synchronous search framework makes beam search possible. That is, for two different word sequences $\mathbf{W}_1$ and $\mathbf{W}_2$, the posterior probabilities $P(\mathbf{W}_1 \mid \mathbf{x}_0^t)$ and $P(\mathbf{W}_2 \mid \mathbf{x}_0^t)$ are always compared based on the same partial acoustic observation $\mathbf{x}_0^t$. This makes the comparison straightforward because the denominator $P(\mathbf{x}_0^t)$ in Eq. (12.5) is the same for both terms and can be ignored. Since the score comparison for each time frame is fair, the only assumption of beam search is that an optimal path should have a good enough partial-path score for each time frame to survive under beam pruning.

The time-synchronous framework is one of the aspects of Viterbi beam search that is critical to its success. Unlike the time-synchronous framework, time-asynchronous search algorithms such as stack decoding require the normalization of likelihood scores over feature streams of different time lengths. This, as we will see in Section 12.5, is the Achilles' heel of that approach.

The straightforward time-synchronous Viterbi beam search is ineffective in dealing with the gigantic search space of high perplexity tasks. However, with a better understanding of the linguistic search space and the advent of techniques for obtaining n-best lists from time-synchronous Viterbi search, described in Chapter 13, time-synchronous Viterbi beam search has turned out to be surprisingly successful in handling tasks of all sizes and all different types of grammars, including FSG, CFG, and n-gram [2, 14, 18, 28, 34, 38, 44]. Therefore, it has become the predominant search strategy for continuous speech recognition.

12.4.2. Viterbi Beam Search

To explain the time-synchronous Viterbi beam search in a formal way [31], we first define some quantities:

$D(t;s_t;w) \equiv$ total cost of the best path up to time t that ends in state s_t of grammar word state w.

$h(t;s_t;w) \equiv$ backtrack pointer for the best path up to time t that ends in state s_t of grammar word state w.

Readers should be aware that w in the two quantities above represents a grammar word state in the search space. It is different from just the word identity since the same word could occur in many different language model states, as in the trigram search space shown in Figure 12.17.

There are two types of dynamic programming (DP) transition rules [30], namely intra-word and inter-word transition. The intra-word transition is just like the Viterbi rule for HMMs and can be expressed as follows:

$$D(t;s_t;w) = \min_{s_{t-1}} \{d(\mathbf{x}_t, s_t \mid s_{t-1};w) + D(t-1;s_{t-1};w)\} \tag{12.17}$$

$$h(t;s_t;w) = h(t-1, b_{\min}(t;s_t;w);w) \tag{12.18}$$

Amazon/VB Assets
Exhibit 1011
Page 651

where $d(\mathbf{x}_t, s_t \mid s_{t-1}; w)$ is the cost associated with taking the transition from state s_{t-1} to state s_t while generating output observation $\mathbf{x}_t$, and $b_{\min}(t; s_t; w)$ is the optimal predecessor state of cell $D(t; s_t; w)$. To be specific, they can be expressed as follows:

$$d(\mathbf{x}_t, s_t \mid s_{t-1}; w) = -\log P(s_t \mid s_{t-1}; w) - \log P(\mathbf{x}_t \mid s_t; w) \tag{12.19}$$

$$b_{\min}(t; s_t; w) = \arg\min_{s_{t-1}} \left\{ d(\mathbf{x}_t, s_t \mid s_{t-1}; w) + D(t-1; s_{t-1}; w) \right\} \tag{12.20}$$

The inter-word transition is basically a null transition without consuming any observation. However, it needs to deal with creating a new history node for the backtracking pointer. Let's define $F(w)$ as the final state of word HMM w and $I(w)$ as the initial state of word HMM w. Moreover, state η is denoted as the pseudo initial state. The inter-word transition can then be expressed as follows:

$$D(t; \eta; w) = \min_{v} \left\{ \log P(w \mid v) + D(t; F(v); v) \right\} \tag{12.21}$$

$$h(t; \eta; w) = \langle v_{\min}, t \rangle :: h(t, F(v_{\min}); v_{\min}) \tag{12.22}$$

where $v_{\min} = \arg\min_{v} \left\{ \log P(w \mid v) + D(t; F(v); v) \right\}$ and :: is a link appending operator.

The time-synchronous Viterbi beam search algorithm assumes that all the intra-word transitions are evaluated before inter-word null transitions take place. The same time index is used intentionally for inter-word transition since the null language model state transition does not consume an observation vector. Since the initial state $I(w)$ for word HMM w could have a self-transition, the cell $D(t; I(w); w)$ might already have an active path. Therefore, we need to perform the following check to advance the inter-word transitions.

$$\begin{aligned} &\text{if } D(t; \eta; w) < D(t; I(w); w) \\ &\quad D(t; I(w); w) = D(t; \eta; w) \text{ and } h(t; I(w); w) = h(t; \eta; w) \end{aligned} \tag{12.23}$$

The time-synchronous Viterbi beam search can be summarized as in Algorithm 12.6. For large-vocabulary speech recognition, the experimental results show that only a small percentage of the entire search space (the beam) needs to be kept for each time interval t without increasing error rates. Empirically, the beam size has typically been found to be between 5% and 10% of the entire search space. In Chapter 13 we describe strategies of using different level of beams for more effectively pruning.

12.5. STACK DECODING (A* SEARCH)

If some reliable heuristics are available to guide the decoding, the search can be done in a depth-first fashion around the best path early on, instead of wasting efforts on unpromising paths via the time-synchronous beam search. Stack decoding represents the best attempt to

Amazon/VB Assets
Exhibit 1011
Page 652

ALGORITHM 12.6: *TIME-SYNCHRONOUS VITERBI BEAM SEARCH*

Step 1: Initialization: For all the grammar word states w which can start a sentence,

$$D(0;I(w);w) = 0$$

$$h(0;I(w);w) = null$$

Step 2: Induction: For time $t = 1$ to T do

For all active states do

Intra-word transitions according to Eq. (12.17) and (12.18)

$$D(t;s_t;w) = \min_{s_{t-1}} \left\{ d(\mathbf{x}_t, s_t \mid s_{t-1};w) + D(t-1;s_{t-1};w) \right\}$$

$$h(t;s_t;w) = h(t-1, b_{\min}(t;s_t;w);w)$$

For all active word-final states do

Inter-word transitions according to Eq. (12.21), (12.22) and (12.23)

$$D(t;\eta;w) = \min_{v} \left\{ \log P(w \mid v) + D(t;F(v);v) \right\}$$

$$h(t;\eta;w) = \langle v_{\min}, t \rangle :: h(t, F(v_{\min});v_{\min})$$

if $D(t;\eta;w) < D(t;I(w);w)$

$$D(t;I(w);w) = D(t;\eta;w) \text{ and } h(t;I(w);w) = h(t;\eta;w)$$

Pruning: Find the cost for the best path and decide the beam threshold

Prune unpromising hypotheses

Step 3: Termination: Pick the best path among all the possible final states of grammar at time T. Obtain the optimal word sequence according to the backtracking pointer $h(t;\eta;w)$

use A* search instead of time-synchronous beam search for continuous speech recognition. Unfortunately, as we will discover in this section, such a heuristic function $h(\bullet)$ (defined in Section 12.1.3) is very difficult to attain in continuous speech recognition, so search algorithms based on A* search are in general less efficient than time-synchronous beam search.

Stack decoding is a variant of the heuristic A* search based on the forward algorithm, where the evaluation function is based on the forward probability. It is a tree search algorithm, which takes a slightly different viewpoint than the time-synchronous Viterbi search. Time-synchronous beam search is basically a breadth-first search, so it is crucial to control the number of all possible language model states as described in Section 12.3. In a typical large-vocabulary Viterbi search with n-gram language models, this number is determined by the equivalent classes of language model histories. On the other hand, stack decoding as a tree search algorithm treats the search as a task for finding a path in a tree whose branches correspond to words in the vocabulary V, non-terminal nodes correspond to incomplete sentences, and terminal nodes correspond to complete sentences. The search tree has a constant branching factor of $|V|$, if we allow every word to be followed by every word. Figure 12.20 illustrates such a search tree for a vocabulary with three words [19].

Amazon/VB Assets
Exhibit 1011
Page 653

An important advantage of stack decoding is its consistency with the forward-backward training algorithm. Viterbi search is a graph search, and paths cannot be easily summed because they may have different word histories. In general, the Viterbi search finds the optimal state sequence instead of optimal word sequence. Therefore, Viterbi approximation is necessary to make the Viterbi search feasible, as described in Section 12.4. Stack decoding is a tree search, so each node has a unique history, and the forward algorithm can be used within word model evaluation. Moreover, all possible beginning and ending times (shaded areas in Figure 12.21) are considered [24]. With stack decoding, it is possible to use an objective function that searches for the optimal word string, rather than the optimal state sequence. Furthermore, it is in principle natural for stack decoding to accommodate long-range language models if the heuristics can guide the search to avoid exploring the overwhelmingly large unpromising grammar states.

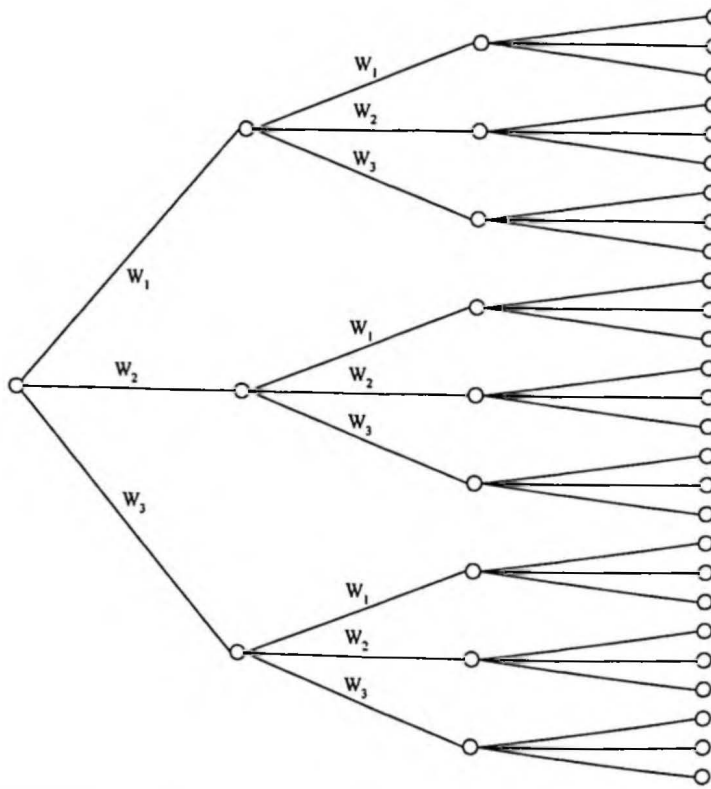

Figure 12.20 A stack decoding search tree for a vocabulary size of three [19].

Amazon/VB Assets
Exhibit 1011
Page 654

By formulating stack decoding in a tree search framework, the graph search algorithms described in Section 12.1 can be directly applied to stack decoding. Obviously, blind-search methods, like depth-first and breadth-first search, that do not take advantage of the goodness measurement of how close we are getting to the goal, are usually computationally infeasible in practical speech recognition systems. A* search is clearly attractive for speech recognition, given the hope of a sufficient heuristic function to guide the tree search in a favorable direction without exploring too many unpromising branches and nodes. In contrast to the Viterbi search, it is not time-synchronous and extends paths of different lengths.

The search begins by adding all possible one-word hypotheses to the *OPEN* list. Then the best path is removed from the *OPEN* list, and all paths from it are extended, evaluated, and placed back in the *OPEN* list. This search continues until a complete path that is guaranteed to be better than all paths in the *OPEN* list has been found.

Unlike Viterbi search, where the acoustic probabilities being compared are always based on the same partial input, it is necessary to compare the goodness of partial paths of different lengths to direct the A* tree search. Moreover, since stack decoding is done asynchronously, we need an effective mechanism to determine when to end a phone/word evaluation and move on to the next phone/word. Therefore, the heart and soul of the stack decoding are clearly in

1. Finding an effective and efficient heuristic function for estimating the future remaining input feature stream and
2. Determining when to extend the search to the next word/phone.

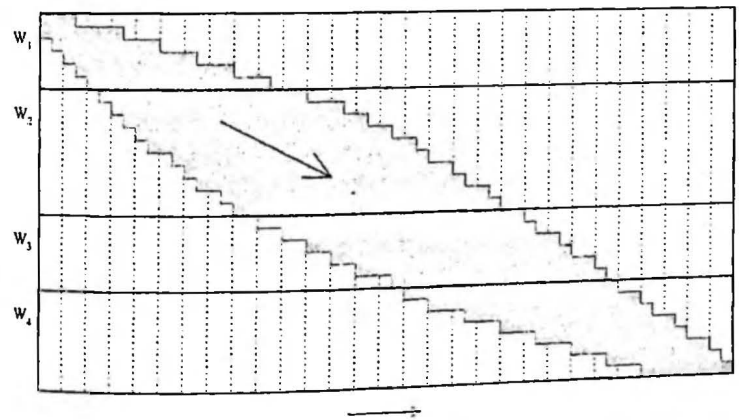

Figure 12.21 The forward trellis space for stack decoding. Each grid point corresponds to a trellis cell in the forward computation. The shaded area represents the values contributing to the computation of the forward score for the optimal word sequence $w_1, w_2, w_3, \ldots$ [24].

Amazon/VB Assets
Exhibit 1011
Page 655

In the following section we describe these two critical components. Readers will note that the solutions to these two issues are virtually the same—using a normalization scheme to compare paths of different lengths.

12.5.1. Admissible Heuristics for Remaining Path

The key issue in heuristic search is the selection of an evaluation function. As described in Section 12.1.3, the heuristic function of the path H_N going through node N includes the cost up to the node and the estimate of the cost to the target node from node N. Suppose path H_N is going through node N at time t; then the evaluation for path H_N can be expressed as follows:

$$f(H_N^t) = g(H_N^t) + h(H_N^{t,T}) \tag{12.24}$$

where $g(H_N^t)$ is the evaluation function for the partial path of H_N up to time t, and $h(H_N^{t,T})$ is the heuristic function of the remaining path from $t+1$ to T for path H_N. The challenge for stack decoders is to devise an admissible function for $h(\bullet)$.

According to Section 12.1.3.1, an admissible heuristic function is one that always underestimates the true cost of the remaining path from $t+1$ to T for path H_N. A trivially admissible function is the zero function. In this case, it results in a very large OPEN list. In addition, since the longer paths tend to have higher cost because of the gradually accumulated cost, the search is likely to be conducted in a breadth-first fashion, which functions very much like a plain Viterbi search. The evaluation function $g(\bullet)$ can be obtained easily by using the HMM forward score as the true cost up to current time t. However, how can we find an admissible heuristic function $h(\bullet)$? We present the basic concept here [19, 35].

The goal of $h(\bullet)$ is to find the expected cost for the remaining path. If we can obtain the expected cost per frame ψ for the remaining path, the total expected cost, $(T-t)*\psi$, is simply the product of ψ and the length of the remaining path. One way to find such expected cost per frame is to gather statistics empirically from training data.

1. After the final training iteration, perform Viterbi forced alignment[9] with each training utterance to get an optimal time alignment for each word.
2. Randomly select an interval to cover the number of words ranging from two to ten. Denote this interval as $[i \ldots j]$.
3. Compute the average acoustic cost per frame within this selected interval according to the following formula and save the value in a set Λ:

[9] Viterbi forced alignment means that the Viterbi is performed on the HMM model constructed from the known word transcription. The term "forced" is used because the Viterbi alignment is forced to be performed on the correct model. Viterbi forced alignment is a very useful tool in spoken language processing as it can provide the optimal state-time alignment with the utterances. This detailed alignment can then be used for different purposes, including discriminant training, concatenated speech synthesis, etc.

Amazon/VB Assets
Exhibit 1011
Page 656

$$\frac{-1}{j-i}\log P(\mathbf{x}_i^j \mid \mathbf{w}_{i\ldots j}) \tag{12.25}$$

where $\mathbf{w}_{i\ldots j}$ is the word string corresponding to interval $[i\ldots j]$.

4. Repeat Steps 2 and 3 for the entire training set.

5. Define $\psi_{\min}$ and ψ_{avg} as the minimum and average value found in set Λ.

Clearly, $\psi_{\min}$ should be a good under-estimate of the expected cost per frame for the future unknown path. Therefore, the heuristic function $h(H_N^{t,T})$ can be derived as:

$$h(H_N^{t,T}) = (T-t)\psi_{\min} \tag{12.26}$$

Although $\psi_{\min}$ is obtained empirically, stack decoding based on Eq. (12.26) will generally find the optimal solution. However, the search using $\psi_{\min}$ usually runs very slowly, since Eq. (12.26) always under-estimates the true cost for any portion of speech. In practice, a heuristic function like ψ_{avg} that may over-estimate has to be used to prune more hypotheses. This speeds up the search at the expense of possible search errors, because ψ_{avg} should represent the average cost per frame for any portion of speech. In fact, there is an argument that one might be able to use a heuristic function even more than ψ_{avg}. The argument is that ψ_{avg} is derived from the correct path (training data) and the average cost per frame for all paths during search should be more than ψ_{avg} because the paths undoubtedly include correct and incorrect ones.

12.5.2. When to Extend New Words

Since stack decoding is executed asynchronously, it becomes necessary to detect when a phone/word ends, so that the search can extend to the next phone/word. If we have a cost measure that indicates how well an input feature vector of any length matches the evaluated model sequence, this cost measure should drop slowly for the correct phone/word and rise sharply for an incorrect phone/word. In order to do so, it implies we must be able to compare hypotheses of different lengths.

The first thing that comes to mind for this cost measure is simply the forward cost $-\log P(\mathbf{x}_1^t, s_t \mid w_1^k)$, which represents the likelihood of producing acoustic observation $\mathbf{x}_1^t$ based on word sequence w_1^k and ending at state s_t. However, it is definitely not suitable because it is deemed to be smaller for a shorter acoustic input vector. This causes the search to almost always prefer short phones/words, resulting in many insertion errors. Therefore, we must derive some normalized score that satisfies the desired property described above. The normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ can be represented as follows [6, 24]:

$$\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k) = -\log\left[\frac{P(\mathbf{x}_1^t, s_t \mid w_1^k)}{\gamma^t}\right] = -\log\left[P(\mathbf{x}_1^t, s_t \mid w_1^k)\right] + t\log\gamma \tag{12.27}$$

where γ $(0<\gamma<1)$ is a constant normalization factor.

Amazon/VB Assets
Exhibit 1011
Page 657

Suppose the search is now evaluating a particular word w_k; we can define $\hat{C}_{\min}(t)$ as the minimum cost for $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ for all the states of w_k, and $\alpha_{\max}(t)$ as the maximum forward probability for $P(\mathbf{x}_1^t, s_t \mid w_1^k)$ for all the states of w_k. That is,

$$\hat{C}_{\min}(t) = \min_{s_t \in w_k} \left[\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k) \right] \tag{12.28}$$

$$\alpha_{\max}(t) = \max_{s_t \in w_k} \left[P(\mathbf{x}_1^t \mid w_1^k, s_t) \right] \tag{12.29}$$

We want $\hat{C}_{\min}(t)$ to be near 0 just as long as the phone/word we are evaluating is the correct one and we have not gone beyond its end. On the other hand, if the phone/word we are evaluating is the incorrect one or we have already passed its end, we want the $\hat{C}_{\min}(t)$ to be rising sharply. Similar to the procedure of finding the admissible heuristic function, we can set the normalized factor γ empirically during training so that $\hat{C}_{\min}(T) = 0$ when we know the correct word sequence $\mathbf{W}$ that produces acoustic observation sequence $\mathbf{x}_1^T$. Based on Eq. (12.27), γ should be set to:

$$\gamma = \sqrt[T]{\alpha_{\max}(T)} \tag{12.30}$$

Figure 12.22 shows a plot of $\hat{C}_{\min}(t)$ as a function of time for correct match. In addition, the cost for the final state $FS(w_k)$ of word w_k, $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$, which is the score for w_k-ending path, is also plotted. There should be a valley centered around 0 for $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$, which indicates the region of possible ending time for the correct phone/word. Sometimes a stretch of acoustic observations may match better than the average cost, pushing the curve below 0. Similarly, a stretch of acoustic observations may match worse than the average cost, pushing the curve above 0.

There is an interesting connection between the normalized factor γ and the heuristic estimate of the expected cost per frame, ψ, defined in Eq. (12.25). Since the cost is simply the logarithm on the inverse posterior probability, we get the following equation:

$$\psi = \frac{-1}{T} \log P(\mathbf{x}_1^T \mid \hat{\mathbf{W}}) = -\log \left[\alpha_{\max}(T)^{1/T} \right] = -\log \gamma \tag{12.31}$$

Equation (12.31) reveals that these two quantities are basically the same estimate. In fact, if we subtract the heuristic function $f(H_N^t)$ defined in Eq. (12.24) by the constant $\log(\gamma^T)$, we get exactly the same quantity as the one defined in Eq. (12.27). Decisions on which path to extend first based on the heuristic function and when to extend the search to the next word/phone are basically centered on comparing partial theories with different lengths. Therefore, the normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ can be used for both purposes.

Based on the connection we have established, the heuristic function, $f(H_N^t)$, which estimates the goodness of a path is simply replaced by the normalized evaluation function $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$. If we plot the un-normalized cost $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ for the optimal path and other

Amazon/VB Assets
Exhibit 1011
Page 658

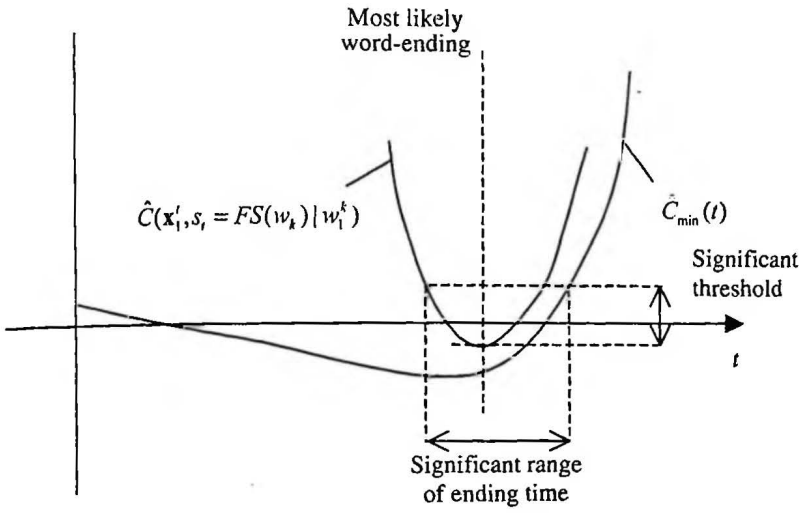

Figure 12.22 $\hat{C}_{\min}(t)$ and $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$ as functions of time t. The valley region represents possible ending times for the correct phone/word.

competing paths as the function time t, the cost values increase as paths get longer (illustrated in Figure 12.23) because every frame adds some non-negative cost to the overall cost. It is clear that using un-normalized cost function $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ generally results in a breadth-first search. What we want is an evaluation that decreases slightly along the optimal path, and hopefully increases along other competing paths. Clearly, the normalized cost function $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ fulfills this role, as shown in Figure 12.24.

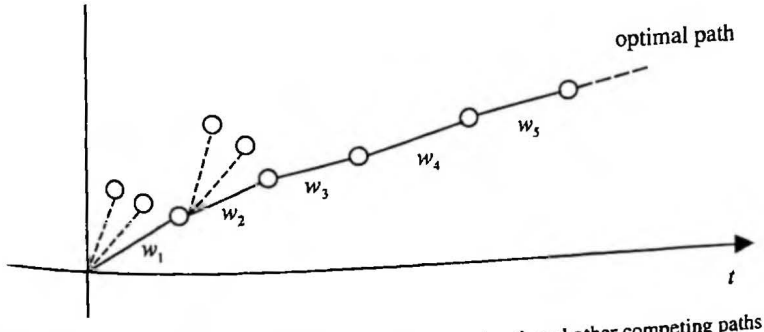

Figure 12.23 Unnormalized cost $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ for optimal path and other competing paths as a function of time.

Amazon/VB Assets
Exhibit 1011
Page 659

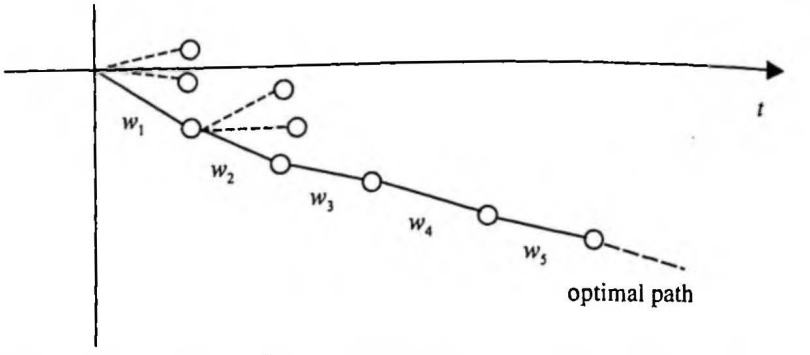

Figure 12.24 Normalized cost $\hat{C}(\mathbf{x}_1^t, s_i \mid w_1^k)$ for the optimal path and other competing paths as a function of time.

Equation (12.30) is a context-less estimation of the normalized factor, which is also referred to as zero-order estimate. To improve the accuracy of the estimate, you can use context-dependent higher-order estimates like [24]:

$\gamma_i = \gamma(\mathbf{x}_i)$	first-order estimate
$\gamma_i = \gamma(\mathbf{x}_i, \mathbf{x}_{i-1})$	second-order estimate
$\gamma_i = \gamma(\mathbf{x}_i, \mathbf{x}_{i-1}, \ldots, \mathbf{x}_{i-N+1})$	n-order estimate

Since the normalized factor γ is estimated from the training data that is also used to train the parameters of the HMMs, the normalized factor γ_i tends to be an overestimate. As a result, $\alpha_{\max}(t)$ might rise slowly for test data even when the correct phone/word model is evaluated. This problem is alleviated by introducing some other scaling factor $\delta < 1$ so that $\alpha_{\max}(t)$ falls slowly for test data for when evaluating the correct phone/word model. The best solution for this problem is to use an independent data set other than the training data to derive the normalized factor γ_i.

12.5.3. Fast Match

Even with an efficient heuristic function and mechanism to determine the ending time for a phone/word, stack decoding could still be too slow for large-vocabulary speech recognition tasks. As described in Section 12.5.1, an effective underestimated heuristic function for the remaining portion of speech is very difficult to derive. On the other hand, a heuristic estimate for the immediate short segment that usually corresponds to the next phone or word may be feasible to attain. In this section, we describe the fast-match mechanism that reduces phone/word candidates for detailed match (expansion).

In asynchronous stack decoding, the most expensive step is to extend the best subpath. For a large-vocabulary search, it implies the calculation of $P(\mathbf{x}_t^{t+k} \mid w)$ over the entire vocabulary size $|V|$. It is desirable to have a fast computation to quickly reduce the possible

Amazon/VB Assets
Exhibit 1011
Page 660

words starting at a given time t to reduce the search space. This process is often referred to as *fast match* [15, 35]. In fact, fast match is crucial to stack decoding, of which it becomes an integral part. Fast match is a method for the rapid computation of a list of candidates that constrain successive search phases. The expensive *detailed match* can then be performed after fast match. In this sense, fast match can be regarded as an additional pruning threshold to meet before new word/phone can be started.

Fast match, by definition, needs to use only a small amount of computation. However, it should also be accurate enough not to prune away any word/phone candidates that participate in the best path eventually. Fast match is, in general, characterized by the approximations that are made in the acoustic/language models in order to reduce computation. There is an obvious trade-off between these two objectives. Fortunately, many systems [15] have demonstrated that one needs to sacrifice very little accuracy in order to speed up the computation considerably.

Similar to *admissibility* in A* search, there is also an *admissibility* property in fast match. A fast match method is called admissible if it never prunes away the word/phone candidates that participate in the optimal path. In other words, a fast match is admissible if the recognition errors that appear in a system using the fast match followed by a detailed match are those that would appear if the detailed match were carried out for all words/phones in the vocabulary. Since fast match can be applied to either word or phone level, as we describe in the next section, we explain the admissibility for the case of word-level fast match for simplicity. The same principle can be easily extended to phone-level fast match.

Let V be the vocabulary and $C(\mathbf{X} \mid w)$ be the cost of a detailed match between input $\mathbf{X}$ and word w. Now $F(\mathbf{X} \mid w)$ is an estimator of $C(\mathbf{X} \mid w)$ that is accurate enough and fast to compute. A word list selected by fast match estimator can be attained by first computing $F(\mathbf{X} \mid w)$ for each word w of the vocabulary. Suppose w_b is the word for which the fast match has a minimum cost value:

$$w_b = \arg\min_{w \in V} F(\mathbf{X} \mid w) \tag{12.32}$$

After computing $C(\mathbf{X} \mid w_b)$, the detailed match cost for w_b, we form the fast match word list, Λ, from the word w in the vocabulary such that $F(\mathbf{X} \mid w)$ is no greater than $C(\mathbf{X} \mid w_b)$. In other words,

$$\Lambda = \{w \in V \mid F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w_b)\} \tag{12.33}$$

Similar to the admissibility condition for A* search [3, 33], the fast match estimator $F(\bullet)$ conducted in the way described above is admissible if and only if $F(\mathbf{X} \mid w)$ is always an under-estimator (lower bound) of detailed match $C(\mathbf{X} \mid w)$. That is,

$$F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w) \qquad \forall \mathbf{X}, w \tag{12.34}$$

The proof is straightforward. If the word w_c has a lower detailed match cost $C(\mathbf{X} \mid w_c)$, you can prove that it must be included in the fast match list Λ because

Amazon/VB Assets
Exhibit 1011
Page 661

$$C(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_b) \text{ and } F(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_c) \Rightarrow F(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_b)$$

Therefore, based on the definition of Λ, $w_c \in \Lambda$.

Now the task is to find an admissible fast match estimator. Bahl et al. [6] proposed one fast match approximation for discrete HMMs. As we will see later, this fast match approximation is indeed equivalent to a simplification of the HMM structure. Given the HMM for word w and an input sequence x_1^T of codebook symbols describing the input signal, the probability that the HMM w produces the VQ sequence x_1^T is given by (according to Chapter 8):

$$P(x_1^T \mid w) = \sum_{s_1, s_2, \ldots s_T} \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \tag{12.35}$$

Since we often use Viterbi approximation instead of the forward probability, the equation above can be approximated by:

$$P(x_1^T \mid w) \cong \max_{s_1, s_2, \ldots s_T} \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \tag{12.36}$$

The detailed match cost $C(\mathbf{X} \mid w)$ can now be represented as:

$$C(\mathbf{X} \mid w) = \min_{s_1, s_2, \ldots s_T} \left\{ -\log \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \right\} \tag{12.37}$$

Since the codebook size is finite, it is possible to compute, for each model w, the highest output probability for every VQ label c among all states s_k in HMM w. Let's define $m_w(c)$ to be the following:

$$m_w(c) = \max_{s_k \in w} P_w(c \mid s_k) = \max_{s_k \in w} b_k(c) \tag{12.38}$$

We can further define the $q_{\max}(w)$ as the maximum state sequence with respect to T, i.e., the maximum probability of any complete path in HMM w.

$$q_{\max}(w) = \max_{T} \left[P_w(s_1, s_2, \ldots s_T) \right] \tag{12.39}$$

Now let's define the fast match estimator $F(\mathbf{A} \mid w)$ as the following:

$$F(\mathbf{X} \mid w) = -\log \left[q_{\max}(w) \prod_{i=1}^{T} m_w(x_i) \right] \tag{12.40}$$

It is easy to show the fast match estimator $F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w)$ is admissible based on Eq. (12.38) to Eq. (12.40).

Amazon/VB Assets
Exhibit 1011
Page 662

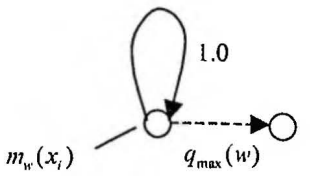

Figure 12.25 The equivalent one-state HMM corresponding to fast match computation defined in Eq. (12.40) [15].

The fast match estimator defined in Eq. (12.40) requires $T+1$ additions for a vector sequence of length T. The operation can be viewed as equivalent to the forward computation with a one-state HMM of the form shown in Figure 12.25. This correspondence can be interpreted as a simplification of the original multiple-state HMM into such a one-state HMM. It thus explains why fast match can be computed much faster than detailed match. Readers should note that this HMM is not actually a true HMM by strict definition, because the output probability distribution $m_w(c)$ and the transition probability distribution do not add up to one.

The fast match computation defined in Eq. (12.40) discards the sequence information with the model unit since the computation is independent of the order of input vectors. Therefore, one needs to decide the acoustic unit for fast match. In general, the longer the unit, the faster the computation is. and, therefore, the smaller the under-estimated cost $F(\mathbf{X}|w)$ is. It thus becomes a trade-off between accuracy and speed.

Now let's analyze the real speedup by using fast match to reduce the vocabulary V to the list Λ, followed by the detailed match. Let $|V|$ and $|\Lambda|$ be the sizes for the vocabulary V and the fast match short list Λ. Suppose t_f and t_d are the times required to compute one fast match score and one detailed match score for one word, respectively. Then, the total time required for the fast match followed by the detailed match is $t_f|V|+t_d|\Lambda|$, whereas the time required in doing the detailed match alone for the entire vocabulary is $t_d|V|$. The speed-up ratio is then given as follows:

$$\frac{1}{\left(\frac{t_f}{t_d}+\frac{|\Lambda|}{|V|}\right)} \tag{12.41}$$

We need t_f to be much smaller than t_d and $|\Lambda|$ to be much smaller than $|V|$ to have a significant speed-up using fast match. Using our admissible fast match estimator in Eq. (12.40), the time complexity of the computation for $F(\mathbf{X}|w)$ is T instead of N^2T for $C(\mathbf{X}|w)$, where N is the number of states in the detailed acoustic model. Therefore, the t_f/t_d saving is about N^2.

In general, in order to make $|\Lambda|$ much smaller than $|V|$, one needs a very accurate fast match estimator that could result in $t_f \approx t_d$. This is why we often relax the constraint of admissibility, although it is a nice principle to adhere to. In practice, most real-time speech recognition systems don't necessarily obey the admissibility principle with the fast match. For example, Bahl et al. [10], Laface et al., [22] and Roe et al., [36] used several techniques

Amazon/VB Assets
Exhibit 1011
Page 663

to construct off-line groups of acoustically similar words. Armed with this grouping, they can use an aggressive fast match to select only a very short list of words, and words acoustically similar to the words in this list are added to form the short word list Λ for further detailed match processing. By doing so, they are able to report a very efficient fast match method that misses the correct word only 2% of the time. When non-admissible fast match is used, one needs to minimize the additional search error introduced by fast match empirically.

Bahl et al. [6] use a one-state HMM as their fast match units and a tree-structure lexicon similar to the lexical tree structures introduced in Chapter 13 to construct the short word list Λ for next-word expansion in stack decoding. Since the fast match tree search is also done in an asynchronous way, the ending time of each phone is detected using normalized scores similar to those described in Section 12.5.2. It is based on the same idea that this normalized score rises slowly for the correct phone, while it drops rapidly once the end of phone is encountered (so the model is starting to go toward the incorrect phones). During the asynchronous lexical tree search, the unpromising hypotheses are also pruned away by a pruning threshold that is constantly changing once a complete hypothesis (a leaf node) is obtained. On a 20,000-word dictation task, such a fast match scheme was about 100 times faster than detailed match and achieved real-time performance on a commercial workstation with only 0.34% increase in the word error rate being introduced by the fast match process.

12.5.4. Stack Pruning

Even with efficient heuristic functions, the mechanism to determine the ending time for phone/word, and fast match, stack decoding might still be too slow for large-vocabulary speech recognition tasks. A beam within the stack, which saves only a small number of promising hypotheses in the *OPEN* list, is often used to reduce search effort. This *stack pruning* is very similar to beam search. A predetermined threshold ε is used to eliminate hypotheses whose cost value is much worse than the best path so far.

Both fast match and stack pruning could introduce search errors where the eventual optimal path is thrown away prematurely. However, the impact could be reduced to a minimum by empirically adjusting the thresholds in both methods.

The implementation of stack decoding is, in general, more complicated, particularly when some inevitable pruning strategies are incorporated to make the search more efficient. The difficulty of devising both an effectively admissible heuristic function for $h(\bullet)$ and an effective estimation of normalization factors for boundary determination has limited the advantage that stack decoders have over Viterbi decoders. Unlike stack decoding, time-synchronous Viterbi beam search can use an easy comparison of same-length path without heuristic determination of word boundaries. As described in the earlier sections, these simple and unified features of Viterbi beam search allow researchers to incorporate various sound techniques to improve the efficiency of search. Therefore, time-synchronous Viterbi Beam search enjoys a much broader popularity in the speech community. However, the principle of stack decoding is essential particularly for n-best and lattice search. As we describe in Chapter 13, stack decoding plays a very crucial part in multiple-pass search strate-

Amazon/VB Assets
Exhibit 1011
Page 664

gies for n-best and lattice search because the early pass is able to establish a near-perfect estimate of the remaining path.

12.5.5. Multistack Search

Even with the help of normalized factor γ or heuristic function $h(\bullet)$, it is still more effective to compare hypotheses of the same length than those of different lengths, because hypotheses with the same length are compared based on the true forward matching score. Inspired by the time-synchronous principle in Viterbi beam search, researchers [8, 35] propose a variant stack decoding based on multiple stacks.

Multistack search is equivalent to a best-first search algorithm running on multiple stacks time-synchronously. Basically, the search maintains a separate stack for each time frame t, so it never needs to compare hypotheses of different lengths. The search runs time-synchronously from left to right just like time-synchronous Viterbi search. For each time frame t, multistack search extracts the best path out of the t-stack, computes one-word extensions, and places all the new paths into the corresponding stacks. When the search finishes, the top path in the last stack is our optimal path. Algorithm 12.7 illustrates the multistack search algorithm.

This time-synchronous multistack search is designed based on the fact that by the time the t^{th} stack is extended, it already contains the best paths that could ever be placed into it. This phenomenon is virtually a variant of the dynamic programming principle introduced in Chapter 8. To make multistack more efficient, some heuristic pruning can be applied to reduce the computation. For example, when the top path of each stack is extended for one more word, we could only consider extensions between minimum and maximum duration. On the other hand, when some heuristic pruning is integrated into the multistack search, one might need to use a small beam in Step 2 of Algorithm 12.7 to extend more than just the best path to guarantee the admissibility.

ALGORITHM 12.7: *MULTISTACK SEARCH*

Step 1: Initialization: for each word v in vocabulary V

for $t = 1, 2, \ldots, T$

Compute $C(\mathbf{x}_1^t \mid v)$ and insert it to t^{th} stack

Step 2: Iteration: for $t = 1, 2, \ldots, T-1$

Sort the t^{th} stack and pop the top path $C(\mathbf{x}_1^t \mid w_1^k)$ out of the stack

for each word v in vocabulary V

for $\tau = t+1, t+2, \ldots, T$

Extend the path $C(\mathbf{x}_1^t \mid w_1^k)$ by word v to get $C(\mathbf{x}_1^\tau \mid w_1^{k+1})$

where $w_1^{k+1} = w_1^k \,||\, v$ and || means string concatenation

Place $C(\mathbf{x}_1^\tau \mid w_1^{k+1})$ in τ^{th} stack

Step 3: Termination: Sort the T^{th} stack and the top path is the optimal word sequence

Amazon/VB Assets
Exhibit 1011
Page 665

12.6. HISTORICAL PERSPECTIVE AND FURTHER READING

Search has been one of the most important topics in artificial intelligence since the origins of the field. It plays the central role in general problem solving [29] and computer games. [43], Nilsson's *Principles of Artificial Intelligence* [32] and Barr and Feigenbaum's *The Handbook of Artificial Intelligence* [11] contain a comprehensive introduction to state-space search algorithms. A* search was first proposed by Hart et al. [17]. A* was thought to be derived from Dijkstra's algorithm [13] and Moore's algorithm [27]. A* search is similar to the *branch-and-bound* algorithm [23, 39], widely used in *operations research*. The proof of admissibility of A* search can be found in [32].

The application of *beam search* in speech recognition was first introduced by the HARPY system [26]. It wasn't widely popular until BBN used it for their BYBLOS system [37]. There are some excellent papers with detailed description of the use of time-synchronous Viterbi beam search for continuous speech recognition [24, 31]. Over the years, many efficient implementations and improvements have been introduced for time-synchronous Viterbi beam search, so real-time large-vocabulary continuous speech recognition can be realized on a general-purpose personal computer.

On the other hand, stack decoding was first developed by IBM [9]. It is successfully used in IBM's large-vocabulary continuous speech recognition systems [3, 16]. Lacking a time-synchronous framework, comparing theories of different lengths and extending theories are more complex as described in this chapter. Because of the complexity of stack decoding, far fewer publications and systems are based on it than on Viterbi beam search [16, 19, 20, 35]. With the introduction of multistack search [8], stack decoding in essence has actually come very close to time-synchronous Viterbi beam search.

Stack decoding is typically integrated with fast match methods to improve its efficiency. Fast match was first implemented for isolated word recognition to obtain a list of potential word candidates [5, 7]. The paper by Gopalakrishnan et al. [15] contains a comprehensive description of fast match techniques to reduce the word expansion for stack decoding. Besides the fast match techniques described in this chapter, there are a number of alternative approaches [5, 21, 41]. Waast's fast match [41], for example, is based on a binary classification tree built automatically from data that comprise both phonetic transcription and acoustic sequence.

REFERENCES

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

[2] Alleva, F., X. Huang, and M. Hwang, "An Improved Search Algorithm for Continuous Speech Recognition," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN, pp. 307-310.

[3] Bahl, L.R. and et. al., "Large Vocabulary Natural Language Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1989, Glasgow, Scotland, pp. 465-467.

Amazon/VB Assets
Exhibit 1011
Page 666

[4] Bahl, L.R., *et al.*, "Language-Model/Acoustic Channel Balance Mechanism," *IBM Technical Disclosure Bulletin*, 1980, **23**(7B), pp. 3464-3465.

[5] 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.

[6] Bahl, L.R., *et al.*, "A Fast Approximate Acoustic Match for Large Vocabulary Speech Recognition," *IEEE Trans. on Speech and Audio Processing*, 1993(1), pp. 59-67.

[7] 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.

[8] 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.

[9] Bahl, L.R., F. Jelinek, and R. Mercer, "A Maximum Likelihood Approach to Continuous Speech Recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1983(2), pp. 179-190.

[10] Bahl, L.R., *et al.*, "Constructing Candidate Word Lists Using Acoustically Similar Word Groups," *IEEE Trans. on Signal Processing*, 1992(1), pp. 2814-2816.

[11] Barr, A. and E. Feigenbaum, *The Handbook of Artificial Intelligence: Volume I*, 1981, Addison-Wesley.

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

[13] Dijkstra, E.W., "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, 1959, **1**, pp. 269-271.

[14] Gauvain, J.L., *et al.*, "The LIMSI Speech Dictation System: Evaluation on the ARPA Wall Street Journal Corpus," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1994, Adelaide, Australia, pp. 129-132.

[15] Gopalakrishnan, P.S. 2and L.R. Bahl, "Fast Match Techniques," in *Automatic Speech and Speaker Recognition*, C.H. Lee, F.K. Soong, and K.K. Paliwal, eds., 1996, Norwell, MA, Kluwer Academic Publishers, pp. 413-428.

[16] Gopalakrishnan, P.S., L.R. Bahl, and R.L. Mercer, "A Tree Search Strategy for Large-Vocabulary Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, MI, pp. 572-575.

[17] Hart, P.E., N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on Systems Science and Cybernetics*, 1968, **4**(2), pp. 100-107.

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

[19] Jelinek, F., *Statistical Methods for Speech Recognition*, 1998, Cambridge, MA, MIT Press.

Amazon/VB Assets
Exhibit 1011
Page 667

[20] Kenny, P., *et al.*, "A*—Admissible Heuristics for Rapid Lexical Access," *IEEE Trans. on Speech and Audio Processing*, 1993, **1**, pp. 49-58.

[21] Kenny, P., *et al.*, "A New Fast Match for Very Large Vocabulary Continuous Speech Recognition," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN, pp. 656-659.

[22] Laface, P., L. Fissore, and F. Ravera, "Automatic Generation of Words toward Flexible Vocabulary Isolated Word Recognition," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan, pp. 2215-2218.

[23] Lawler, E.W. and D.E. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, 1966(14), pp. 699-719.

[24] Lee, K.F. and F.A. Alleva, "Continuous Speech Recognition" in *Recent Progress in Speech Signal Processing*, S. Furui and M. Sondhi, eds., 1990, Marcel Dekker, Inc.

[25] Lee, K.F., H.W. Hon, and R. Reddy, "An Overview of the SPHINX Speech Recognition System," *IEEE Trans. on Acoustics, Speech and Signal Processing*, 1990, **38**(1), pp. 35-45.

[26] Lowerre, B.T., *The HARPY Speech Recognition System*, PhD Thesis in Computer Science Department, 1976, Carnegie Mellon University.

[27] Moore, E.F., "The Shortest Path Through a Maze," *Int. Symp. on the Theory of Switching*, 1959, Cambridge, MA, Harvard University Press, pp. 285-292.

[28] 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.

[29] Newell, A. and H.A. Simon, *Human Problem Solving*, 1972, Englewood Cliffs, NJ, Prentice Hall.

[30] Ney, H. and X. Aubert, "Dynamic Programming Search: From Digit Strings to Large Vocabulary Word Graphs," in *Automatic Speech and Speaker Recognition*, C.H. Lee, F. Soong and K.K. Paliwal, eds., 1996, Boston, Kluwer Academic Publishers, pp. 385-412.

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

[32] Nilsson, N.J., *Principles of Artificial Intelligence*, 1982, Berlin, Germany, Springer Verlag.

[33] Nilsson, N.J., *Artificial Intelligence: A New Synthesis*, 1998, Academic Press/Morgan Kaufmann.

[34] Normandin, Y., R. Cardin, and R.D. Mori, "High-Performance Connected Digit Recognition Using Maximum Mutual Information Estimation," *IEEE Trans. on Speech and Audio Processing*, 1994, **2**(2), pp. 299-311.

[35] Paul, D.B., "An Efficient A* Stack Decoder Algorithm for Continuous Speech Recognition with a Stochastic Language Model," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1992, San Francisco, California, pp. 25-28.

Amazon/VB Assets
Exhibit 1011
Page 668

[36] Roe, D.B. and M.D. Riley, "Prediction of Word Confusabilities for Speech Recognition," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan, pp. 227-230.

[37] Schwartz, R., *et al.*, "Context-Dependent Modeling for Acoustic-Phonetic Recognition of Speech Signals," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1985, Tampa, FLA, pp. 1205-1208.

[38] 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.

[39] Taha, H.A., *Operations Research: An Introduction*, 6th ed, 1996, Prentice Hall.

[40] Tanimoto, S.L., *The Elements of Artificial Intelligence : An Introduction Using Lisp*, 1987, Computer Science Press, Inc.

[41] Waast, C. and L.R. Bahl, "Fast Match Based on Decision Tree," *Proc. of the European Conf. on Speech Communication and Technology*, 1995, Madrid, Spain, pp. 909-912.

[42] Winston, P.H., *Artificial Intelligence*, 1984, Reading, MA, Addison-Wesley.

[43] Winston, P.H., *Artificial Intelligence*, 3rd ed, 1992, Reading, MA, Addison-Wesley.

[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.

Amazon/VB Assets
Exhibit 1011
Page 669

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

Amazon/VB Assets
Exhibit 1011
Page 671

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 a 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.

[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.

Amazon/VB Assets
Exhibit 1011
Page 672

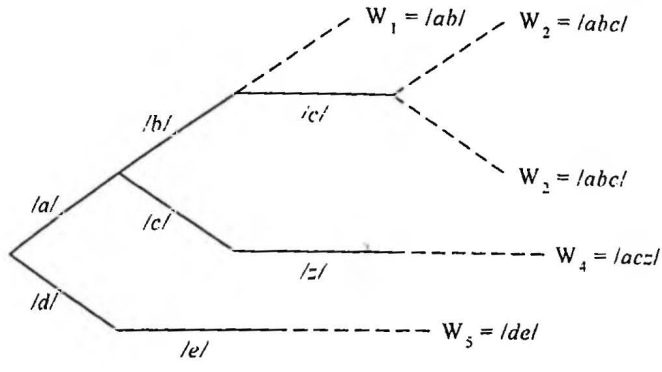

Figure 13.1 An example of a lexical tree, where each branch corresponds to a shared phoneme and the leaf corresponds to a word.

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 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*.

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 a 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 fifth 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

Amazon/VB Assets
Exhibit 1011
Page 673

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 et al. [32] report that the search effort using bigrams is increased by only a factor of 2 over the unigram

Amazon/VB Assets
Exhibit 1011
Page 674

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.

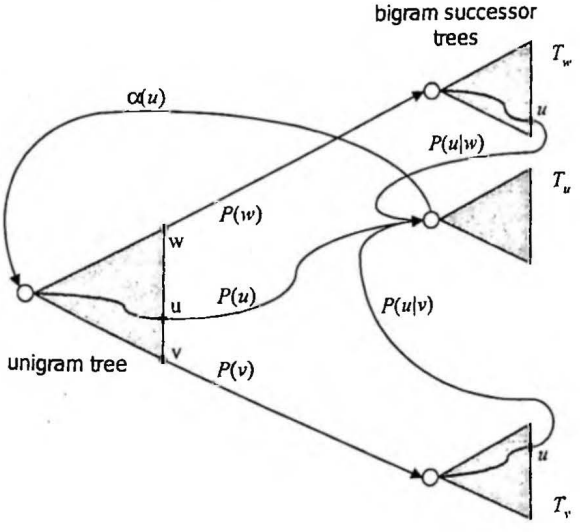

Figure 13.2 Successor trees and unigram trees for bigram search [13].

Amazon/VB Assets
Exhibit 1011
Page 675

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

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

Amazon/VB Assets
Exhibit 1011
Page 676

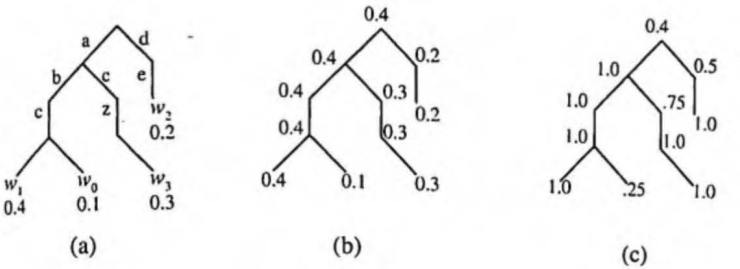

Figure 13.3 (a) Unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree $F^*(n)$ [4].

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.

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

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.

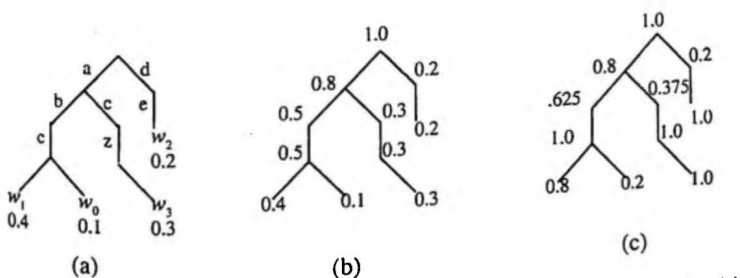

Figure 13.4 Using sum instead of upper bound when factoring tree, the corresponding (a) unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree with computed $F^*(n)$ [4].

Amazon/VB Assets
Exhibit 1011
Page 677

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_l is just $P(w_l \mid w_1^{l-1})$. 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_l. However, for the tree lexicon, it is necessary to compute $E(p_j \mid p_1^{j-1}, w_i^{l-1})$, the expectation of phone p_j given the phonetic prefix p_1^{j-1} and the linguistic context w_1^{l-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:

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

where $c = \arg\max_k(w_k \mid w_1^{l-1}, \phi(j, w_k) = p_1^j)$ and $p = \arg\max_k(w_k \mid w_1^{l-1}, \phi(j-1, w_k) = p_1^{j-1})$. 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.

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.

ALGORITHM 13.1: *ENCODING THE LEXICAL SUCCESSOR TREES (LST)*

For each linguistic context:
Step 1: Distribute the probabilities according to Eq. (13.1).
Step 2: Factor the probabilities according to Eq. (13.2).
Step 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 of the direct ancestor at the depth in (a),
(c) the word identity.

Amazon/VB Assets
Exhibit 1011
Page 678

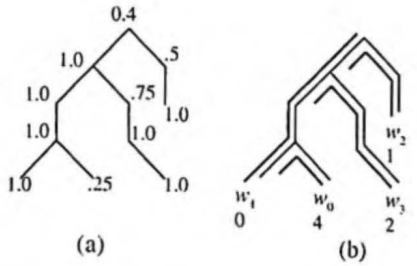

Figure 13.5 (a) Factored tree; (b) tree with common prefix-length annotation.

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 decoding 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.

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.

Amazon/VB Assets
Exhibit 1011
Page 679

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 indistinguishability 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].

Amazon/VB Assets
Exhibit 1011
Page 680

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,[1] 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.

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.

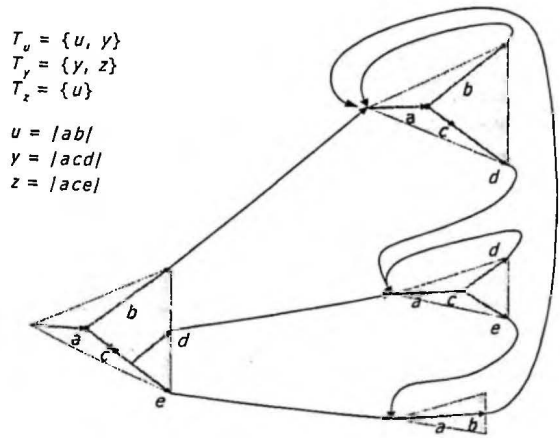

Figure 13.6 An example of a lexical tree network without shared-tail optimization [12]. The vocabulary includes three words, u, y, and z. T_u, T_y, and T_z are the successor trees for u, y, and z respectively [13].

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

Amazon/VB Assets
Exhibit 1011
Page 681

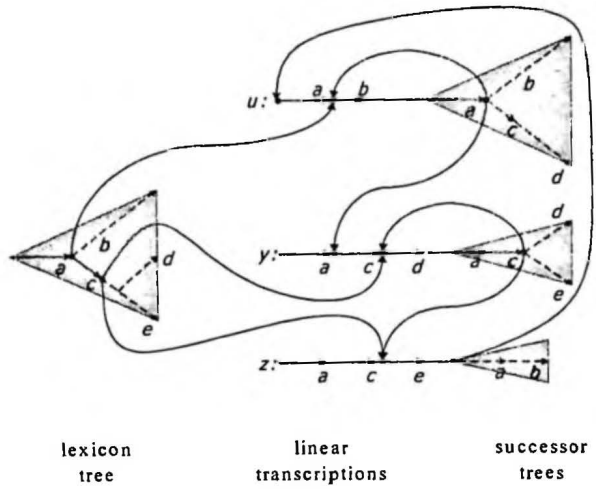

Figure 13.7 The lexical tree network in Figure 13.6 after shared-tail optimization [12].

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

Amazon/VB Assets
Exhibit 1011
Page 682

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.

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, of that of the best global path. 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 transited from 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 cost 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.

ALGORITHM 13.2: *HANDLING MULTIPLE LINGUISTIC CONTEXTS IN A LEXICAL TREE*

1. $d = Cost(s_n, c) + (-\log P(s_m \mid s_n, c))$
2. **if** $InHeap(s_m, c)$ **then**
   if $d < Cost(s_m, c)$ **then**
   $Cost(s_m, c) = d$
   $StateInfo(s_m, c) = StateInfo(s_n, c)$
else if $d < BestCost(s_m) + \varepsilon$ **then**
   $Add(s_m, c)$; $StateInfo(s_m, c) = StateInfo(s_n, c)$
   $Cost(s_m, c) = d$
   else
   $w = WorstContext(s_m)$
   if $d < Cost(s_m, w)$ **then**
   $Delete(s_m, w)$
   $Add(s_m, c)$; $StateInfo(s_m, c) = StateInfo(s_n, c)$
   $Cost(s_m, c) = d$

Amazon/VB Assets
Exhibit 1011
Page 683

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

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

Amazon/VB Assets
Exhibit 1011
Page 684

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 Aglorithm 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 Section 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 is used to determine the promising

Amazon/VB Assets
Exhibit 1011
Page 685

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.

Amazon/VB Assets
Exhibit 1011
Page 686

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 the 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 reduces 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

Amazon/VB Assets
Exhibit 1011
Page 687

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:

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

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 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.

Amazon/VB Assets
Exhibit 1011
Page 688

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.

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

Amazon/VB Assets
Exhibit 1011
Page 689

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 hundreds. 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.

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.

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 Table 13.4. 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:

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

[6] We will use the term *word lattice* in the rest of this chapter..

Amazon/VB Assets
Exhibit 1011
Page 690

- *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.

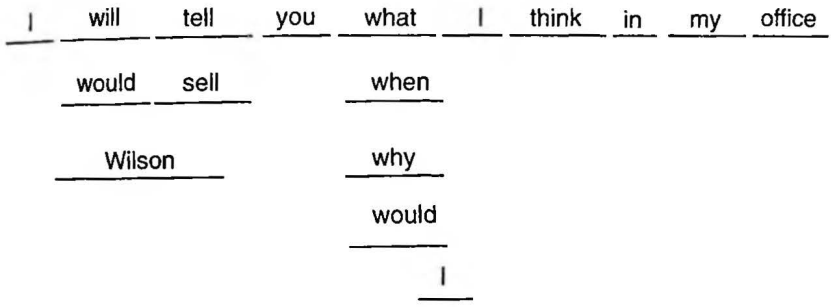

Figure 13.8 A word lattice example. Each word has an explicit time interval associated with it.

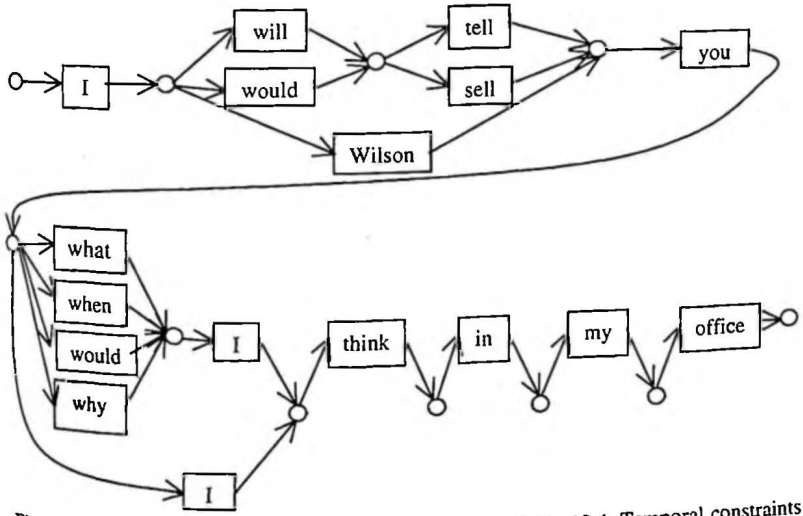

Figure 13.9 A word graph example for the n-best list in Table 13.4. Temporal constraints are implicit in the topology.

Amazon/VB Assets
Exhibit 1011
Page 691

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.

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 generators 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.

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.

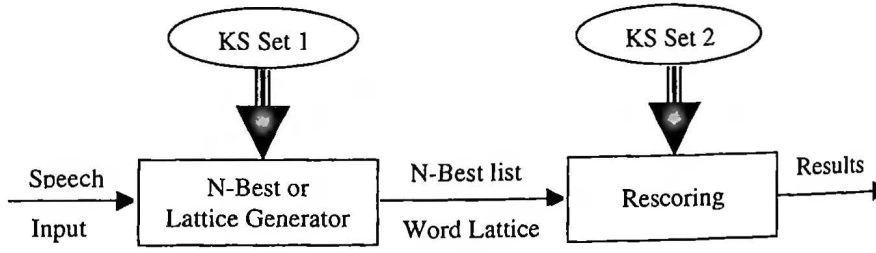

Figure 13.10 N-best/lattice search framework. The most discriminant and inexpensive knowledge sources (KSs 1) are used first to generate the n-best/lattice. The remaining knowledge sources (KSs 2, usually expensive to apply) are used in the rescoring phase to pick up the optimal solution [40].

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

Amazon/VB Assets
Exhibit 1011
Page 692

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.

Amazon/VB Assets
Exhibit 1011
Page 693

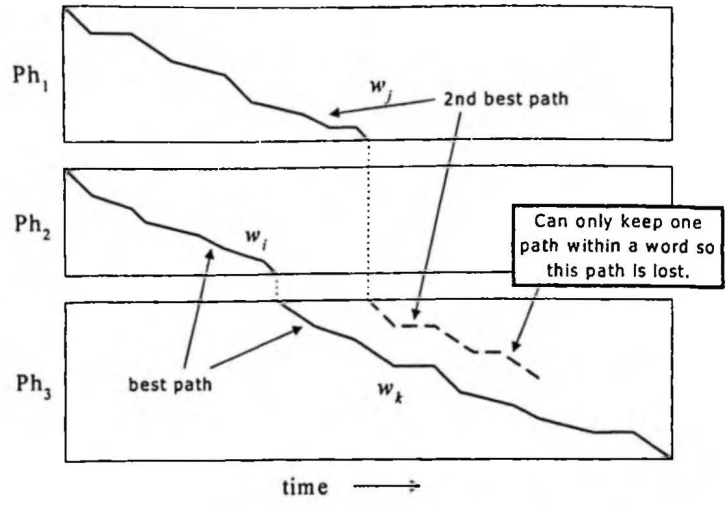

Figure 13.11 Deficiency in traceback-based n-best algorithm. The best subpath, $w_i - w_k$, will prune away subpath $w_j - w_k$ while searching the word w_k; the second-best subpath cannot be recovered [40].

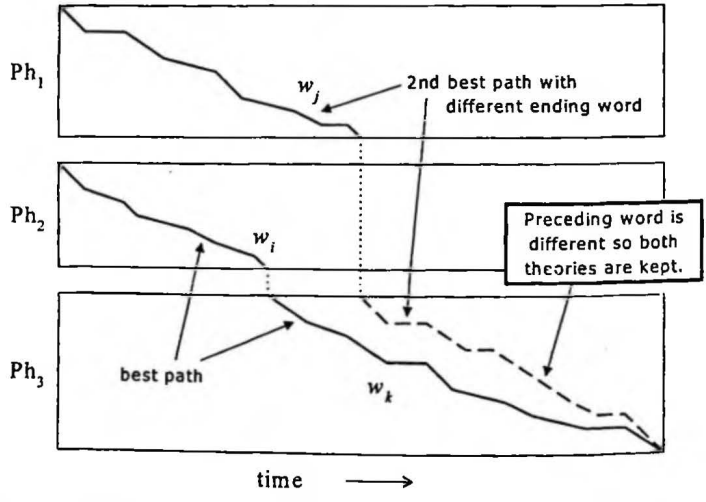

Figure 13.12 Word-dependent n-best algorithm. Both subpaths $w_i - w_k$ and $w_j - w_k$ are kept under the word-dependent assumption [40].

Amazon/VB Assets
Exhibit 1011
Page 694

The *word-dependent* n-best algorithm [38] can alleviate the deficiency of the traceback-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 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 et 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 cost 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 significant. 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.

Amazon/VB Assets
Exhibit 1011
Page 695

At the end of the utterance, the word lattice 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*

Step 1: 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 $h(w_j; \tau(t; w_i, w_j), t)$;
Step 2: Output 1-best result;
Step 3: 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 or word lattice.

Amazon/VB Assets
Exhibit 1011
Page 696

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 guaranteed 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:

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

Amazon/VB Assets
Exhibit 1011
Page 697

Finally, the new path ph', including the one-word (w_i) extension, is inserted into the stack, ordered by the cost score $\alpha_t(w_i)+\beta_t(ph_{w_i})$. The heuristic function (forward scores α) allows the backward A* search to concentrate search on extending only a 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

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

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.

Amazon/VB Assets
Exhibit 1011
Page 698

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 a 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

Amazon/VB Assets
Exhibit 1011
Page 699

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.

Amazon/VB Assets
Exhibit 1011
Page 700

- 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

Amazon/VB Assets
Exhibit 1011
Page 701

threshold to retain the correct path. For example, one can adjust the pruning threshold for fast match if a word in $\tilde{W}$ 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 the 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

Amazon/VB Assets
Exhibit 1011
Page 702

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.

[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.

Amazon/VB Assets
Exhibit 1011
Page 703

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

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.

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

Amazon/VB Assets
Exhibit 1011
Page 704

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

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 created 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.

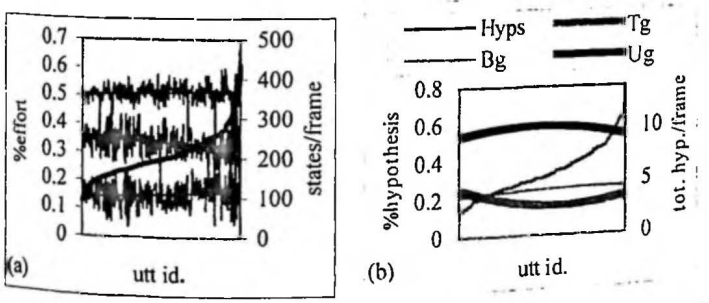

Figure 13.13 (a) Search effort for different linguistic contexts measured by number of active states in each of the three different linguistic contexts. The top series is for the bigram, then the unigram and trigram. The remaining series is the effort per utterance and is plotted on the secondary y-axis. (b) The distribution of word hypotheses 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].

Amazon/VB Assets
Exhibit 1011
Page 705

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.

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.

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

Amazon/VB Assets
Exhibit 1011
Page 706

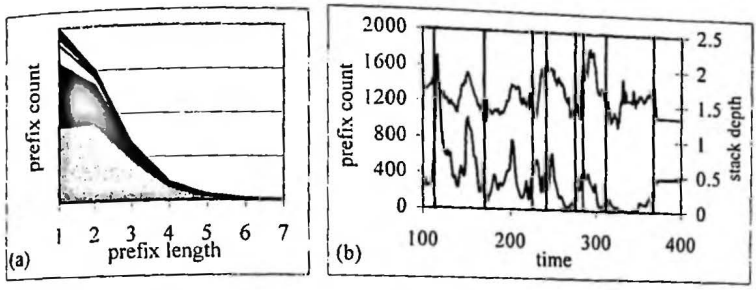

Figure 13.14 (a) A cumulative graph of the prefix count for each stack depth, starting with depth 1, showing the distribution according to prefix length. (b) The prefix count and the average stack depth with respect to one utterance. The vertical bars show the word boundaries [3].

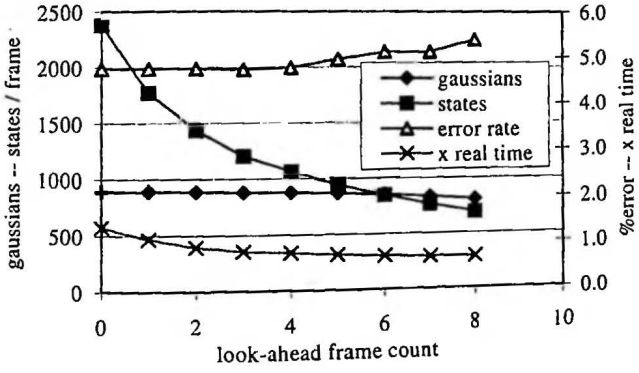

Figure 13.15 Search effort, percent error rate, and real-time factor as a function of the acoustic look-ahead. Note that search effort is the number of Gaussians evaluated per frame and the number of states evaluated per frame [3].

13.6. Historical Perspective 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 the 1990s [4, 34, 43].

Common wisdom in 1980s 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 frame-

Amazon/VB Assets
Exhibit 1011
Page 707

work. 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 and Ortmanns [33] and Alleva [3] 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 times 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.

Amazon/VB Assets
Exhibit 1011
Page 708

[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, ed., Academic Press, 1998, London, pp. 231-256.

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

[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.

Amazon/VB Assets
Exhibit 1011
Page 709

[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, Kluwer Academic Publishers, pp. 481-508.

[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.

Amazon/VB Assets
Exhibit 1011
Page 710

[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, NJ, 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, Klewer Academic Publishers, pp. 57-81.

[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," *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.

Amazon/VB Assets
Exhibit 1011
Page 711

PART IV

TEXT-TO-SPEECH SYSTEMS

Amazon/VB Assets
Exhibit 1011
Page 713

CHAPTER 14

Text and Phonetic Analysis

Text-to-speech can be viewed as a speech coding system that yields an extremely high compression ratio coupled with a high degree of flexibility in choosing style, voice, rate, pitch range, and other playback effects. In this view of TTS, the text file that is input to a speech synthesizer is a form of coded speech. Thus, TTS subsumes coding technologies discussed in Chapter 7 with the following goals:

- *Compression ratios superior to digitized wave files*—Compression yields benefits in many areas, including fast Internet transmission of spoken messages.
- *Flexibility in output characteristics*—Flexibility includes easy change of gender, average pitch, pitch range, etc., enabling application developers to give their systems' spoken output a unique individual personality. Flexibility also implies easy change of message content; it is generally easier to retype text than it is to record and deploy a digitized speech file.
- *Ability for perfect indexing between text and speech forms*—Preservation of the correspondence between textual representation and the speech wave form allows synchronization with other media and output modes, such as word-by-word reverse video highlighting in a literacy tutor reading aloud.

Amazon/VB Assets
Exhibit 1011
Page 715

- *Alternative access of text content*—TTS is the most effective alternative access of text for the blind, hands-free/eyes-free and displayless scenarios.

At first sight, the process of converting text into speech looks straightforward. However, when we analyze how complicated speakers read a text aloud, this simplistic view quickly falls apart. First, we need to convert words in written forms into speakable forms. This process is clearly nontrivial. Second, to sound natural, the system needs to convey the intonation of the sentences properly. This second process is clearly an extremely challenging one. One good analogy is to think how difficult it is to drop a foreign accent when speaking a second language—a process still not quite understood by human beings.

The ultimate goal of simulating the speech of an understanding, effective human speaker from plain text is as distant today as the corresponding Holy Grail goals of the fields of speech recognition and machine translation. This is because such humanlike rendition depends on common-sense reasoning about the world and the text's relation to it, deep knowledge of the language itself in all its richness and variability, and even knowledge of the actual or expected audience—its goals, assumptions, presuppositions, and so on. In typical audio books or recordings for the visually challenged today, the human reader has enough familiarity with and understanding of the text to make appropriate choices for rendition of emotion, emphasis, and pacing, as well as handling both dialog and exposition. While computational power is steadily increasing, there remains a substantial knowledge gap that must be closed before fully human-sounding simulated voices and renditions can be created.

While no TTS system to date has approached optimal quality in the Turing test,[1] a large number of experimental and commercial systems have yielded fascinating insights. Even the relatively limited-quality TTS systems of today have found practical applications.

The basic TTS system architecture is illustrated in Chapter 1. In the present chapter we discuss text analysis and phonetic analysis whose objective is to convert words into speakable phonetic representation. The techniques discussed here are relevant to what we discussed for language modeling in Chapter 11 (like text normalization before computing *n*-gram) and for pronunciation modeling in Chapter 9. The next two modules—prosodic analysis and speech synthesis—are treated in the next two chapters.

14.1. MODULES AND DATA FLOW

The text analysis component, guided by presenter controls, is typically responsible for determining document structure, conversion of nonorthographic symbols, and parsing of language structure and meaning. The phonetic analysis component converts orthographic words to phones (unambiguous speech sound symbols). Some TTS systems assume dependency between text analysis, phonetic analysis, prosodic analysis, and speech synthesis, particularly systems based on very large databases containing long stretches of original, unmodified

[1] A test proposed by British mathematician Allan Turing of the ability of a computer to flawlessly imitate human performance on a given speech or language task [29].

Amazon/VB Assets
Exhibit 1011
Page 716

digitized speech with their original pitch contours. We discuss our high-level linguistic description of those modules, based on modularity, transparency, and reusability of components, although some aspects of text and phonetic analysis may be unnecessary for some particular systems.

We assume that the entire text (word, sentence, paragraph, document) to be spoken is contained in a single, wholly visible buffer. Some systems may be faced with special requirements for continuous flow-through or visibility of only small (word, phrase, sentence) *chunks* at a time, or extremely complex timing and synchronization requirements. The basic functional processes within the text and phonetic analysis are shown schematically in Figure 14.1.

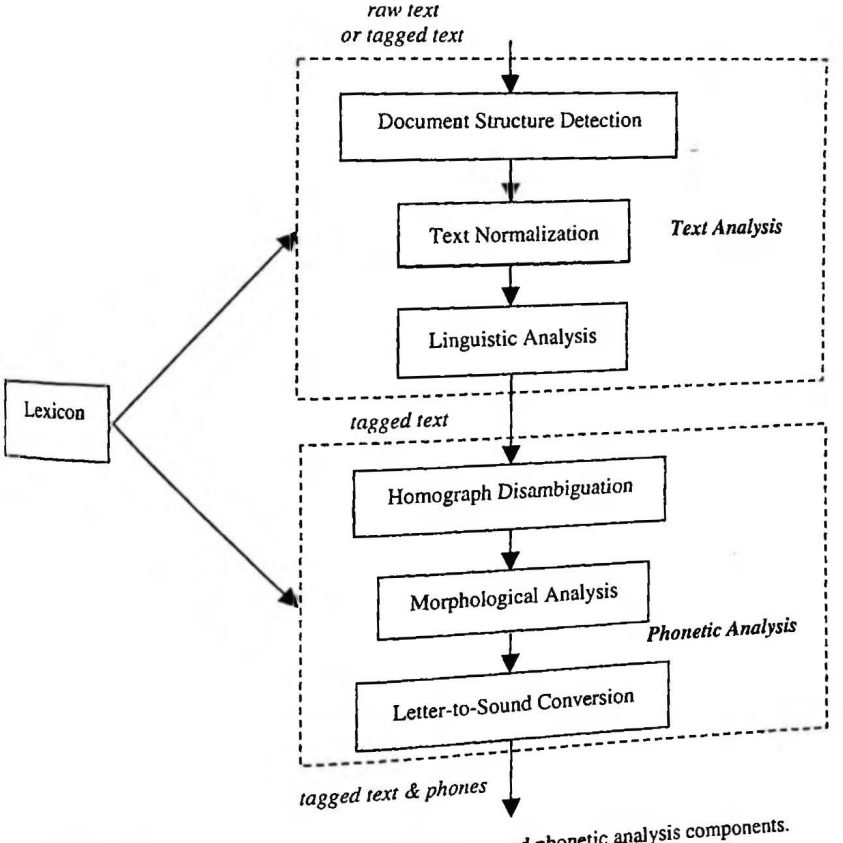

Figure 14.1 Modularized functional blocks for text and phonetic analysis components.

Amazon/VB Assets
Exhibit 1011
Page 717

The architecture in Figure 14.1 brings the standard benefits of modularity and transparency. Modularity in this case means that the analysis at each level can be supplied by the most expert knowledge source, or a variety of different sources, as long as the markup conventions for expressing the analysis are uniform. Transparency means that the results of each stage could be reused by other processes for other purposes.

14.1.1. Modules

The *text analysis module* (TAM) is responsible for indicating all knowledge about the text or message that is not specifically phonetic or prosodic in nature. Very simple systems do little more than convert nonorthographic items, such as numbers, into words. More ambitious systems attempt to analyze whitespaces and punctuations to determine document structure, and perform sophisticated syntax and semantic analysis on sentences to determine attributes that help the phonetic analysis to generate correct phonetic representation and prosodic generation to construct superior pitch contours. As shown in Figure 14.1, text analysis for TTS involves three related processes:

- *Document structure detection*—Document structure is important to provide a context for all later processes. In addition, some elements of document structure, such as sentence breaking and paragraph segmentation, may have direct implications for prosody.
- *Text normalization*—Text normalization is the conversion from the variety of symbols, numbers, and other nonorthographic entities of text into a common orthographic transcription suitable for subsequent phonetic conversion.
- *Linguistic analysis*—Linguistic analysis recovers the syntactic constituency and semantic features of words, phrases, clauses, and sentences, which is important for both pronunciation and prosodic choices in the successive processes.

The task of the phonetic analysis is to convert lexical orthographic symbols to phonemic representation along with possible diacritic information, such as stress placement. Phonetic analysis is thus often referred to as grapheme-to-phoneme conversion. The purpose is obvious, since phonemes are the basic units of sound, as described in Chapter 2. Even though future TTS systems might be based on word sounding units with increasing storage technologies, homograph disambiguation and phonetic analysis for new words (either true new words being invented over time or morphologically transformed words) are still necessary for systems to correctly utter every word.

Grapheme-to-phoneme conversion is trivial for languages where there is a simple relationship between orthography and phonology. Such a simple relationship can be well captured by a handful of rules. Languages such as Spanish and Finnish belong to this category and are referred to as *phonetic languages*. English, on the other hand, is remote from pho-

Amazon/VB Assets
Exhibit 1011
Page 718