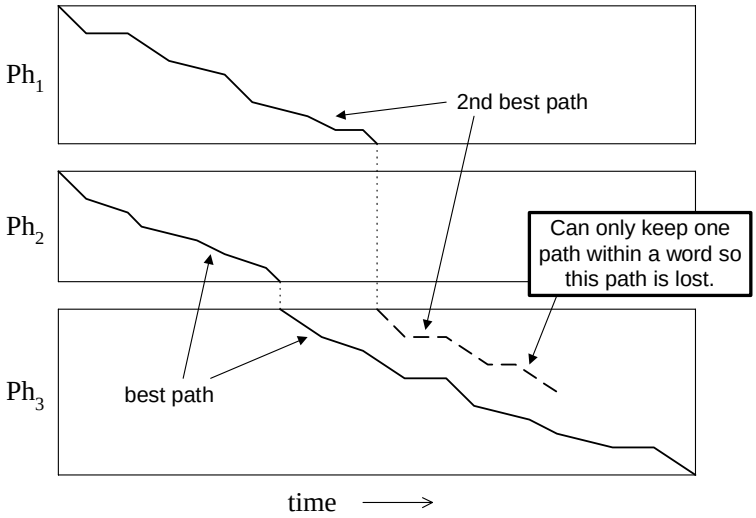

Figure 13.11 Deficiency in traceback-based *n*-best algorithm. The best subpath, w_j - w_k , will prune away subpath w_i - w_k while searching the word w_k ; the second-best subpath cannot be recovered [40].

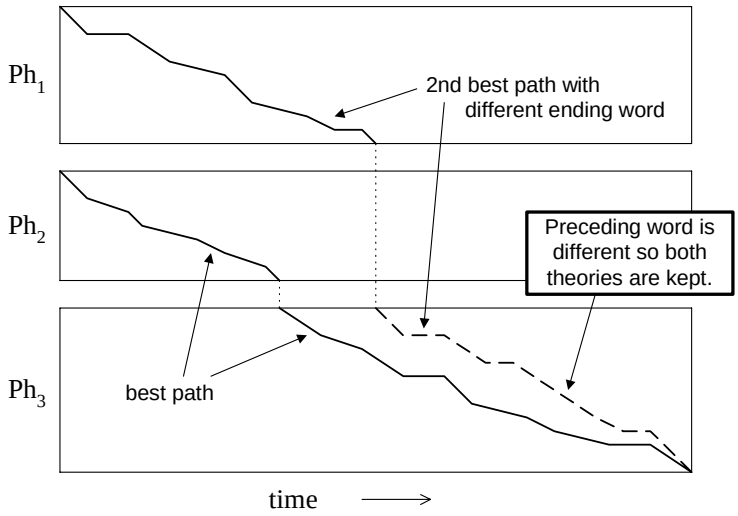

Figure 13.12 Word-dependent *n*-best algorithm. Both subpaths w_j - w_k and w_i - w_k are kept under the word-dependent assumption [40].

The *word-dependent n*-best algorithm [38] can alleviate the deficiency of the trace-back-based *n*-best algorithm, in which only one starting time is kept for each word, so the starting time is independent of the preceding words. On the other hand, in the sentence-dependent *n*-best algorithm, the starting time for a word clearly depends on all the preceding words, since different histories are kept separately. A good compromise is the so-called word-dependent assumption: *The starting time of a word depends only on the immediate preceding word. That is, given a word pair and its ending time, the boundary between these two words is independent of further predecessor words.*

In the word-dependent assumption, the history to be considered for a different path is no longer the entire word sequence; instead, it is only the immediately preceding word. This allows you to keep k ($<< n$) different records for each state and each time frame in Viterbi search. Differing slightly from the exact *n*-best algorithm, a traceback must be performed to find the *n*-best list at the end of search. The algorithm is illustrated in Figure 13.12. A word-dependent *n*-best algorithm has a time complexity proportional to k. However, it is no longer admissible because of the word-dependent approximation. In general, this approximation is quite reasonable if the preceding word is long. The loss it entails is insignificant [6].

13.3.3.1. One-Pass *N*-Best and Word-Lattice Algorithm

As presented in Section 13.1, one-pass Viterbi beam search can be implemented very efficiently using a tree lexicon. Section 13.1.2 states that multiple copies of lexical trees are necessary for incorporating language models other than the unigram. When bigram is used in lexical tree search, the successor lexical tree is predecessor-dependent. This predecessor-dependent property immediately translates into the word-dependent property,[8] as defined in Section 13.3.3, because the starting time of a word clearly depends on the immediately preceding word. This means that different word-dependent partial paths are automatically saved under the framework of predecessor-dependent successor trees. Therefore, one-pass predecessor-dependent lexical tree search can be modified slightly to output *n*-best lists or word graphs.

Ney el al. [31] used a word graph builder with a one-pass predecessor-dependent lexical tree search. The idea is to exploit the word-dependent property inherited from the predecessor-dependent lexical tree search. During predecessor-dependent lexical tree search, two additional quantities are saved whenever a word ending state is processed.

$\tau(t; w_i, w_j)$ —Representing the optimal word boundary between word w_i and w_j, given word w_j ending at time *t*.

$h(w_j; \tau(t; w_i, w_j), t) = -\log P(\mathbf{x}_\tau^t \mid w_j)$ —Representing the cumulative score that word w_j produces acoustic vector $\mathbf{x}_\tau, \mathbf{x}_{\tau+1}, \cdots \mathbf{x}_t$.

[8] When higher order n-gram models are used, the boundary dependence will be even more accurate. For example, when trigrams are used, the boundary for a word juncture depends on the previous two words. Since we generally want a fast method of generating word lattices/graphs, bigram is often used instead of higher order n-gram to generate word lattices/graphs.

At the end of the utterance, the word graph or *n*-best list is constructed by tracing back all the permutations of word pairs recorded during the search. The algorithm is summarized in Algorithm 13.3.

ALGORITHM 13.3 ONE-PASS PREDECESSOR-DEPENDENT LEXICAL TREE SEARCH FOR *N*-BEST OR WORD-LATTICE CONSTRUCTION

For $t = 1..T$,
    1-best predecessor-dependent lexical tree search;
    $\forall(w_i, w_j)$ ending at t
        record word-dependent crossing time $\tau(t; w_i, w_j)$;
        record cumulative word score $h(w_j; \tau(t; w_i, w_j), t)$;
Output 1-best result;
Construct *n*-best or word-lattice by tracing back the word-pair records (τ and h).

13.3.4. The Forward-Backward Search Algorithm

As described Chapter 12, the ability to predict how well the search fares in the future for the remaining portion of the speech helps to reduce the search effort significantly. The one-pass search strategy in general has very little chance of predicting the cost for the portion that it has not seen. This difficulty can be alleviated by multipass search strategies. In successive phases the search should be able to provide good estimates for the remaining paths, since the entire utterance has been examined by the earlier passes. In this section we investigate a special type of multipass search strategy—forward-backward search.

The idea is to first perform a forward search, during which partial forward scores α for each state can be stored. Then perform a second pass search backward—that is, the second pass starts by taking the final frame of speech and searches its way back until it reaches the start of the speech. During the backward search, the partial forward scores α can be used as an accurate estimate of the heuristic function or the fast match score for the remaining path. Even though different KSs might be used in forward and backward phases, this estimate is usually close to perfect, so the search effort for the backward phase can be significantly reduced.

The forward search must be very fast and is generally a time-synchronous Viterbi search. As in the multipass search strategy, simplified acoustic and language models are often used in forward search. For backward search, either time-synchronous search or time-asynchronous A* search can be employed to find the *n* best word sequences.

13.3.4.1. Forward-Backward Search

Stack decoding, as described in Chapter 12, is based on the admissible A* search, so the first complete hypothesis found with a cost below that of all the hypotheses in the stack is guar-

anteed to be the best word sequence. It is straightforward to extend stack decoding to produce the *n* best hypotheses by continuing to extend the partial hypotheses according to the same A* criterion until *n* different hypotheses are found. These *n* different hypotheses are destined to be the *n* best hypotheses under a proof similar to that presented in Chapter 12. Therefore, stack decoding is a natural choice for producing the *n* best hypotheses.

However, as described in Chapter 12, the difficulty of finding a good heuristic function that can accurately under-estimate the remaining path has limited the use of stack decoding. Fortunately, this difficulty can be alleviated by *tree-trellis forward-backward search* algorithms [41]. First, the search performs a time-synchronous forward search. At each time frame *t*, it records the score of the final state of each word ending. The set of words whose final states are active (surviving in the beam) at time *t* is denoted as Δ_t. The score of the final state of each word w in Δ_t is denoted as $\alpha_t(w)$, which represents the sum of the cost of matching the utterance up to time *t* given the most likely word sequence ending with word w and the cost of the language model score for that word sequence. At the end of the forward search, the best cost is obtained and denoted as α^T.

After the forward pass is completed, the second search is run in reverse (backward), i.e., considering the last frame *T* as the beginning one and the first frame as the final one. Both the acoustic models and language models need to be reversed. The backward search is based on A* search. At each time frame *t*, the best path is removed from the stack and a list of possible one-word extensions for that path is generated. Suppose this best path at time *t* is ph_{w_j}, where w_j is the first word of this partial path (the last expanded during backward A* search). The exit score of path ph_{w_j} at time *t*, which now corresponds to the score of the initial state of the word HMM w_j, is denoted as $\beta_t(ph_{w_j})$.

Let us now assume we are concerned about the one-word extension of word w_i for path ph_{w_j}. Remember that there are two fundamental issues for the implementation of A* search algorithm—(1) finding an effective and efficient heuristic function for estimating the future remaining input feature stream and (2) finding the best crossing time between w_i and w_j.

The stored forward score α can be used for solving both issues effectively and efficiently. For each time *t*, the sum $\alpha_t(w_i)+\beta_t(ph_{w_j})$ represents the cost score of the best complete path including word w_i and partial path ph_{w_j}. $\alpha_t(w_i)$ clearly represents a very good heuristic estimate of the remaining path from the start of the utterance until the end of the word w_i, because it is indeed the best score computed in the forward path for the same quantity. Moreover, the optimal crossing time t^* between w_i and w_j can be easily computed by the following equation:

$$t^* = \arg\min_t \left[\alpha_t(w_i) + \beta_t(ph_{w_j}) \right] \tag{13.7}$$

Finally, the new path ph' including the one-word (w_i) extension is inserted into the stack, ordered by the cost score $\alpha_{t^*}(w_i) + \beta_{t^*}(ph_{w_j})$. The heuristic function (forward scores α) allows the backward A* search to concentrate search on extending only few truly promising paths.

As a matter of fact, if the same acoustic and language models are used in both the forward and backward search, this heuristic estimate (forward scores α) is indeed a perfect estimate of the best score the extended path will achieve. The first complete hypothesis generated by backward A* search coincides with the best one found in the time-synchronous forward search and is truly the best hypothesis. Subsequent complete hypotheses correspond sequentially to the *n*-best list, as they are generated in increasing order of cost. Under this condition, the size of the stack in the backward A* search need only be *N*. Since the estimate of future is exact, the $(N+1)^{\text{th}}$ path in the stack has no chance to become part of the *n*-best list. Therefore, the backward search is executed very efficiently to obtain the *n* best hypotheses without exploring many unpromising branches. Of course, tree-trellis forward-backward search can also be used like most other multipass search strategies—inexpensive KSs are used in the forward search to get an estimate of α, and more expensive KSs are used in the backward A* search to generate the *n*-best list.

The same idea of using forward score α can be applied to time-synchronous Viterbi search in the backward search instead of backward A* search [7, 34]. For large-vocabulary tasks, the backward search can run 2 to 3 orders of magnitude faster than a normal Viterbi beam search. To obtain the *n*-best list from time-synchronous forward-backward search, the backward search can also be implemented in a similar way as a time-synchronous word-dependent *n*-best search.

13.3.4.2. Word-Lattice Generation

The forward-backward *n*-best search algorithm can be easily modified to generate word lattices instead of *n*-best lists. A forward time-synchronous Viterbi search is performed first to compute $\alpha_t(\omega)$, the score of each word ω ending at time *t*. At the end of the search, this best score α^T is also recorded to establish the global pruning threshold. Then, a backward time-synchronous Viterbi search is performed to compute $\beta_t(\omega)$, the score of each word ω beginning at time *t*. To decide whether to include word juncture $\omega_i - \omega_j$ in the word lattice/graph at time *t*, we can check whether the forward-backward score is below a global pruning threshold. Specifically, supposed bigram probability $P(\omega_j \mid \omega_i)$ is used, if

$$\alpha_t(\omega) + \beta_t(\omega) + \left[-\log P(\omega_j \mid \omega_i)\right] < \alpha^T + \theta \tag{13.8}$$

where θ is the pruning threshold, we will include $\omega_i - \omega_j$ in the word lattice/graph at time *t*. Once word juncture $\omega_i - \omega_j$ is kept, the search continues looking for the next word-pair, where the first word ω_i will be the second word of the next word-pair.

The above formulation is based on the assumption of using the same acoustic and language models in both forward and backward search. If different KSs are used in forward and backward search, the normalized α and β scores should be used instead.

13.3.5. One-Pass vs. Multipass Search

There are several real-time one-pass search engines [4, 5]. Is it necessary to build a multipass search engine based on *n*-best or word-lattice rescoring? We address this issue by discussing the disadvantages and advantages of multipass search strategies.

One criticism of multipass search strategies is that they are not suitable for real-time applications. No matter how fast the first pass is, the successive (backward) passes cannot start until users finish speaking. Thus, the search results need to be delayed for at least the time required to execute the successive (backward) passes. This is why the successive passes must be extremely fast in order to shorten the delay. Fortunately, it is possible to keep the delays minimum (under one second) with clever implementation of multipass search algorithms, as demonstrated by Nguyen et al. [18].

Another criticism for multipass search strategies is that each pass has the potential to introduce inadmissible pruning, because decisions made in earlier passes are based on simplified models (KSs). Search is a constraint-satisfaction problem. When a pruning decision in each search pass is made on a subset of constraints (KSs), pruning error is inevitable and is unrecoverable by successive passes. However, inadmissible pruning, like beam pruning and fast match, is often necessary to implement one-pass search in order to cope with the large active search space caused jointly by complex KSs and large-vocabulary tasks. Thus, the problem of inadmissibility is actually shared by both real-time one-pass search and multipass search for different reasons. Fortunately, in both cases, search errors can be reduced to minimum by clever implementation and by empirically designing all the pruning thresholds carefully, as demonstrated in various one-pass and multipass systems [4, 5, 18].

Despite these concerns regarding multipass search strategies, they remain important components in developing spoken language systems. We list here several important aspects:

1. It might be necessary to use multipass search strategies to incorporate very expensive KSs. Higher-order *n*-gram, long-distance context-dependent models, and natural language parsing are examples that make the search space unmanageable for one-pass search. Multipass search strategies might be compelling even for some small-vocabulary tasks. For example, there are only a couple of million legal credit card numbers for the authentication task of 16-digit credit card numbers. However, it is very expensive to incorporate all the legal numbers explicitly in the recognition grammar. To first reduce search space down to an *n*-best list or word lattice/graph might be a desirable approach.
2. Multipass search strategies could be very compelling for spoken language understanding systems. It is problematic to incorporate most natural language understanding technologies in one-pass search. On the other hand, *n*-best lists or word lattices provide a trivial interface between speech recognition and

natural language understanding modules. Such an interface also provides a convenient mechanism for integrating different KSs in a modular way. This is important because the KSs could come from different modalities (like video or pen) that make one-pass integration almost infeasible. This high degree of modality allows different component subsystems to be optimized and implemented independently.

3. *N*-best lists or word lattices are very powerful offline tools for developing new algorithms for spoken language systems. It is often a significant task to fully integrate new modeling techniques, such as segment models, into a one-pass search. The complexity could sometimes slow down the progress of the development of such techniques, since recognition experiments are difficult to conduct. Rescoring of *n*-best list and lattice provides a quick and convenient alternative for running recognition experiments. Moreover, the computation and storage complexity can be kept relatively constant for offline *n*-best or word lattice/graph search strategies even when experimenting with highly expensive new modeling techniques. New modeling techniques can be experimented with using *n*-best/word-graph framework first, being integrated into the system only after significant improvement is demonstrated.
4. Besides being an alternative search strategy, *n*-best generation is also essential for discriminant training. Discriminant training techniques, like MMIE, and MCE described in Chapter 4, often need to compute statistics of all possible rival hypotheses. For isolated word recognition using word models, it is easy to enumerate all the word models as the rival hypotheses. However, for continuous speech recognition, one needs to use an all-phone or all-word model to generate all possible phone sequences or all possible word sequences during training. Obviously, that is too expensive. Instead, one can use *n*-best search to find all the near-miss sentence hypotheses that we want to discriminate against [15, 36].

13.4. SEARCH-ALGORITHM EVALUATION

Throughout this chapter we are careful in following dynamic programming principles, using admissible criteria as much as possible. However, many heuristics are still unavoidable to implement large-vocabulary continuous speech recognition in practice. Those nonadmissible heuristics include:

- Viterbi score instead of forward score described in Chapter 12.
- Beam pruning or stack pruning described in Section 13.2.2 and Chapter 12.
- Subtree dominance pruning described in Section 13.1.5.
- Fast match pruning described in Section 13.2.3.
- Rich-get-richer pruning described in Section 13.2.3.2.
- Multipass search strategies described in Section 13.3.5.

Nonadmissible heuristics generate suboptimal searches where the found path is not necessarily the path with the minimum cost. The question is, how different is this suboptimal from the true optimal path? Unfortunately, there is no way to know the optimal path unless an exhaustive search is conducted. The practical question is whether the suboptimal search hurts the search result. In a test condition where the true result is specified, you can easily compare the search result with the true result to find whether any error occurs. Errors could be due to inaccurate models (including acoustic and language models), suboptimal search, or end-point detection. The error caused by a suboptimal search algorithm is referred to as *search error* or *pruning error*.

How can we find out whether the search commits a pruning error? One of the procedures most often used is straightforward. Let $\hat{\mathbf{W}}$ be the recognized word sequence from the recognizer and $\tilde{\mathbf{W}}$ be the true word sequence. We need to compare the cost for these two word sequences:

$$- \log P(\hat{\mathbf{W}} \mid \mathbf{X}) \propto -\log\left[P(\hat{\mathbf{W}}) P(\mathbf{X} \mid \hat{\mathbf{W}}) \right] \tag{13.9}$$

$$- \log P(\tilde{\mathbf{W}} \mid \mathbf{X}) \propto -\log\left[P(\tilde{\mathbf{W}}) P(\mathbf{X} \mid \tilde{\mathbf{W}}) \right] \tag{13.10}$$

The quantity in Eq. (13.9) is supposed to be minimum among all possible word sequences if the search is admissible. Thus, if the quantity in Eq. (13.10) is greater than that in Eq. (13.9), the error is not attributed to search pruning. On the other hand, if the quantity in Eq. (13.10) is smaller than that in Eq. (13.9), there is a search error. The rationale behind the procedure described here is obvious. In the case of search errors, the suboptimal search (or nonadmissible pruning) has obviously pruned the correct path, because the cost of the correct path is smaller than the one found by the recognizer. Although we can conclude that search errors are found in this case, it does not guarantee that the search result is correct if the search can be made optimal. The reason is simply that there might be one pruned path with an incorrect word sequence and lower cost under the same suboptimal search. Therefore, the search errors represent only the upper bound that one can improve on if an optimal search is carried out. Nonetheless, finding search errors by comparing quantities in Eqs. (13.9) and (13.10) is a good measure in different search algorithms.

During the development of a speech recognizer, it is a good idea to always include the correct path in the search space. By including such a path, and some bookkeeping, one can use the correct path to help in determining all the pruning thresholds. If the correct path is pruned away during search by some threshold, some adjustment can be made to relax such a threshold to retain the correct path. For example, one can adjust the pruning threshold for fast match if a word in $\tilde{\mathbf{W}}$ fails to appear on the list supplied by the fast match.

13.5. CASE STUDY—MICROSOFT WHISPER

We use the decoder of Microsoft's Whisper [26, 27] discussed in Chapter 9 as a case study for reviewing the search techniques we have presented in this chapter. Whisper can handle

both context-free grammars for small-vocabulary tasks and *n*-gram language models for large-vocabulary tasks. We describe these two different cases.

13.5.1. The CFG Search Architecture

Although context-free grammars (CFGs) have the disadvantage of being too restrictive and unforgiving, particularly with novice users, they are still one of the most popular configurations for building limited-domain applications because of following advantages:

- Compact representation results in a small memory footprint.
- Efficient operation during decoding in terms of both space and time.
- Ease of grammar creation and modification for new tasks.

As mentioned in Chapter 12, the CFG grammar consists of a set of productions or rules that expand nonterminals into a sequence of terminals and nonterminals. Nonterminals in the grammar tend to refer to high-level task-specific concepts such as dates, font names, and commands. The terminals are words in the vocabulary. A grammar also has a nonterminal designated as its start state. Whisper also allows some regular expression operators on the right-hand side of the production for notational convenience. These operators are: or '|'; repeat zero or more times '*'; repeat one or more times '+'; and optional ([ ]). The following is a simple CFG example for *binary number:*

```
%start BINARY_NUMBER
BINARY_NUMBER: (zero | one)*
```

Without losing generality, Whisper disallows the left recursion for ease of implementation [2]. The grammar is compiled into a binary linked list format. The binary format currently has a direct one-to-one correspondence with the text grammar components, but is more compact. The compiled format is used by the search engine during decoding. The binary representation consists of variable-sized nodes linked together. The grammar format achieves sharing of subgrammars through the use of shared nonterminal definition rules.

The CFG search is conducted according to RTN framework (see Chapter 12). During decoding, the search engine pursues several paths through the CFG at the same time. Associated with each of the paths is a grammar state that describes completely how the path can be extended further. When the decoder hypothesizes the end of the current word of a path, it asks the grammar module to extend the path further by one word. There may be several alternative successor words for the given path. The decoder considers all the successor word possibilities. This may cause the path to be extended to generate several more paths to be considered, each with its own grammar state. Also note that the same word might be under consideration by the decoder in the context of different paths and grammar states at the same time.

The decoder uses beam search to prune unpromising paths with three different beam thresholds. The state pruning threshold τ_s and new phone pruning threshold τ_p work as described in Section 13.2.2. When extending a path, if the score of the extended path does

not exceed the threshold τ_h, the path to be extended is put into a pool. At each frame, for each word in the vocabulary, a winning path that extends to that word is picked from the pool, based on the score. All the remaining paths in the pool are pruned. This level of pruning gives us finer control in the creation of new paths that have potential to grow exponentially.

When two paths representing different word sequences thus far reach the end of the current word with the same grammar state at the same time, only the better path of the two is allowed to continue on. This optimization is safe, except that it does not take into account the effect of different interword left acoustic contexts on the scores of the new word that is started.

Besides beam pruning, the RGR strategy, described in Section 13.2.3.2, is used to avoid unnecessary senone computation. The basic idea is to use the linear combination of context-independent senone score and context-independent look-ahead score to determine whether the context-dependent senone evaluation is worthwhile to pursue.

All of these pruning techniques enable Whisper to perform typical 100- to 200-word CFG tasks in real time running on a 486 PC with 2 MB RAM. Readers might think it is not critical to make CFG search efficient on such a low-end platform.[9] However, it is indeed important to keep the CFG engine fast and lean. The speech recognition engine is eventually only part of an integrated application. The application will benefit if the resources (both CPU and memory) used by the speech decoder are kept as small as possible, so there are more resources left for the application module to use. Moreover, in recognition server applications, several channels of speech recognition can be performed on a single server platform if each speech recognition engine takes only a small portion of the total resources.

13.5.2. The *N*-Gram Search Architecture

The CFG decoder is best suited for limited domain command and control applications. For dictation or natural conversational systems, a stochastic grammar such as *n*-grams provides a more natural choice. Using bigrams or trigrams leads to a large number of states to be considered by the search process, requiring an alternative search architecture.

Whisper's *n*-gram search architecture is based on lexical tree search as described in Section 13.1. To keep the runtime memory[10] as small as possible, Whisper does not to allocate the entire lexical tree network statically. Instead it dynamically builds only the portion that needs to be active. To cope with the problem of delayed application of language model scores, Whisper uses the factorization algorithm described in Section 13.1.3 to distribute the language model probabilities through the tree branches. To reduce the memory overhead of the factored language model probabilities, an efficient data structure is used for representing the lexical tree as described in Section 13.1.3.1. This data structure allows Whisper to encode factored language model probabilities in no more than the space required for the origi-

[9] Thanks to the progress predicted by Moore's law, the current mainstream PC configuration is an order of magnitude more powerful than the configuration we list here (486 PC with 2 MB RAM) in both speed and memory.

[10] Here the runtime memory means the virtual memory for the decoder that is the entire image of the decoder.

nal *n*-gram probabilities. Thus, there is absolutely no storage overhead for using factored lexical trees.

Table 13.5 Configuration of the first seven levels of the 20,000-word WSJ (*Wall Street Journal*) tree;.the large initial fan-out is due to the use of context-dependent acoustic models [4].

Tree Level	Number of Nodes	Fan-Out
1	655	655.0
2	3174	4.85
3	9388	2.96
4	13,703	1.46
5	14,918	1.09
6	13,907	0.93
7	11,389	0.82

The basic acoustic subword model in Whisper is a context-dependent senone. It also incorporates inter-word triphone models in the lexical tree search as described in Section 13.1.6. Table 13.5 shows the distribution of phoneme arcs for 20,000-word WSJ lexical tree using senones as acoustic models. Context-dependent units certainly prohibit more prefix sharing when compared with Table 13.1. However, the overall arcs in the lexical tree still represent quite a saving when compared with a linear lexicon with about 140,000 phoneme arcs. Most importantly, similar to the case in Table 13.1, most sharing is realized in the beginning prefixes where most computation resides. Moreover, with the help of context-dependent and interword senone models, the search is able to use more reliable knowledge to perform efficient pruning. Therefore, lexical tree with context-dependent models can still enjoy all the benefits associated with lexical tree search.

The search organization is evaluated on the 1992 development test set for the *Wall Street Journal* corpus with a back-off trigram language model. The trigram language model has on the order of 10^7 linguistic equivalent classes, but the number of classes generated is far fewer due to the constraints provided by the acoustic model. Figure 13.13(a) illustrates that the relative effort devoted to the trigram, bigram, and unigram is constant regardless of total search effort, across a set of test utterances. This is because the ratio of states in the language model is constant. The language model is using $\sim 2 \times 10^6$ trigrams, $\sim 2 \times 10^6$ bigrams, and 6×10^4 unigrams. Figure 13.13(b) illustrates different relative order when word hypotheses are considered. The most common context for word hypotheses is the unigram context, followed by the bigram and trigram contexts. The reason for the reversal from the state-level transitions is the partially overlapping evaluations required by each bigram context. The trigram context is more common than the bigram context for utterances that generate few hypotheses overall. This is likely because the language model models those utterances well.

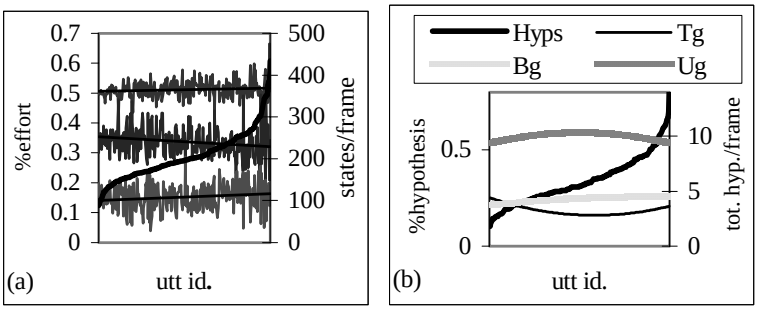

Figure 13.13 (a) Search effort for different linguistic contexts measured by number of active states in each of the three different linguistic contexts. The top series is for the bigram, then the unigram and trigram. The remaining series is the effort per utterance and is plotted on the secondary *y*-axis. **(b)** The distribution of word hypothesis with respect to their context. The top line is the unigram context, then the bigram and trigram. The remaining series is the average number of hypotheses per frame for each utterance and is plotted on the secondary *y*-axis [3].

Table 13.6 Effect of heap threshold on contexts/node, states/frame-of-speech (fos), word error rate, and search time [4].

ε	Context / node	states / fos	%error	search time
0	1.000	8805	16.4	1.0x
1.0	1.001	8808	15.5	1.0x
2.0	1.008	8898	14.4	1.0x
3.0	1.018	9252	12.4	1.07x
4.0	1.056	10224	10.5	1.16x
5.0	1.147	11832	10.3	1.36x
6.0	1.315	13749	10.0	1.60x
7.0	1.528	15342	9.9	1.81x
8.0	1.647	15984	9.9	1.86x

To improve efficiency in dealing with tree copies due to the use of higher-order *n*-gram, one needs to reduce redundant computations in subtrees that are not explicitly part of the given linguistic context. One solution is to use successor trees to include only nonzero successors, as described in Section 13.1.2. Since Whisper builds the search space dynamically, it is not effective for Whisper to use the optimization techniques of the successor-tree network, such as FSN optimization, subtree isomorphism, and sharing tail optimization. Instead, Whisper uses polymorphic linguistic context assignment to reduce redundancy, as described in Section 13.1.5. This involves keeping a single copy of the lexical tree, so that each node in the tree is evaluated at most once. To avoid early inadmissible pruning of different linguistic contexts, an ε-heap of storing paths of different linguistic contexts is cre-

ated for each node in the tree. The operation of such ε-heaps is in accordance with Algorithm 13.2. The depth of each heap varies dynamically according to a changing threshold that allows more contexts to be retained for promising nodes.

Table 13.6 illustrates how the depth of the ε-heap, the active states per frame of speech, word error rate, and search time change when the value of threshold ε increases for the 20,000-word WSJ dictation task. As we can see from the table, the average heap size for active nodes is only about 1.6 for the most accurate configuration. Figure 13.14(a) illustrates the distribution of stack depths for a large data set, showing that the stack depth is small even for tree initial nodes. Figure 13.14(b) illustrates the profile of the average stack depth for a sample utterance, showing that the average stack depth remains small across an utterance.

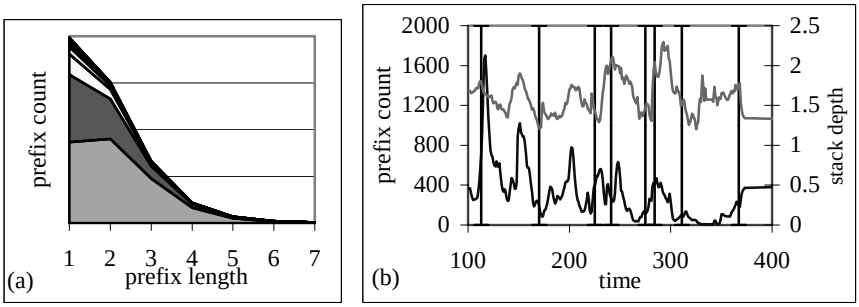

Figure 13.14 (a) A cumulative graph of the prefix count for each stack depth, starting with depth 1, showing the distribution according to prefix length. **(b)** The prefix count and the average stack depth with respect to one utterance. The vertical bars show the word boundaries [3].

Whisper also employs look-ahead techniques to further reduce the search effort. The acoustic look-ahead technique described in Section 13.2.3.1 attempts to estimate the probability that a phonetic HMM will participate in the final result [3]. Whisper implements acoustic look-ahead by running a CI phone-net synchronously with the search process but offset *N* frames in the future. One side effect of the acoustic look-ahead is to provide information for the RGR strategy, as described in Section 13.2.3.2, so the search can avoid unnecessary Gaussian computation. Figure 13.15 demonstrates the effectiveness of varying the frame look-ahead from 0 to *N* frames in terms of states evaluated.

When the look-ahead is increased from 0 to 3 frames, the search effort, in terms of real time, is reduced by ~40% with no loss in accuracy; however most of that is due to reducing the number of states evaluated per frame. There is no effect on the number of Gaussians evaluated per frame (the system using continuous density) until we begin to negatively impact error rate, indicating that the acoustic space represented by the pruned states is redundant and adequately covered by the retained states prior to the introduction of search errors.

With the techniques discussed here, Whisper is able to achieved real-time performance for the continuous WSJ dictation task (60,000-word) on Pentium-class PCs. The recognition accuracy is identical to that of a standard Viterbi beam decoder with a linear lexicon.

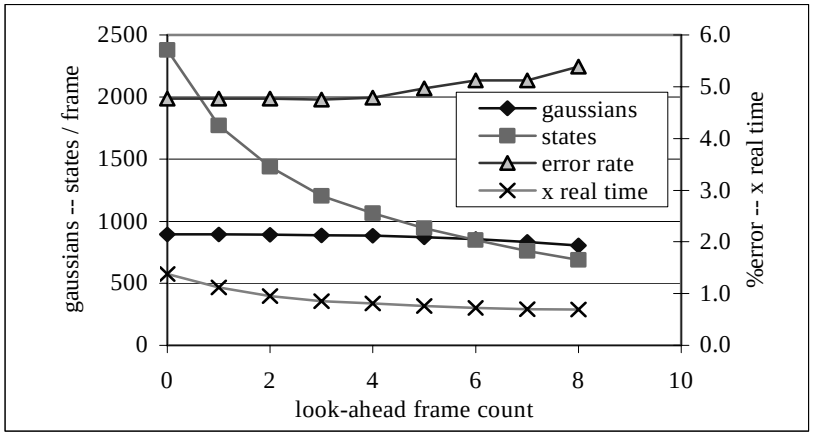

Figure 13.15 Search effort, percent error rate, and real-time factor as a function of the acoustic look-ahead. Note that search effort is the number of Gaussians evaluated per frame and the number of states evaluated per frame [3].

13.6. HISTORICAL PERSPECTIVES AND FURTHER READING

Large-vocabulary continuous speech recognition is a computationally intensive task. Real-time systems started to emerge in the late 1980s. Before that, most systems achieved real-time performance with the help of special hardware [11, 16, 25, 28]. Thanks to Moore's law and various efficient search techniques, real-time systems became a reality on a single-chip general-purpose personal computer in 1990s [4, 34, 43].

Common wisdom in 1980's saw stack decoding as more efficient for large-vocabulary continuous speech recognition with higher-order *n*-grams. Time-synchronous Viterbi beam search, as described in Sections 13.1 and 13.2, emerged as the most efficient search framework. It has become the most widely used search technique today. The lexical tree representation was first used by IBM as part of its allophonic fast match system [10]. Ney proposed the use of the lexical tree as the primary representation for the search space [32]. The ideas of language model factoring [4, 19] [5] and subtree polymorphism [4] enabled real-time single-pass search with higher-order language models (bigrams and trigrams). Alleva [3] and Ney [33] are two excellent articles regarding the detailed Viterbi beam search algorithm with lexical tree representation.

As mentioned in Chapter 12, fast match was first invented to speed up stack decoding [8, 9]. Ney et al. and Alleva et al. extended the fast match idea to phone look-ahead in time-synchronous search by using context-independent model evaluation. In Haeb-Umbach et al. [22] a word look-ahead is implemented for a 12.3k-word speaker-dependent continuous speech recognition task. The look-ahead is performed on a lexical tree, with beam search executed every other frame. The results show a factor of 3–5 of reduction for search space

compared to the standard Viterbi beam search, while only 1—2% extra errors are introduced by word look-ahead.

The idea of multipass search strategy has long existed for knowledge-based speech recognition systems [17], where first a phone recognizer is performed, then a lexicon hypothesizer is used to locate all the possible words to form a word lattice, and finally a language model is used to search for the most possible word sequence. However, HMM's popularity predominantly shifted the focus to the unified search approach to achieve global optimization. Computation concerns led many researchers to revisit the multipass search strategy. The first *n*-best algorithm, described in Section 13.3.2, was published by researchers at BBN [39]. Since then, *n*-best and word-lattice based multipass search strategies have become important search frameworks for rapid system deployment, research tools, and spoken language understanding systems. Schwartz et al.'s paper [40] is a good tutorial on the *n*-best or word-lattice generation algorithms. Most of the *n*-best search algorithms can be made to generate word lattices/graphs with minor modifications. Other excellent discussions of multipass search can be found in [14, 24, 30].

REFERENCES

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, 1974, Addison-Wesley Publishing Company.

[2] Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers : Principles, Techniques, and Tools*, 1985, Addison-Wesley.

[3] Alleva, F., "Search Organization in the Whisper Continuous Speech Recognition System," *IEEE Workshop on Automatic Speech Recognition*, 1997.

[4] Alleva, F., X. Huang, and M.Y. Hwang, "Improvements on the Pronunciation Prefix Tree Search Organization," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1996, Atlanta, Georgia pp. 133-136.

[5] Aubert, X., *et al.*, "Large Vocabulary Continuous Speech Recognition of Wall Street Journal Corpus," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1994, Adelaide, Australia pp. 129-132.

[6] Aubert, X. and H. Ney, "Large Vocabulary Continuous Speech Recognition Using Word Graphs," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, MI pp. 49-52.

[7] Austin, S., R. Schwartz, and P. Placeway, "The Forward-Backward Search Algorithm for Real-Time Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1991, Toronto, Canada pp. 697-700.

[8] Bahl, L.R., *et al.*, "Obtaining Candidate Words by Polling in a Large Vocabulary Speech Recognition System," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1988 pp. 489-492.

[9] Bahl, L.R., *et al.*, "Matrix Fast Match: a Fast Method for Identifying a Short List of Candidate Words for Decoding," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1989, Glasgow, Scotland pp. 345-347.

[10] Bahl, L.R., P.S. Gopalakrishnan, and R.L. Mercer, "Search Issues in Large Vocabulary Speech Recognition," *Proc. of the 1993 IEEE Workshop on Automatic Speech Recognition*, 1993, Snowbird, UT.

[11] Bisiani, R., T. Anantharaman, and L. Butcher, "BEAM: An Accelerator for Speech Recognition," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1989 pp. 782-784.

[12] Brugnara, F. and M. Cettolo, "Improvements in Tree-Based Language Model Representation," *Proc. of the European Conf. on Speech Communication and Technology*, 1995, Madrid, Spain pp. 1797-1800.

[13] Cettolo, M., R. Gretter, and R.D. Mori, "Knowledge Integration" in *Spoken Dialogues with Computers*, R.D. Mori, Editor 1998, London, pp. 231-256, Academic Press.

[14] Cettolo, M., R. Gretter, and R.D. Mori, "Search and Generation of Word Hypotheses" in *Spoken Dialogues with Computers*, R.D. Mori, Editor 1998, London, pp. 257-310, Academic Press.

[15] Chou, W., C.H. Lee, and B.H. Juang, "Minimum Error Rate Training Based on N-best String Models," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN pp. 652-655.

[16] Chow, Y.L., *et al.*, "BYBLOS: The BBN Continuous Speech Recognition System," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1987 pp. 89-92.

[17] Cole, R.A., *et al.*, "Feature-Based Speaker Independent Recognition of English Letters," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1983 pp. 731-734.

[18] Davenport, J.C., R. Schwartz, and L. Nguyen, "Towards A Robust Real-Time Decoder," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1999, Phoenix, Arizona pp. 645-648.

[19] Federico, M., *et al.*, "Language Modeling for Efficient Beam-Search," *Computer Speech and Language*, 1995, pp. 353-379.

[20] Gauvain, J.L., L. Lamel, and M. Adda-Decker, "Developments in Continuous Speech Dictation using the ARPA WSJ Task," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, MI pp. 65-68.

[21] Gillick, L.S. and R. Roth, "A Rapid Match Algorithm for Continuous Speech Recognition," *Proc. of the Speech and Natural Language Workshop*, 1990, Hidden Valley, PA pp. 170-172.

[22] Haeb-Umbach, R. and H. Ney, "A Look-Ahead Search Technique for Large Vocabulary Continuous Speech Recognition," *Proc. of the European Conf. on Speech Communication and Technology*, 1991, Genova, Italy pp. 495-498.

[23] Haeb-Umbach, R. and H. Ney, "Improvements In Time-Synchronous Beam-Search For 10000-Word Continuous Speech Recognition," *IEEE Trans. on Speech and Audio Processing*, 1994, **2**(4), pp. 353-365.

[24] Hetherington, I.L., *et al.*, "A* Word Network Search for Continuous Speech Recognition," *Proc. of the European Conf. on Speech Communication and Technology*, 1993, Berlin, Germany pp. 1533-1536.

[25] Hon, H.W., *A Survey of Hardware Architectures Designed for Speech Recognition*, 1991, Carnegie Mellon University, Pittsburgh, PA.

[26] Huang, X., *et al.*, "From Sphinx II to Whisper: Making Speech Recognition Usable" in *Automatic Speech and Speaker Recognition*, C.H. Lee, F.K. Soong, and K.K. Paliwal, eds. 1996, Norwell, MA, pp. 481-508, Kluwer Academic Publishers.

[27] Huang, X., *et al.*, "Microsoft Windows Highly Intelligent Speech Recognizer: Whisper," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995 pp. 93-96.

[28] Jelinek, F., "The Development of an Experimental Discrete Dictation Recognizer," *Proc. of the IEEE*, 1985, **73**(1), pp. 1616-1624.

[29] Marino, J. and E. Monte, "Generation of Multiple Hypothesis in Connected Phonetic-Unit Recognition by a Modified One-Stage Dynamic Programming Algorithm," *Proc. of EuroSpeech*, 1989, Paris pp. 408-411.

[30] Murveit, H., *et al.*, "Large Vocabulary Dictation Using SRI's DECIPHER Speech Recognition System: Progressive Search Techniques," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN pp. 319-322.

[31] Ney, H. and X. Aubert, "A Word Graph Algorithm for Large Vocabulary," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan pp. 1355-1358.

[32] Ney, H., *et al.*, "Improvements in Beam Search for 10000-Word Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1992, San Francisco, California pp. 9-12.

[33] Ney, H. and S. Ortmanns, *Dynamic Programming Search for Continuous Speech Recognition*, in *IEEE Signal Processing Magazine*, 1999. pp. 64-83.

[34] Nguyen, L., *et al.*, "Search Algorithms for Software-Only Real-Time Recognition with Very Large Vocabularies," *Proc. of ARPA Human Language Technology Workshop*, 1993, Plainsboro, NJ pp. 91-95.

[35] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence*, 1971, New York, McGraw-Hill.

[36] Normandin, Y., "Maximum Mutual Information Estimation of Hidden Markov Models" in *Automatic Speech and Speaker Recognition*, C.H. Lee, F.K. Soong, and K.K. Paliwal, eds. 1996, Norwell, MA, Kluwer Academic Publishers.

[37] Odell, J.J., *et al.*, "A One Pass Decoder Design for Large Vocabulary Recognition," *Proc. of the ARPA Human Language Technology Workshop*, 1994, Plainsboro, New Jersey pp. 380-385.

[38] Schwartz, R. and S. Austin, "A Comparison of Several Approximate Algorithms for Finding Multiple (N-BEST) Sentence Hypotheses," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1991, Toronto, Canada pp. 701-704.

[39] Schwartz, R. and Y.L. Chow, "The N-Best Algorithm: an Efficient and Exact Procedure for Finding the N Most Likely Sentence Hypotheses," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1990, Albuquerque, New Mexico pp. 81-84.

[40] Schwartz, R., L. Nguyen, and J. Makhoul, "Multiple-Pass Search Strategies" in *Automatic Speech and Speaker Recognition*, C.H. Lee, F.K. Soong, and K.K. Paliwal, eds. 1996, Norwell, MA, pp. 57-81, Klewer Academic Publishers.

[41] Soong, F.K. and E.F. Huang, "A Tree-Trellis Based Fast Search for Finding the N Best Sentence Hypotheses in Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1991, Toronto, Canada pp. 705-708.

[42] Steinbiss, V., "Sentence-Hypotheses Generation in a Continuous Speech Recognition Recognition," *Proc. of EuroSpeech*, 1989, Paris pp. 51-54.

[43] Steinbiss, V., *et al.*, "The Philips Research System for Large-Vocabulary Continuous-Speech Recognition," *Proc. of the European Conf. on Speech Communication and Technology*, 1993, Berlin, Germany pp. 2125-2128.

[44] Woodland, P.C., *et al.*, "Large Vocabulary Continuous Speech Recognition Using HTK," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1994, Adelaide, Australia pp. 125-128.