[102] Sproat, R. and M. Riley, "Compilation of Weighted Finite-State Transducers from Decision Trees," *ACL-96*, 1996, Santa Cruz, pp. 215-222.

[103] Tajchman, G., E. Fosler, and D. Jurafsky, "Building Multiple Pronunciation Models for Novel Words Using Exploratory Computational Phonology," *Eurospeech*, 1995, pp. 2247-2250.

[104] Tebelskis, J. and A. Waibel, "Large Vocabulary Recognition Using Linked Predictive Neural Networks," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1990, Albuquerque, NM, pp. 437-440.

[105] Waibel, A.H. and K.F. Lee, *Readings in Speech Recognition*, 1990, San Mateo, CA, Morgan Kaufman Publishers.

[106] Watrous, R., "Speaker Normalization and Adaptation Using Second-Order Connectionist Networks," *IEEE Trans. on Neural Networks*, 1994, 4(1), pp. 21-30.

[107] Welling, L., S. Kanthak, and H. Ney, "Improved Methods for Vocal Tract Normalization," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1999, Phoenix, AZ.

[108] Wilpon, J.G., C.H. Lee, and L.R. Rabiner, "Connected Digit Recognition Based on Improved Acoustic Resolution," *Computer Speech and Language*, 1993, 7(1), pp. 15-26.

[109] Woodland, P.C., *et al.*, "The 1994 HTK Large Vocabulary Speech Recognition System," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, pp. 73-76.

[110] Wooters, C. and A. Stolcke, "Multiple-Pronunciation Lexical Modeling in a Speaker Independent Speech Understanding System," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan, pp. 1363-1366.

[111] Young, S.J. and P.C. Woodland, "The Use of State Tying in Continuous Speech Recognition," *Proc. of Eurospeech*, 1993, Berlin pp. 2203-2206.

[112] Zavaliagkos, G., R. Schwartz, and J. Makhoul, "Batch, Incremental and Instantaneous Adaptation Techniques for Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, pp. 676-679.

[113] Zavaliagkos, G., *et al.*, "A Hybrid Segmental Neural Net/Hidden Markov Model System for Continuous Speech Recognition," *IEEE Trans. on Speech and Audio Processing*, 1994, **2**, pp. 151-160.

[114] Zhan, P. and M. Westphal, "Speaker Normalization Based on Frequency Warping" in *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing* 1997, Munich, Germany, pp. 1039-1042.

[115] Zue, V., *et al.*, "The MIT SUMMIT System: A Progress Report," *Proc. of DARPA Speech and Natural Language Workshop*, 1989, pp. 179-189.

CHAPTER 10

Environmental Robustness

A speech recognition system trained in the lab with clean speech may degrade significantly in the real world if the clean speech used in training doesn't match real-world speech. If its accuracy doesn't degrade very much under mismatched conditions, the system is called *robust*. There are several reasons why real-world speech may differ from clean speech; in this chapter we focus on the influence of the *acoustical environment*, defined as the transformations that affect the speech signal from the time it leaves the mouth until it is in digital format.

Chapter 9 discussed a number of variability factors that are critical to speech recognition. Because the acoustical environment is so important to practical systems, we devote this chapter to ways of increasing the environmental robustness, including microphone, echo cancellation, and a number of methods that enhance the speech signal, its spectrum, and the corresponding acoustic model in a speech recognition system.

10.1. THE ACOUSTICAL ENVIRONMENT

The acoustical environment is defined as the set of transformations that affect the speech signal from the time it leaves the speaker's mouth until it is in digital form. Two main sources of distortion are described here: additive noise and channel distortion. Additive noise, such as a fan running in the background, door slams, or other speakers' speech, is common in our daily life. Channel distortion can be caused by reverberation, the frequency response of a microphone, the presence of an electrical filter in the A/D circuitry, the response of the local loop of a telephone line, a speech codec, etc. Reverberation, caused by reflections of the acoustical wave in walls and other objects, can also dramatically alter the speech signal.

10.1.1. Additive Noise

Additive noise can be stationary or nonstationary. Stationary noise, such as that made by a computer fan or air conditioning, has a power spectral density that does not change over time. Nonstationary noise, caused by door slams, radio, TV, and other speakers' voices, has statistical properties that change over time. A signal captured with a close-talking microphone has little noise and reverberation, even though there may be lip smacks and breathing noise. A microphone that is not close to the speaker's mouth may pick up a lot of noise and/or reverberation.

As described in Chapter 5, a signal $x[n]$ is defined as white noise if its power spectrum is flat, $S_{xx}(f) = q$, a condition equivalent to different samples being uncorrelated, $R_{xx}[n] = q\delta[n]$. Thus, a white noise signal has to have zero mean. This definition tells us about the second-order moments of the random process, but not about its distribution. Such noise can be generated synthetically by drawing samples from a distribution $p(x)$; thus we could have uniform white noise if $p(x)$ is uniform, or Gaussian white noise if $p(x)$ is Gaussian. While typically subroutines are available that generate uniform white noise, we are often interested in white Gaussian noise, as it resembles better the noise that tends to occur in practice. See Algorithm 10.1 for a method to generate white Gaussian noise. Variable x is normally continuous, but it can also be discrete.

White noise is useful as a conceptual entity, but it seldom occurs in practice. Most of the noise captured by a microphone is *colored*, since its spectrum is not flat. *Pink* noise is a particular type of colored noise that has a low-pass nature, as it has more energy at the low frequencies and rolls off at higher frequencies. The noise generated by a computer fan, an air conditioner, or an automobile engine can be approximated by pink noise. We can synthesize pink noise by filtering white noise with a filter whose magnitude squared equals the desired power spectrum.

A great deal of additive noise is nonstationary, since its statistical properties change over time. In practice, even the noises from a computer, an air conditioning system, or an automobile are not perfectly stationary. Some nonstationary noises, such as keyboard clicks, are caused by physical objects. The speaker can also cause nonstationary noises such as lip

smacks and breath noise. The *cocktail party effect* is the phenomenon under which a human listener can focus onto one conversation out of many in a cocktail party. The noise of the conversations that are not focused upon is called *babble* noise. When the nonstationary noise is correlated with a known signal, the adaptive echo-canceling (AEC) techniques of Section 10.3 can be used.

ALGORITHM 10.1: *WHITE NOISE GENERATION*

To generate white noise in a computer, we can first generate a random variable ρ with a Rayleigh distribution:

$$p_\rho(\rho) = \rho e^{-\rho^2/2} \tag{10.1}$$

from another random variable r with a uniform distribution between (0, 1), $p_r(r) = 1$, by simply equating the probability mass $p_\rho(\rho)|d\rho| = p_r(r)|dr|$ so that $\left|\frac{dr}{d\rho}\right| = \rho e^{-\rho^2/2}$; with integration, it results in $r = e^{-\rho^2/2}$ and the inverse is given by

$$\rho = \sqrt{-2\ln r} \tag{10.2}$$

If r is uniform between (0, 1), and ρ is computed through Eq. (10.2), it follows a Rayleigh distribution as in Eq. (10.1). We can then generate Rayleigh white noise by drawing independent samples from such a distribution.

If we want to generate white Gaussian noise, the method used above does not work, because the integral of the Gaussian distribution does not exist in closed form. However, if ρ follows a Rayleigh distribution as in Eq. (10.1), obtained using Eq. (10.2) where r is uniform between (0, 1), and θ is uniformly distributed between (0, 2π), then the white Gaussian noise can be generated as the following two variables x and y:

$$\begin{aligned} x &= \rho\cos(\theta) \\ y &= \rho\sin(\theta) \end{aligned} \tag{10.3}$$

They are independent Gaussian random variables with zero mean and unity variance, since the Jacobian of the transformation is given by

$$J = \begin{vmatrix} \frac{\partial p_x}{\partial \rho} & \frac{\partial p_x}{\partial \theta} \\ \frac{\partial p_y}{\partial \rho} & \frac{\partial p_y}{\partial \theta} \end{vmatrix} = \begin{vmatrix} \cos\theta & -\rho\sin\theta \\ \sin\theta & \rho\cos\theta \end{vmatrix} = \rho \tag{10.4}$$

and the joint density $p(x, y)$ is given by

$$\begin{aligned} p(x,y) &= \frac{p(\rho,\theta)}{J} = \frac{p(\rho)p(\theta)}{\rho} = \frac{1}{2\pi} e^{-\rho^2/2} \\ &= \frac{1}{2\pi} e^{-(x^2+y^2)/2} = N(x,0,1)N(y,0,1) \end{aligned} \tag{10.5}$$

The presence of additive noise can sometimes change the way the speaker speaks. The *Lombard effect* [40] is a phenomenon by which a speaker increases his vocal effort in the presence of background noise. When a large amount of noise is present, the speaker tends to shout, which entails not only a higher amplitude, but also often higher pitch, slightly different formants, and a different coloring of the spectrum. It is very difficult to characterize these transformations analytically, but recently some progress has been made [36].

10.1.2. Reverberation

If both the microphone and the speaker are in an *anechoic*[1] chamber or in free space, a microphone picks up only the direct acoustic path. In practice, in addition to the direct acoustic path, there are reflections of walls and other objects in the room. We are well aware of this effect when we are in a large room, which can prevent us from understanding if the reverberation time is too long. Speech recognition systems are much less robust than humans and they start to degrade with shorter reverberation times, such as those present in a normal office environment.

As described in Chapter 2, the signal level at the microphone is inversely proportional to the distance r from the speaker for the direct path. For the kth reflected sound wave, the sound has to travel a larger distance r_k, so that its level is proportionally lower. This reflection also takes time $T_k = r_k / c$ to arrive, where c is the speed of sound in air.[2] Moreover, some energy absorption a takes place each time the sound wave hits a surface. The impulse response of such filter looks like

$$h[n] = \sum_{k=0}^{\infty} \frac{\rho_k}{r_k} \delta[n - T_k] = \frac{1}{c} \sum_{k=0}^{\infty} \frac{\rho_k}{T_k} \delta[n - T_k] \tag{10.6}$$

where ρ_k is the combined attenuation of the kth reflected sound wave due to absorption. Anechoic rooms have $\rho_k \approx 0$. In general ρ_k is a (generally decreasing) function of frequency, so that instead of impulses $\delta[n]$ in Eq. (10.6), other (low-pass) impulse responses are used.

Often we have available a large amount of speech data recorded with a close-talking microphone, and we would like to use the speech recognition system with a far field microphone. To do that we can filter the clean-speech training database with a filter $h[n]$, so that the filtered speech resembles speech collected with the far field microphone, and then retrain the system. This requires estimating the impulse response $h[n]$ of a room. Alternatively, we can filter the signal from the far field microphone with an inverse filter to make it resemble the signal from the close-talking microphone.

---

[1] An anechoic chamber is a room that has walls made of special fiberglass or other sound-absorbing materials so that it absorbs all echoes. It is equivalent to being in free space, where there are neither walls nor reflecting surfaces.

[2] In air at standard atmospheric pressure and humidity the speed of sound is $c = 331.4 + 0.6T \ (m/s)$. It varies with different media and different levels of humidity and pressure.

One way to estimate the impulse response is to play a white noise signal $x[n]$ through a loudspeaker or artificial mouth; the signal $y[n]$ captured at the microphone is given by

$$y[n] = x[n] * h[n] + v[n] \tag{10.7}$$

where $v[n]$ is the additive noise present at the microphone. This noise is due to sources such as air conditioning and computer fans and is an obstacle to measuring $h[n]$. The impulse response can be estimated by minimizing the error over N samples

$$E = \frac{1}{N}\sum_{n=0}^{N-1}\left(y[n] - \sum_{m=0}^{M-1} h[m]x[n-m] \right)^2 \tag{10.8}$$

which, taking the derivative with respect to $h[m]$ and equating to 0, results in our estimate $\hat{h}[l]$:

$$\begin{aligned}
\left.\frac{\partial E}{\partial h[l]}\right|_{h[l]=\hat{h}[l]} &= \frac{1}{N}\sum_{n=0}^{N-1}\left(y[n] - \sum_{m=0}^{M-1} \hat{h}[m]x[n-m] \right)x[n-l] \\
&= \frac{1}{N}\sum_{n=0}^{N-1} y[n]x[n-l] - \sum_{m=0}^{M-1}\hat{h}[m]\left(\frac{1}{N}\sum_{n=0}^{N-1} x[n-m]x[n-l] \right) \\
&= \frac{1}{N}\sum_{n=0}^{N-1} y[n]x[n-l] - \hat{h}[l] - \sum_{m=0}^{M-1}\hat{h}[m]\left(\frac{1}{N}\sum_{n=0}^{N-1} x[n-m]x[n-l] - \delta[m-l] \right) = 0
\end{aligned} \tag{10.9}$$

Since we know our white process is ergodic, it follows that we can replace time averages by ensemble averages as $N \to \infty$:

$$\lim_{N\to\infty}\frac{1}{N}\sum_{n=0}^{N-1} x[n-m]x[n-l] = E\{x[n-m]x[n-l]\} = \delta[m-l] \tag{10.10}$$

so that we can obtain a reasonable estimate of the impulse response as

$$\hat{h}[l] = \frac{1}{N}\sum_{n=0}^{N-1} y[n]x[n-l] \tag{10.11}$$

Inserting Eq. (10.7) into Eq. (10.11) ,we obtain

$$\hat{h}[l] = h[l] + e[l] \tag{10.12}$$

where the estimation error $e[n]$ is given by

$$e[l] = \frac{1}{N}\sum_{n=0}^{N-1} v[n]x[n-l] + \sum_{m=0}^{M-1} h[m]\left(\frac{1}{N}\sum_{n=0}^{N-1} x[n-m]x[n-l] - \delta[m-l] \right) \tag{10.13}$$

If $v[n]$ and $x[n]$ are independent processes, then $E\{e[l]\} = 0$, since $x[n]$ is zero-mean, so that the estimate of Eq. (10.11) is unbiased. The covariance matrix decreases to 0 as

$N \to \infty$, with the dominant term being the noise $v[n]$. The choice of N for a low-variance estimate depends on the filter length M and the noise level present in the room.

The filter $h[n]$ could also be estimated by playing sine waves of different frequencies or a chirp[3] [52]. Since playing a white noise signal or sine waves may not be practical, another method is based on collecting stereo recordings with a close-talking microphone and a far field microphone. The filter $h[n]$ of length M is estimated so that when applied to the close-talking signal $x[n]$ it minimizes the squared error with the far field signal $y[n]$, which results in the following set of M linear equations:

$$\sum_{m=0}^{M-1} h[m]R_{xx}[m-n] = R_{xy}[n] \tag{10.14}$$

which is a generalization of Eq. (10.11) when $x[n]$ is not a white noise signal.

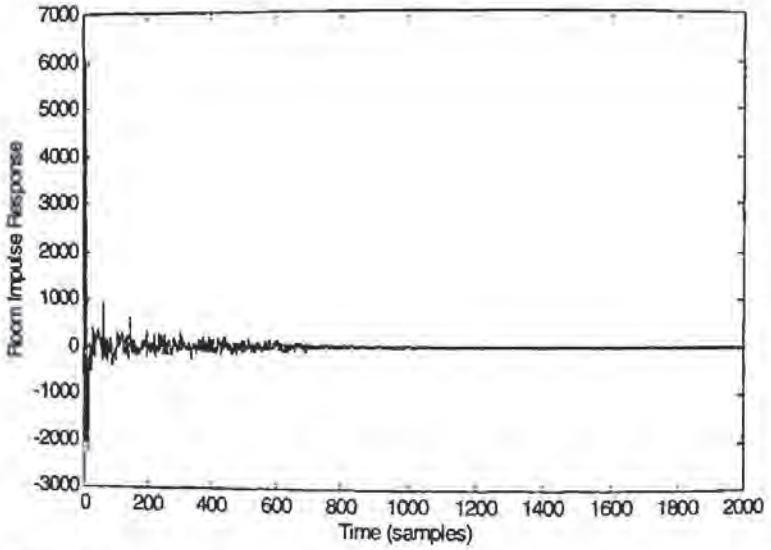


Figure 10.1 Typical impulse response of an average office. Sampling rate was 16 kHz. It was estimated by driving a 4-minute segment of white noise through an artificial mouth and using Eq. (10.11). The filter length is about 125 ms.

It is not uncommon to have reverberation times of over 100 milliseconds in office rooms. In Figure 10.1 we show the typical impulse response of an average office.

10.1.3. A Model of the Environment

A widely used model of the degradation encountered by the speech signal when it gets corrupted by both additive noise and channel distortion is shown in Figure 10.2. We can derive

[3] A chirp function continuously varies its frequency. For example, a linear chirp varies its frequency linearly with time: $\sin(n(\omega_0 + \omega_1 n))$.

the relationships between the clean signal and the corrupted signal both in power-spectrum and cepstrum domains based on such a model [2].

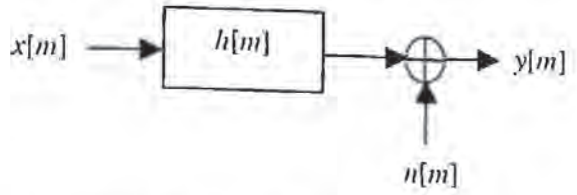


Figure 10.2 A model of the environment.

In the time domain, additive noise and linear filtering results in

$$y[m] = x[m] * h[m] + n[m] \tag{10.15}$$

It is convenient to express this in the frequency domain using the short-time analysis methods of Chapter 6. To do that, we window the signal, take a $2K$-point DFT in Eq. (10.15) and then the magnitude squared:

$$\begin{aligned} |Y(f_k)|^2 &= |X(f_k)|^2 |H(f_k)|^2 + |N(f_k)|^2 + 2\,\mathrm{Re}\{X(f_k)H(f_k)N^*(f_k)\} \\ &= |X(f_k)|^2 |H(f_k)|^2 + |N(f_k)|^2 + 2|X(f_k)||H(f_k)||N(f_k)|\cos(\theta_k) \end{aligned} \tag{10.16}$$

where $k = 0, 1, \cdots, K$, we have used upper case for frequency domain linear spectra, and θ_k is the angle between the filtered signal and the noise for bin k.

The expected value of the *cross-term* in Eq. (10.16) is zero, since $x[m]$ and $n[m]$ are statistically independent. In practice, this term is not zero for a given frame, though it is small if we average over a range of frequencies, as we often do when computing the popular mel-cepstrum (see Chapter 6). When using a filterbank, we can obtain a relationship for the energies at each of the M filters:

$$|Y(f_i)|^2 \approx |X(f_i)|^2 |H(f_i)|^2 + |N(f_i)|^2 \tag{10.17}$$

where it has been shown experimentally that this assumption works well in practice.

Equation (10.17) is also implicitly assuming that the length of $h[n]$, the filter's impulse response, is much shorter than the window length $2N$. That means that for filters with long reverberation times, Eq. (10.17) is inaccurate. For example, for $|N(f)|^2 = 0$, a window shift of T, and a filter's impulse response $h[n] = \delta[n-T]$, we have $Y_t[f_m] = X_{t-1}[f_m]$, i.e., the output spectrum at frame t does not depend on the input spectrum at that frame. This is a more serious assumption, which is why speech recognition systems tend to fail under long reverberation times.

By taking logarithms in Eq. (10.17), and after some algebraic manipulation, we obtain

$$\ln|Y(f_i)|^2 \approx \ln|X(f_i)|^2 + \ln|H(f_i)|^2 + \ln\left(1+\exp\left(\ln|N(f_i)|^2 - \ln|X(f_i)|^2 - \ln|H(f_i)|^2\right)\right) \tag{10.18}$$

Since most speech recognition systems use cepstrum features, it is useful to see the effect of the additive noise and channel distortion directly on the cepstrum. To do that, let's define the following length-$(M+1)$ cepstrum vectors:

$$\begin{aligned}
\mathbf{x} &= \mathbf{C}\left(\ln|X(f_0)|^2 \quad \ln|X(f_1)|^2 \quad \cdots \quad \ln|X(f_M)|^2\right)\\
\mathbf{h} &= \mathbf{C}\left(\ln|H(f_0)|^2 \quad \ln|H(f_1)|^2 \quad \cdots \quad \ln|H(f_M)|^2\right)\\
\mathbf{n} &= \mathbf{C}\left(\ln|N(f_0)|^2 \quad \ln|N(f_1)|^2 \quad \cdots \quad \ln|N(f_M)|^2\right)\\
\mathbf{y} &= \mathbf{C}\left(\ln|Y(f_0)|^2 \quad \ln|Y(f_1)|^2 \quad \cdots \quad \ln|Y(f_M)|^2\right)
\end{aligned} \tag{10.19}$$

where $\mathbf{C}$ is the DCT matrix and we have used lower-case bold to represent cepstrum vectors.

Combining Eqs. (10.18) and (10.19) results in

$$\mathbf{y} = \mathbf{x} + \mathbf{h} + \mathbf{g}(\mathbf{n} - \mathbf{x} - \mathbf{h}) \tag{10.20}$$

where the nonlinear function $\mathbf{g}(\mathbf{z})$ is given by

$$\mathbf{g}(\mathbf{z}) = \mathbf{C}\ln\left(1+e^{\mathbf{C}^{-1}\mathbf{z}}\right) \tag{10.21}$$

Equations (10.20) and (10.21) say that we can compute the cepstrum of the corrupted speech if we know the cepstrum of the clean speech, the cepstrum of the noise, and the cepstrum of the filter. In practice, the DCT matrix $\mathbf{C}$ is not square, so that the dimension of the cepstrum vector is much smaller than the number of filters. This means that we are losing resolution when going back to the frequency domain, and thus Eqs. (10.20) and (10.21) represent only an approximation, though it has been shown to work reasonably well.

As discussed in Chapter 9, the distribution of the cepstrum of $\mathbf{x}$ can be modeled as a mixture of Gaussian densities. Even if we assume that $\mathbf{x}$ follows a Gaussian distribution, $\mathbf{y}$ in Eq. (10.20) is no longer Gaussian because of the nonlinearity in Eq. (10.21).

It is difficult to visualize the effect on the distribution, given the nonlinearity involved. To provide some insight, let's consider the frequency-domain version of Eq. (10.18) when no filtering is done, i.e., $H(f)=1$:

$$y = x + \ln\left(1+\exp(n-x)\right) \tag{10.22}$$

where x, n, and y represent the log-spectral energies of the clean signal, noise, and noisy signal, respectively, for a given frequency. Using simulated data, not real speech, we can

analyze the effect of this transformation. Let's assume that both x and n are Gaussian random variables. We can use Monte Carlo simulation to draw a large number of points from those two Gaussian distributions and obtain the corresponding noisy values y using Eq. (10.22). Figure 10.3 shows the resulting distribution for several values of σ_x. We fixed $\mu_n = 0\,\text{dB}$, since it is only a relative level, and set $\sigma_n = 2\,\text{dB}$, a typical value. We also set $\mu_x = 25\,\text{dB}$ and see that the resulting distribution can be bimodal when σ_x is very large. Fortunately, for modern speech recognition systems that have many Gaussian components, σ_x is never that large and the resulting distribution is unimodal.

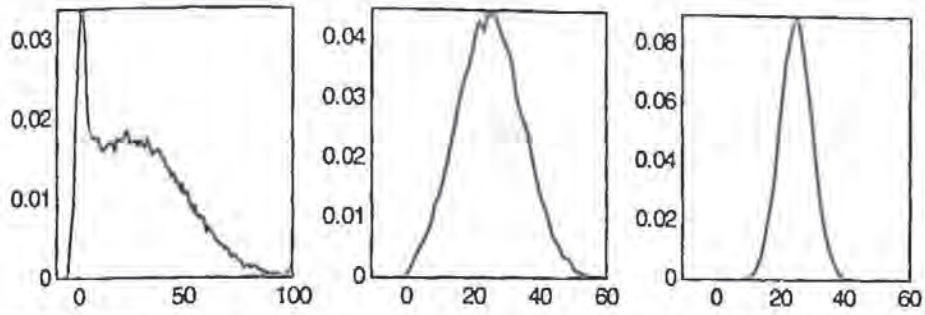


Figure 10.3 Distributions of the corrupted log-spectra y of Eq. (10.22) using simulated data. The distribution of the noise log-spectrum n is Gaussian with mean 0 dB and standard deviation of 2 dB. The distribution of the clean log-spectrum x is Gaussian with mean 25 dB and standard deviations of 25, 10, and 5 dB, respectively (the x-axis is expressed in dB). The first distribution is bimodal, whereas the other two are approximately Gaussian. Curves are plotted using Monte Carlo simulation.

Figure 10.4 shows the distribution of y for two values of μ_x, given the same values for the noise distribution, $\mu_n = 0\,\text{dB}$ and $\sigma_n = 2\,\text{dB}$, and a more realistic value for $\sigma_x = 5\,\text{dB}$. We see that the distribution is always unimodal, though not necessarily symmetric, particularly for low SNR ($\mu_x - \mu_n$).

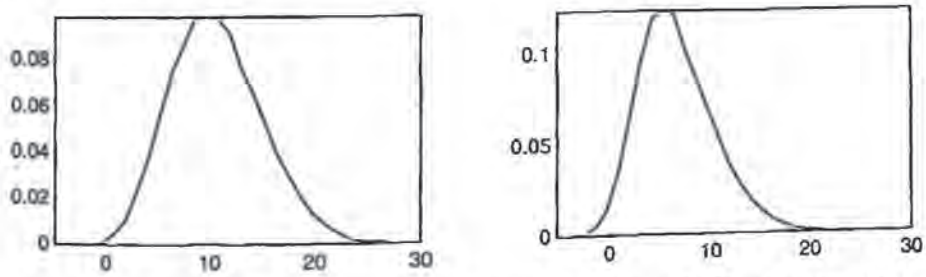


Figure 10.4 Distributions of the corrupted log-spectra y of Eq. (10.22) using simulated data. The distribution of the noise log-spectrum n is Gaussian with mean 0 dB and standard deviation of 2 dB. The distribution of the clean log-spectrum is Gaussian with standard deviation of 5 dB and means of 10 and 5 dB, respectively. The first distribution is approximately Gaussian while the second is nonsymmetric. Curves are plotted using Monte Carlo simulation.

The distributions used in an HMM are mixtures of Gaussians so that, even if each Gaussian component is transformed into a non-Gaussian distribution, the composite distribution can be modeled adequately by another mixture of Gaussians. In fact, if you retrain the model using the standard Gaussian assumption on corrupted speech, you can get good results, so this approximation is not bad.

10.2. ACOUSTICAL TRANSDUCERS

Acoustical transducers are devices that convert the acoustic energy of sound into electrical energy (microphones) and vice versa (loudspeakers). In the case of a microphone this transduction is generally realized with a diaphragm, whose movement in response to sound pressure varies the parameters of an electrical system (a variable-resistance conductor, a condenser, etc.), producing a variable voltage that constitutes the microphone output. We focus on microphones because they play an important role in designing speech recognition systems.

There are near field or close-talking microphones, and far field microphones. Close-talking microphones, either head-mounted or telephone handsets, pick up much less background noise, though they are more sensitive to throat clearing, lip smacks, and breath noise. Placement of such a microphone is often very critical, since, if it is right in front of the mouth, it can produce pops in the signal with plosives such as /p/. Far field microphones can be lapel mounted or desktop mounted and pick up more background noise than near field microphones. Having a small but variable distance to the microphone could be worse than a larger but more consistent distance, because the corresponding HMM may have lower variability.

When used in speech recognition systems, the most important measurement is the signal-to-noise ratio (SNR), since the lower the SNR the higher the error rate. In addition, different microphones have different transfer functions, and even the same microphone offers different transfer functions depending on the distance between mouth and microphone. Varying noise and channel conditions are a challenge that speech recognition systems have to address, and in this chapter we present some techniques to combat them.

The most popular type of microphone is the condenser microphone. We shall study in detail its directionality patterns, frequency response, and electrical characteristics.

10.2.1. The Condenser Microphone

A *condenser microphone* has a capacitor consisting of a pair of metal plates separated by an insulating material called a dielectric (see Figure 10.5). Its capacitance C is given by

$$C = \varepsilon_0 \pi b^2 / h \tag{10.23}$$

where ε_0 is a constant, b is the width of the plate, and h is the separation between the plates. If we polarize the capacitor with a voltage V_{cc}, it acquires a charge Q given by

$$Q = CV_{cc} \tag{10.24}$$

One of the plates is free to move in response to changes in sound pressure, which results in a change in the plate separation Δh, thereby changing the capacitance and producing a change in voltage $\Delta V = \Delta h V_{cc} / h$. Thus, the sensitivity[4] of the microphone depends on the polarizing voltage V_{cc}, which is why this voltage can often be 100 V or more.

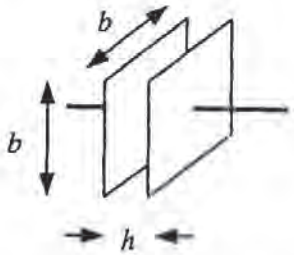


Figure 10.5 A diagram of a condenser microphone.

Electret microphones are a type of condenser microphones that do not require a special polarizing voltage V_{cc}, because a charge is impressed on either the diaphragm or the back plate during manufacturing and it remains for the life of the microphone. Electret microphones are light and, because of their small size, they offer good responses at high frequencies.

From the electrical point of view, a microphone is equivalent to a voltage source $v(t)$ with an impedance Z_M, as shown in Figure 10.6. The microphone is connected to a preamplifier which has an equivalent impedance R_L.

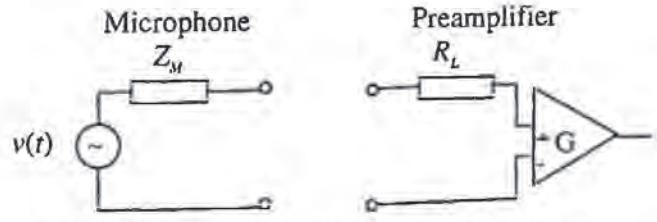


Figure 10.6 Electrical equivalent of a microphone.

[4] The sensitivity of a microphone measures the *open-circuit voltage* of the electric signal the microphone delivers for a sound wave for a given sound pressure level, often 94 dB SPL, when there is no load or a high impedance. This voltage is measured in dBV, where the 0-dB reference is 1 V rms.

From Figure 10.6 we can see that the voltage on R_L is

$$v_R(t) = v(t)\frac{R_L}{(R_M + R_L)} \tag{10.25}$$

Maximization of $v_R(t)$ in Eq. (10.25) results in $R_L = \infty$, or in practice $R_L >> R_M$, which is called *bridging*. Thus, for highest sensitivity the impedance of the amplifier has to be at least 10 times higher than that of the microphone. If the microphone is connected to an amplifier with lower impedance, there is a *load loss* of signal level. Most low-impedance microphones are labeled as 150 ohms, though the actual values may vary between 100 and 300. Medium impedance is 600 ohms and high impedance is 600–10,000 ohms. In practice, the microphone impedance is a function of frequency. Signal power is measured in dBm, where the 0-dB reference corresponds to 1 mW dissipated in a 600-ohm resistor. Thus, 0 dBm is equivalent to 0.775 V.

Since the output impedance of a condenser microphone is very high (~ 1 Mohm), a JFET transistor must be coupled to lower the equivalent impedance. Such a transistor needs to be powered with DC voltage through a different wire, as in Figure 10.7. A standard sound card has a jack with the audio on the tip, ground on the sleeve, DC bias V_{DD} on the ring, and a medium impedance. When using *phantom power*, the V_{CC} bias is provided directly in the audio signal, which must be *balanced* to ground.

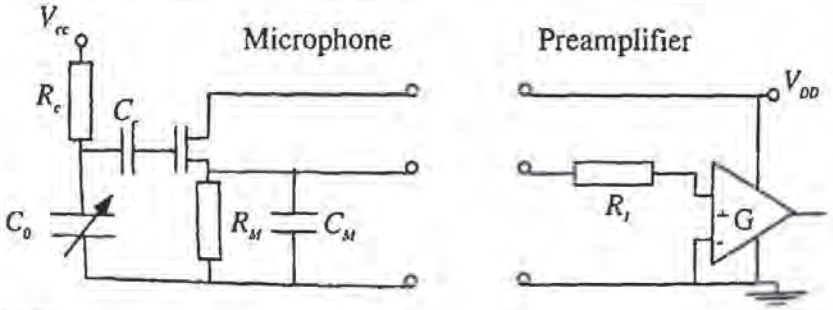


Figure 10.7 Equivalent circuit for a condenser microphone with DC bias on a separate wire.

It is important to understand how noise affects the signal of a microphone. If thermal noise arises in the resistor R_L, it will have a power

$$P_N = 4kTB \tag{10.26}$$

where $k = 1.38 \times 10^{-23}$ J/K is the Bolzmann's constant, T is the temperature in °K, and B is the bandwidth in Hz. The thermal noise in Eq. (10.26) at room temperature ($T = 297$°K) and for a bandwidth of 4 kHz is equivalent to –132 dBm. In practice, the noise is significantly higher than this because of preamplifier noise, radio-frequency noise and electromagnetic interference (poor grounding connections). It is, thus, important to keep the signal path between the microphone and the preamp as short as possible to avoid extra noise. It is desir-

able to have a microphone with low impedance to decrease the effect of noise due to radio-frequency interference, and to decrease the signal loss if long cables are used. Most microphones specify their SNR and range where they are linear (dynamic range). For condenser microphones, a power supply is necessary (DC bias required). Microphones with balanced output (the signal appears across two inner wires not connected to ground, with the shield of the cable connected to ground) are more resistant to radio frequency interference.

10.2.2. Directionality Patterns

A microphone's directionality pattern measures its sensitivity to a particular direction. Microphones may also be classified by their directional properties as *omnidirectional* (or *nondirectional*) and *directional*, the latter subdivided into bidirectional and unidirectional, based upon their response characteristics.

10.2.2.1. Omnidirectional Microphones

By definition, the response of an omnidirectional microphone is independent of the direction from which the encroaching sound wave is coming. Figure 10.8 shows the polar response of an omnidirectional mike. A microphone's *polar response*, or *pickup pattern*, graphs its output voltage for an input sound source with constant level at various angles around the mic. Typically, a polar response assumes a preferred direction, called the major axis or front of the microphone, which corresponds to the direction at which the microphone is most sensitive. The front of the mike is labeled as zero degrees on the polar plot, but since an omnidirectional mic has no particular direction at which it is the most sensitive, the omnidirectional mike has no true front and hence the zero-degree axis is arbitrary. Sounds coming from any direction around the microphone are picked up equally. Omnidirectional microphones provide no noise cancellation.

Figure 10.8 (a) Polar response of an ideal omnidirectional microphone and (b) its cross section.

Figure 10.8 shows the mechanics of the ideal[5] omnidirectional condenser microphone. A sound wave creates a pressure all around the microphone. The pressure enters the opening of the mike and the diaphragm moves. An electrical circuit converts the diaphragm move-

[5] Ideal omnidirectional microphones do not exist.

ment into an electrical voltage, or response. Sound waves impinging on the mike create a pressure at the opening regardless of the direction from which they are coming; therefore we have a nondirectional, or omnidirectional, microphone. As we have seen in Chapter 2, if the source signal is $Be^{j\omega t}$, the signal at a distance r is given by $(A/r)e^{j\omega t}$ independently of the angle.

This is the most inexpensive of the condenser microphones, and it has the advantage of a flat frequency response that doesn't change with the angle or distance to the microphone. On the other hand, because of its uniform polar pattern, it picks up not only the desired signal but also noise from any direction. For example, if a pair of speakers is monitoring the microphone output, the sound from the speakers can reenter the microphone and create an undesirable sound called *feedback*.

10.2.2.2. Bidirectional Microphones

The bidirectional microphone is a *noise-canceling* microphone; it responds less to sounds incident from the sides. The bidirectional mike utilizes the properties of a gradient microphone to achieve its noise-canceling polar response. You can see how this is accomplished by looking at the diagram of a simplified gradient bidirectional condenser microphone, as shown in Figure 10.9. A sound impinging upon the front of the microphone creates a pressure at the front opening. A short time later, this same sound pressure enters the back of the microphone. The sound pressure never arrives at the front and back at the same time. This creates a displacement of the diaphragm and, just as with the omnidirectional mike, a corresponding electrical signal. For sounds impinging from the side, however, the pressure from an incident sound wave at the front opening is identical to the pressure at the back. Since both openings lead to one side of the diaphragm, there is no displacement of the diaphragm, and the sound is not reproduced.

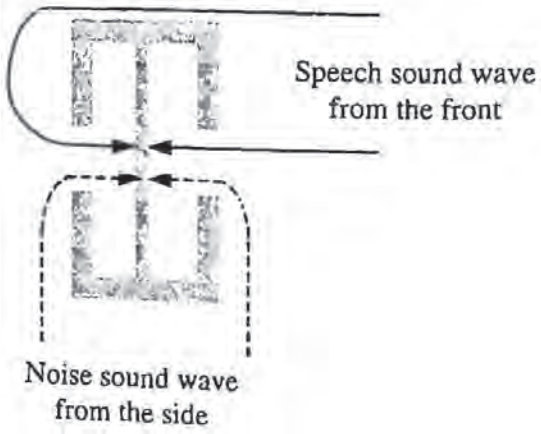


Figure 10.9 Cross section of an ideal bidirectional microphone.

To compute the polar response of this gradient microphone let's make the approximation of Figure 10.10, where the microphone signal is the difference between the signal at the front and rear of the diaphragm, the separation between plates is $2d$, and r is the distance between the source and the center of the microphone.

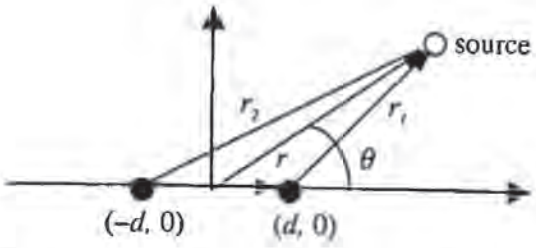


Figure 10.10 Approximation to the noise-canceling microphone of Figure 10.9.

You can see that r_1, the distance between the source and the front of the diaphragm, is the norm of the vector specifying the source location minus the vector specifying the location of the front of the diaphragm

$$r_1 = \left| re^{j\theta} - d \right| \tag{10.27}$$

Similarly, you obtain the distance between the source and the rear of the diaphragm

$$r_2 = \left| re^{j\theta} + d \right| \tag{10.28}$$

The source arrives at the front of the diaphragm with a delay $\delta_1 = r_1 / c$, where c is the speed of sound in air. Similarly, the delay to the rear of the diaphragm is $\delta_2 = r_2 / c$. If the source is a complex exponential $e^{j\omega t}$, the difference signal between the front and rear is given by

$$x(t) = \frac{A}{r_1} e^{j2\pi f(t-\delta_1)} - \frac{A}{r_2} e^{j2\pi f(t-\delta_2)} = \frac{A}{r} e^{j2\pi ft} G(f,\theta) \tag{10.29}$$

where A is a constant and, using Eqs. (10.27), (10.28) and (10.29), the gain $G(f,\theta)$ is given by

$$G(f,\theta) = \frac{e^{-j2\pi \left| e^{j\theta} - \lambda \right| \tau f}}{\left| e^{j\theta} - \lambda \right|} - \frac{e^{-j2\pi \left| e^{j\theta} + \lambda \right| \tau f}}{\left| e^{j\theta} + \lambda \right|} \tag{10.30}$$

where we have defined $\lambda = d/r$ and $\tau = r/c$.

The magnitude of Eq. (10.30) is used to plot the polar response of Figure 10.11. As can be seen by the plot, the pattern resembles a figure eight. The bidirectional mike has an interchangeable front and back, since the response has a maximum in two opposite directions. In practice, this bidirectional microphone is an ideal case, and the polar response has to be measured empirically.

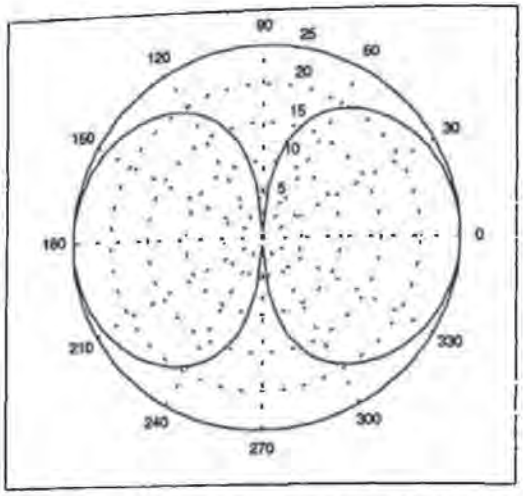


Figure 10.11 Polar response of a bidirectional microphone obtained through Eq. (10.30) with $d = 1$ cm, $r = 50$ cm, $c = 33{,}000$ cm/s, and $f = 1000$ Hz.

According to the idealized model, the frequency response of omnidirectional microphones is constant with frequency, and this approximately holds in practice for real omnidirectional microphones. On the other hand, the polar pattern of directional microphones is not constant with frequency. Clearly it is a function of frequency, as can be seen in Eq. (10.30). In fact, the frequency response of a bidirectional microphone at 0° is shown in Figure 10.12 for both near field and far field conditions.

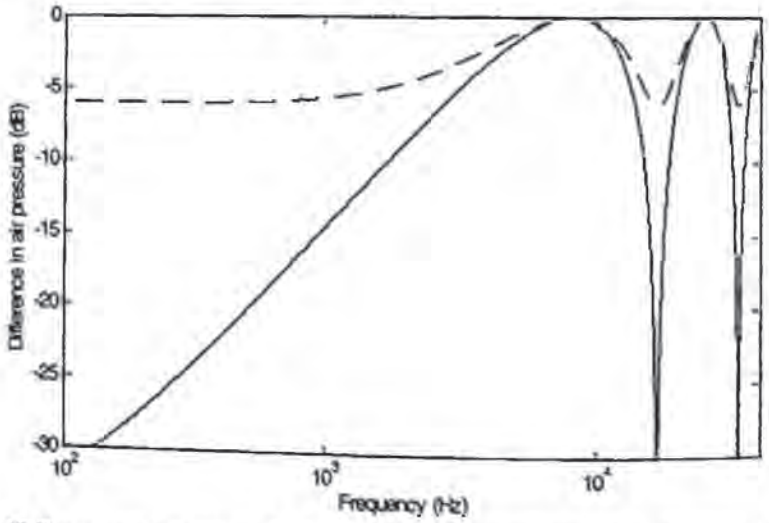


Figure 10.12 Frequency response of a bidirectional microphone with $d = 1$ cm at 0° obtained through Eq. (10.30). The larger the distance between plates, the lower the frequency of the maxima. The highest values are obtained for 8250 Hz and 24,750 Hz and the null for 16,500 Hz. The solid line corresponds to far field conditions ($\lambda = 0.02$) and the dotted line to near field conditions ($\lambda = 0.5$).

It can be shown, after taking the derivative of $G(f,0)$ in Eq. (10.30) and equating to zero, that the maxima are given by

$$f_n = \frac{c}{4d}(2n-1) \tag{10.31}$$

with $n = 1,2,\cdots$. We can observe from Eq. (10.31) that the larger the width of the diaphragm, the lower the first maximum.

The increase in frequency response, or sensitivity, in the near field, compared to the far field, is a measure of noise cancellation. Consequently the microphone is said to be noise canceling. The microphone is also referred to as a differential or gradient microphone, since it measures the gradient (difference) in sound pressure between two points in space. The boost in low-frequency response in the near field is also referred to as the *proximity effect*, often used by singers to boost their bass levels by getting the microphone closer to their mouths.

By evaluating Eq. (10.30) it can be seen that low-frequency sounds in a bidirectional microphone are not reproduced as well as higher frequencies, leading to a *thin* sounding mike.

Let's interpret Figure 10.12. The net sound pressure between these two points, separated by a distance $D = 2d$, is influenced by two factors: phase shift and inverse square law.

The influence of the sound-wave phase shift is less at low frequencies than at high, because the distance D between the front and rear port entries becomes a small fraction of the low-frequency wavelength. Therefore, there is little phase shift between the ports at low frequencies, as the opposite sides of the diaphragm receive nearly equal amplitude and phase. The result is slight diaphragm motion and a weak microphone output signal. At higher frequencies, the distance D between sound ports becomes a larger fraction of the wavelength. Therefore, more phase shift exists across the diaphragm. This causes a higher microphone output.

The pressure difference caused by phase shift rises with frequency at a rate of 20 dB per decade. As the frequency rises to where the microphone port spacing D equals half a wavelength, the net pressure is at its maximum. In this situation, the diaphragm movement is also at its maximum, since the front and rear see equal amplitude but in opposite polarities of the wave front. This results in a peak in the microphone frequency response, as illustrated in Figure 10.12. As the frequency continues to rise to where the microphone port spacing D equals one complete wavelength, the net pressure is at its minimum. Here, the diaphragm does not move at all, since the front and rear sides see equal amplitude at the same polarity of the wave front. This results in a dip in the microphone frequency response, as shown in Figure 10.12.

A second factor creating a net pressure difference across the diaphragm is the impact of the inverse square law. If the sound-pressure difference between the front and rear ports of a noise-canceling microphone were measured near the sound source and again further from the source, the near field measurement would be greater than the far field. In other words, the microphone's net pressure difference and, therefore, output signal, is greater in the near sound field than in the far field. The inverse-square-law effect is independent of frequency. The net pressure that causes the diaphragm to move is a combination of both the phase shift and inverse-square-law effect. These two factors influence the frequency response of the microphone differently, depending on the distance to the sound source. For distant sound, the influence of the net pressure difference from the inverse-square-law effect is weaker than the phase-shift effect; thus, the rising 20-dB-per-decade frequency response dominates the total frequency response. As the microphone is moved closer to the sound source, the influence of the net pressure difference from the inverse square law is greater than that of the phase shift; thus the total microphone frequency response is largely flat.

The difference in near field to far field frequency response is a characteristic of all noise-canceling microphones and applies equally to both acoustic and electronic types.

10.2.2.3. Unidirectional Microphones

Unidirectional microphones are designed to pick-up the speaker's voice by directing the audio reception toward the speaker, focusing on the desired input and rejecting sounds emanating from other directions that can negatively impact clear communications, such as computer noise from fans or other sounds.

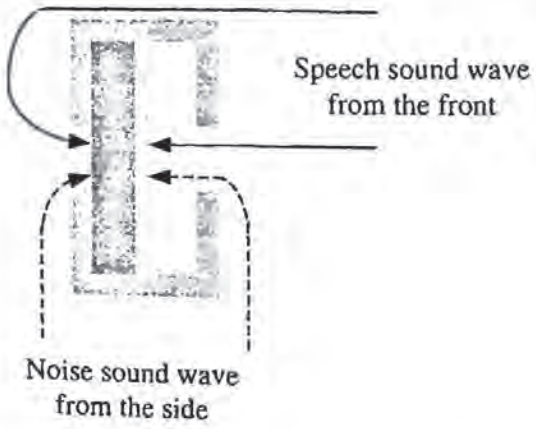


Figure 10.13 Cross section of a unidirectional microphone.

Figure 10.13 shows the cross-section of a unidirectional microphone, which also relies upon the principles of a gradient microphone. Notice that the unidirectional mic looks similar to the bidirectional, except that there is a resistive material (often cloth or foam) between the diaphragm and the opening of one end. The material's resistive properties *slow down* the pressure on its path from the back opening to the diaphragm. If the additional delay through the back plate is given by τ_0, the gain can be given by

$$G(f,\theta)=\frac{e^{-j2\pi\left|e^{j\theta}-\lambda\right|rf}}{\left|e^{j\theta}-\lambda\right|}-\frac{e^{-j2\pi\left(\tau_0+\left|e^{j\theta}+\lambda\right|r\right)f}}{\left|e^{j\theta}+\lambda\right|} \tag{10.32}$$

which was obtained by modifying Eq. (10.30). Unidirectional microphones have the greatest response to sound waves impinging from one direction, typically referred to as the front, or major axis of the microphone. One typical response of a unidirectional microphone is the *cardioid* pattern shown in the polar plot of Figure 10.14, plotted from Eq. (10.32). The frequency response at 0° is similar to that of Figure 10.12. Because the cardioid pattern of polar response is so popular among them, unidirectional mikes are often referred to as cardioid mikes.

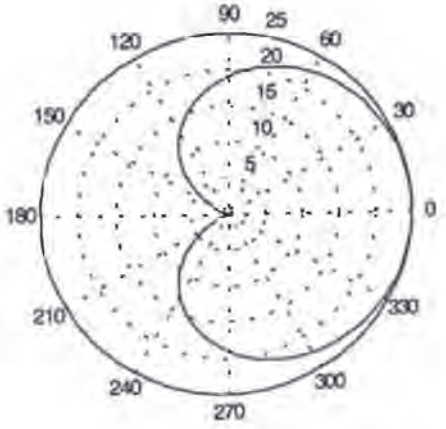


Figure 10.14 Polar response of a unidirectional microphone. The polar response was obtained through Eq. (10.32) with $d = 1$ cm, $r = 50$ cm, $c = 33{,}000$ cm/s, $f = 1$ kHz, and $\tau_0 = 0.06$ ms.

Equation (10.32) was derived under a simplified schematic based on Figure 10.10, which is an idealized model so that, in practice, the polar response of a real microphone has to be measured empirically. The frequency response and polar pattern of a commercial microphone are shown in Figure 10.15.

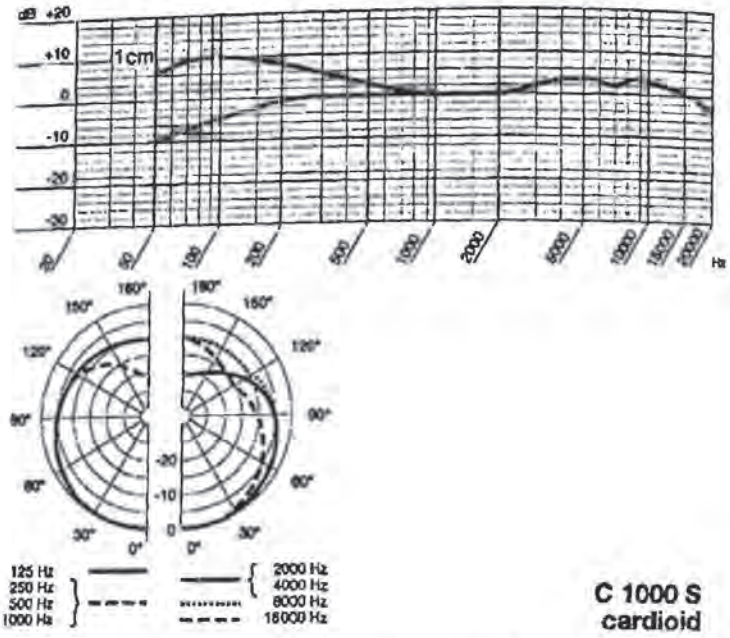


Figure 10.15 Characteristics of an AKG C1000S cardioid microphone: (top) frequency response for near and far field conditions (note the proximity effect) and (bottom) polar pattern for different frequencies.

Although this noise cancellation decreases the overall response to sound pressure (*sensitivity*) of the microphone, the directional and frequency-response improvements far outweigh the lessened sensitivity. It is particularly well suited for use as a desktop mic or as part of an embedded microphone in a laptop or desktop computer. Unidirectional microphones achieve superior noise-rejection performance over omnidirectionals. Such performance is necessary for clean audio input and for audio signal processing algorithms such as acoustic echo cancellation, which form the core of speakerphone applications.

10.2.3. Other Transduction Categories

In a *passive microphone*, sound energy is directly converted to electrical energy, whereas an *active microphone* requires an external energy source that is modulated by the sound wave. Active transducers thus require phantom power, but can have higher sensitivity.

We can also classify microphones according to the physical property to which the sound wave responds. A *pressure microphone* has an electrical response that corresponds to the pressure in a sound wave, while a *pressure gradient microphone* has a response corresponding to the difference in pressure across some distance in a sound wave. A pressure microphone is a fine reproducer of sound, but a gradient microphone typically has a response greatest in the direction of a desired signal or talker and rejects undesired background sounds. This is particularly beneficial in applications that rely upon the reproduction of only a desired signal, where any undesired signal entering the reproduction severely degrades performance. Such is the case in voice recognition or speakerphone applications.

In terms of the mechanism by which they create an electrical signal corresponding to the sound wave they detect, microphones are classified as *electromagnetic*, *electrostatic*, and *piezoelectric*. *Dynamic* microphones are the most popular type of electromagnetic microphone and *condenser* microphones the most popular type of electrostatic microphone.

Electromagnetic microphones induce voltage based on a varying magnetic field. *Ribbon microphones* are a type of electromagnetic microphones that employ a thin metal ribbon suspended between the poles of a magnet. *Dynamic* microphones are electromagnetic microphones that employ a moving coil suspended by a light diaphragm (see Figure 10.16), acting like a speaker but in reverse. The diaphragm moves with changes in sound pressure, which in turns moves the coil, which causes current to flow as lines of flux from the magnet are cut. Dynamic microphones need no batteries or power supply, but they deliver low signal levels that need to be preamplified.

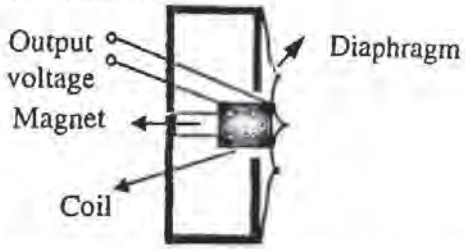


Figure 10.16 Dynamic microphone schematics.

Piezoresistive and piezoelectric microphones are based on the variation of electric resistance of their sensor induced by changes in sound pressure. *Carbon button microphones* consist of a small cylinder packed with tiny granules of carbon that, when compacted by sound pressure, reduce the electric resistance. Such microphones, often used in telephone handsets, offer a worse frequency response than condenser microphones, and lower dynamic range.

10.3. ADAPTIVE ECHO CANCELLATION (AEC)

If a spoken language system allows the user to talk while speech is being output through the loudspeakers, the microphone picks up not only the user's voice, but also the speech from the loudspeaker. This problem may be avoided with a *half-duplex* system that does not listen when a signal is being played through the loudspeaker, though such systems offer an unnatural user experience. On the other hand, a *full-duplex* system that allows *barge-in* by the user to interrupt the system offers a better user experience. For barge-in to work, the signal played through the loudspeaker needs to be canceled. This is achieved with echo cancellation (see Figure 10.17), as discussed in this section.

In hands-free conferencing the local user's voice is output by the remote loudspeaker, whose signal is captured by the remote microphone and after some delay is output by the local loudspeaker. People are tolerant to these echoes if either they are greatly attenuated or the delay is short. Perceptual studies have shown that the longer the delay, the greater the attenuation needed for user acceptance.

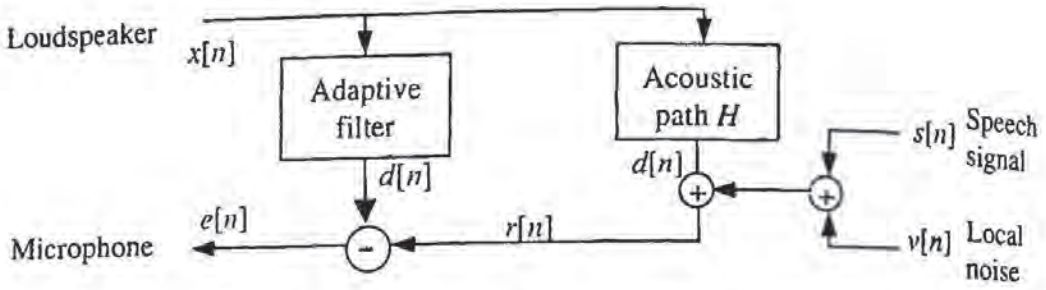


Figure 10.17 Block diagram of an echo-canceling application. $x[n]$ represents the signal from the loudspeaker, $s[n]$ the speech signal, $v[n]$ the local background noise, and $e[n]$ the signal that goes to the microphone.

The use of *echo cancellation* is mandatory in telephone communications and hands-free conferencing when it is desired to have full-duplex voice communication. This is particularly important when the call is routed through a satellite that can have delays larger than 200 ms. A block diagram is shown in Figure 10.18.

In Figure 10.17, the return signal $r[n]$, assuming no local noise, is the sum

$$r[n] = d[n] + s[n] \tag{10.33}$$

where $s[n]$ is the speech signal and $d[n]$ is the attenuated and possibly distorted version of the loudspeaker's signal $x[n]$. The purpose of the echo canceler is to remove the echo $d[n]$ from the return signal $r[n]$, which is done by means of an adaptive FIR filter whose coefficients are computed to minimize the energy of the canceled signal $e[n]$. The filter coefficients are reestimated *adaptively* to track slowly changing line conditions.

This problem is essentially that of adaptive filtering *only* when $s[n] = 0$, or in other words when the user is silent. For this reason, you have to implement a *double-talk detection* module that detects when the speaker is silent. This is typically feasible because the echo $d[n]$ is usually small, and if the return signal $r[n]$ has high energy it means that the user is

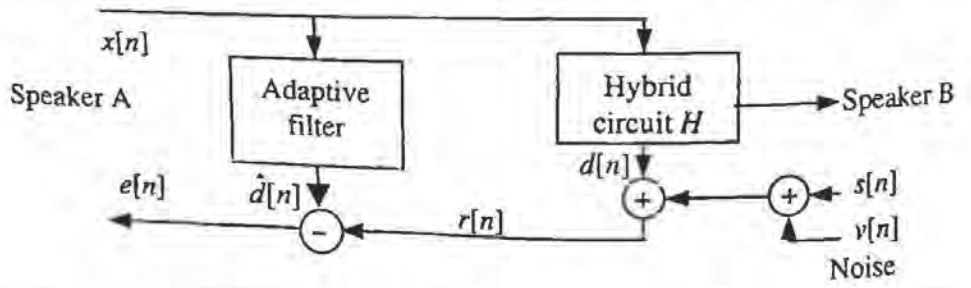


Figure 10.18 Block diagram of echo canceling for a telephone communication. $x[n]$ represents the remote call signal, $s[n]$ the local outgoing signal. The hybrid circuit H does a 2-4 wire conversion and is nonideal because of impedance mismatches.

not silent. Errors in double-talk detection result in divergence of the filter, so it is generally preferable to be conservative in the decision and when in doubt not adapt the filter coefficients. Initialization could be done by sending a known signal with white spectrum. The quality of the filtering is measured by the so-called *echo-return loss enhancement* (ERLE):

$$ERLE(dB) = 10\log_{10}\frac{E\{d^2[n]\}}{E\{(d[n]-\hat{d}[n])^2\}} \tag{10.34}$$

The filter coefficients are chosen to maximize the ERLE. Since the telephone-line characteristics, or the acoustic path (due to speaker movement), can change over time, the filter is often adaptive. Another reason for adaptive filters is that reliable ERLE maximization requires a large number of samples, and such a delay is not tolerable.

In the following sections, we describe the fundamentals of adaptive filtering. While there are some nonlinear adaptive filters, the vast majority are linear FIR filters, with the LMS algorithm being the most important. We introduce the LMS algorithm, study its convergence properties, and present two extensions: the normalized LMS algorithm and transform-domain LMS algorithms.

10.3.1. The LMS Algorithm

Let's assume that a desired signal $d[n]$ is generated from an input signal $x[n]$ as follows

$$d[n] = \sum_{k=0}^{L-1} g_k x[n-k] + u[n] = \mathbf{G}^T\mathbf{X}[n] + u[n] \tag{10.35}$$

with $\mathbf{G} = \{g_0, g_1, \cdots g_{L-1}\}$, the input signal vector $\mathbf{X}[n] = \{x[n], x[n-1], \cdots x[n-L+1]\}$, and $u[n]$ being noise that is independent of $x[n]$.

We want to estimate $d[n]$ in terms of the sum of previous samples of $x[n]$. To do that we define the estimate signal $y[n]$ as

$$y[n] = \sum_{k=0}^{L-1} w_k[n]x[n-k] = \mathbf{W}^T[n]\mathbf{X}[n] \tag{10.36}$$

where $\mathbf{W}[n] = \{w_0[n], w_1[n], \cdots w_{L-1}[n]\}$ is the time-dependent coefficient vector. The instantaneous error between the desired and the estimated signal is given by

$$e[n] = d[n] - \mathbf{W}^T[n]\mathbf{X}[n] \tag{10.37}$$

The *least mean square* (LMS) algorithm updates the value of the coefficient vector in the steepest descent direction

$$\mathbf{W}[n+1] = \mathbf{W}[n] + \varepsilon e[n]\mathbf{X}[n] \tag{10.38}$$

where ε is the step size. This algorithm is very popular because of its simplicity and effectiveness [58].

10.3.2. Convergence Properties of the LMS Algorithm

The choice of ε is important: if it is too small, the adaptation rate will be slow and it might not even track the nonstationary trends of $x[n]$, whereas if ε is too large, the error might actually increase. We analyze the conditions under which the LMS algorithm converges.

Let's define the error in the coefficient vector $\mathbf{V}[n]$ as

$$\mathbf{V}[n] = \mathbf{G} - \mathbf{W}[n] \tag{10.39}$$

and combine Eqs. (10.35), (10.37), (10.38), and (10.39) to obtain

$$\mathbf{V}[n+1] = \mathbf{V}[n] - \varepsilon\mathbf{X}[n]\mathbf{X}^T[n]\mathbf{V}[n] - \varepsilon u[n]\mathbf{X}[n] \tag{10.40}$$

Taking expectations in Eq. (10.40) results in

$$E\{\mathbf{V}[n+1]\} = E\{\mathbf{V}[n]\} - \varepsilon E\{\mathbf{X}[n]\mathbf{X}^T[n]\mathbf{V}[n]\} \tag{10.41}$$

where we have assumed that $u[n]$ and $x[n]$ are independent and that either is a zero-mean process. Finally, we express the autocorrelation of $\mathbf{X}[n]$ as

$$\mathbf{R}_{xx} = E\{\mathbf{X}[n]\mathbf{X}^T[n]\} = \mathbf{Q}\Lambda\mathbf{Q}^T \tag{10.42}$$

where $\mathbf{Q}$ is a matrix of its eigenvectors and Λ is a diagonal matrix of its eigenvalues $\{\lambda_0, \lambda_1, \cdots, \lambda_{L-1}\}$, which are all real valued because of the symmetry of $\mathbf{R}_{xx}$.

Although we know that $\mathbf{X}[n]$ and $\mathbf{V}[n]$ are not statistically independent, we assume in this section that they are, so that we can obtain some insight on the convergence properties. With this assumption, Eq. (10.41) can be expressed as

$$E\{\mathbf{V}[n+1]\} = E\{\mathbf{V}[n]\}(1 - \varepsilon\mathbf{R}_{xx}) \tag{10.44}$$

which, applied recursively, leads to

$$E\{\mathbf{V}[n+1]\} = E\{\mathbf{V}[0]\}(1 - \varepsilon\mathbf{R}_{xx})^n \tag{10.45}$$

Using Eqs. (10.39) and (10.42) in (10.45), we can express the $(i+1)$th element of $E\{\mathbf{W}[n]\}$ as

$$E\{w_i[n]\} = g_i + \sum_{j=0}^{L-1} q_{ij}(1-\varepsilon\lambda_j)^n E\{\tilde{v}_j[0]\} \tag{10.46}$$

where q_{ij} is the $(i+1, j+1)$th element of the eigenvector matrix $\mathbf{Q}$, and $\tilde{v}_i[n]$ is the rotated coefficient error vector defined as

$$\tilde{\mathbf{V}}[n] = \mathbf{Q}^T\mathbf{V}[n] \tag{10.47}$$

From Eq. (10.46) we see that the mean value of the LMS filter coefficients converges exponentially to the true value if

$$0 < \varepsilon < 1/\lambda_j \tag{10.48}$$

so that the adaptation constant ε must be determined from the largest eigenvalue of $\mathbf{X}[n]$ for the mean LMS algorithm to converge.

10.3.3. Normalized LMS Algorithm

In practice, mean convergence doesn't tell us the nature of the fluctuations that the coefficients experience. Analysis of the variance of $\mathbf{V}[n]$ together with some more approximations result in mean-squared convergence if

$$0 < \varepsilon < \frac{K}{L\sigma_x^2} \tag{10.49}$$

with $\sigma_x^2 = E\{x^2[n]\}$ being the input signal power and K a constant that depends weakly on the nature of the input signal statistics but not on its power.

Because of the inaccuracies of the independence assumptions above, a rule of thumb used in practice to determine the adaptation constant ε is

$$0 < \varepsilon < \frac{0.1}{L\sigma_x^2} \tag{10.50}$$

The choice of largest value for ε in Eq. (10.49) makes the LMS algorithm track nonstationary variations in x fastest, and achieve faster convergence. On the other hand, the misadjustment of the filter coefficients increases as both the filter length L and adaptation constant ε increase. For this reason, often the adaptation constant can be made a function of

n ($\varepsilon[n]$), with larger values at first and smaller values once convergence has been determined.

The *normalized LMS algorithm* (NLMS) uses the result of Eq. (10.49) and, therefore, defines a normalized step size

$$\varepsilon[n]=\frac{\varepsilon}{\delta+L\hat{\sigma}_x^2[n]} \tag{10.51}$$

where the constant δ avoids a division by 0 and $\hat{\sigma}_x^2[n]$ is an estimate of the input signal power, which is typically done with an exponential window

$$\hat{\sigma}_x^2[n]=(1-\beta)\hat{\sigma}_x^2[n-1]+\beta x^2[n] \tag{10.52}$$

or a sliding rectangular window

$$\hat{\sigma}_x^2[n]=\frac{1}{N}\sum_{i=0}^{N-1}x^2[n-i]=\hat{\sigma}_x^2[n-1]+\frac{1}{N}\left(x^2[n]-x^2[n-N]\right) \tag{10.53}$$

where both β and N control the effective memory of the estimators in Eqs. (10.52) and (10.53), respectively. Finally, we need to pick ε so that $0<\varepsilon<2$ to assure convergence. Choice of the NLMS algorithm simplifies the selection of ε, and the NLMS often converges faster than the LMS algorithm in practical situations.

10.3.4. Transform-Domain LMS Algorithm

As discussed in Section 10.3.2, convergence of the LMS algorithm is determined by the largest eigenvalue of the input. Since complex exponentials are approximate eigenvectors for LTI systems, the LMS algorithm's convergence is dominated by the frequency band with largest energy, and convergence in other frequency bands is generally much slower. This is the rationale for the *subband LMS algorithm*, which performs independent LMS algorithms for different frequency bands, as proposed by Boll [14].

The *block LMS* (BLMS) algorithm keeps the coefficients unchanged for a block k of L samples

$$\mathbf{W}[k+1]=\mathbf{W}[k]+\varepsilon\sum_{m=0}^{L-1}e[kL+m]\mathbf{X}[kL+m] \tag{10.54}$$

which is represented by a linear convolution and therefore can be implemented efficiently using length-$2N$ FFTs according to overlap-save method of Figure 10.19. Notice that implementing a linear convolution with a circular convolution operator such as the FFT requires the use of the dashed box.

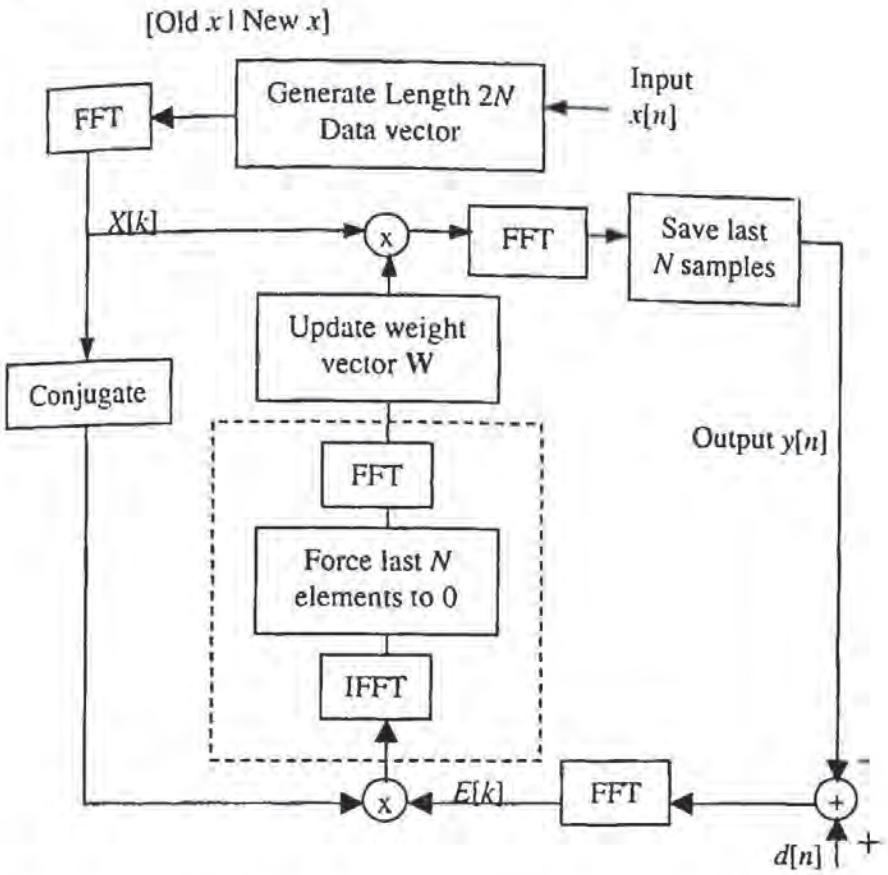


Figure 10.19 Block diagram of the constrained frequency-domain block LMS algorithm. The unconstrained version of this algorithm eliminates the computation inside the dashed box.

An unconstrained frequency-domain LMS algorithm can be implemented by removing the constraint in Figure 10.19, therefore implementing a circular instead of a linear convolution. While this is not exact, the algorithm requires only three FFTs instead of five. In some practical applications, there is no difference in convergence between the constrained and unconstrained cases.

10.3.5. The RLS Algorithm

The search for the optimum filter can be accelerated when the gradient vector is properly deviated toward the minimum. This approach uses the Newton-Raphson method to iteratively compute the root of $f(x)$ (see Figure 10.20) so that the value at iteration $i + 1$ is given by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{10.55}$$

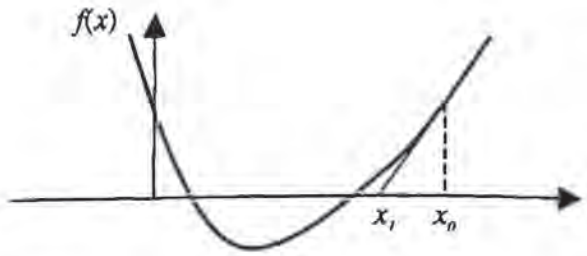


Figure 10.20 Newton-Raphson method to compute the roots of a function.

To minimize function $f(x)$ we thus compute the roots of $f'(x)$ through the above method:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)} \tag{10.56}$$

In the case of a vector, Eq. (10.56) is transformed into

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \varepsilon[n]\left(\nabla^2 e(\mathbf{w}_i)\right)^{-1} \nabla e(\mathbf{w}_i) \tag{10.57}$$

where we add a step size $\varepsilon[n]$, and where $\nabla^2 e(\mathbf{w}_i)$ is the *Hessian* of the least-squares function which, for Eq. (10.37), equals the autocorrelation of $\mathbf{x}$:

$$\nabla^2 e(\mathbf{w}_i) = \mathbf{R}[n] = E\{\mathbf{x}[n]\mathbf{x}^T[n]\} \tag{10.58}$$

The *recursive least squares* (RLS) algorithm specifies a method of estimating Eq. (10.58) using an exponential window:

$$\mathbf{R}[n] = \lambda \mathbf{R}[n-1] + \mathbf{x}[n]\mathbf{x}^T[n] \tag{10.59}$$

While the RLS algorithm converges faster than the LMS algorithm, it also is more computationally expensive, as it requires a matrix inversion for every sample. Several algorithms have been derived to speed it up [54].

10.4. MULTIMICROPHONE SPEECH ENHANCEMENT

The use of more than one microphone is motivated by the human auditory system, in which the use of both ears has been shown to enhance detection of the direction of arrival, as well as increase SNR when one ear is covered. The methods the human auditory system uses to accomplish this task are still not completely known, and the techniques described in this section do not mimic that behavior.

Microphone arrays use multiple microphones and knowledge of the microphone locations to predict delays and thus create a beam that focuses on the direction of the desired

speaker and rejects signals coming from other angles. Reverberation, as discussed in Section 10.1.2, can be combated with these techniques. Blind source separation techniques are another family of statistical techniques that typically do not use spatial constraints, but rather statistical independence between different sources.

While in this section we describe only linear processing, i.e., the output speech is a linearly filtered version of the microphone signals, we could also combine these techniques with the nonlinear methods of Section 10.5.

10.4.1. Microphone Arrays

The goals of microphone arrays are twofold: finding the position of a sound source in a room, and improving the SNR of the received signal. *Steering* is helpful in videoconferencing, where a camera has to follow the current speaker. Since the speaker is typically far away from the microphone, the received signal likely contains a fair amount of additive noise. Microphone arrays can also be used to increase the SNR.

Let $x[n]$ be the signal at the source S. Microphone i picks up a signal

$$y_i[n] = x[n] * g_i[n] + v_i[n] \tag{10.60}$$

that is a filtered version of the source plus additive noise $v_i[n]$. If we have N such microphones, we can attempt to recover $s[n]$ because all the signals $y_i[n]$ should be correlated.

A typical assumption made is that all the filters $g_i[n]$ are delayed versions of the same filter $g[n]$

$$g_i[n] = g[n - D_i] \tag{10.61}$$

with the delay $D_i = d_i / c$, d_i being the distance between the source S and microphone i, and c the speed of sound in air. We cannot recover signal $x[n]$ without knowledge of $g[n]$ or the signal itself, so the goal is to obtain the filtered signal $y[n]$

$$y[n] = x[n] * g[n] \tag{10.62}$$

so that, combining Eqs. (10.60), (10.61), and (10.62),

$$y_i[n] = y[n - D_i] + v_i[n] \tag{10.63}$$

Assuming $v_i[n]$ are independent and Gaussianly distributed, the optimal estimate of $x[n]$ is given by

$$\bar{y}[n] = \frac{1}{N} \sum_{i=0}^{N-1} y_i[n + D_i] = y[n] + v[n] \tag{10.64}$$

which is the so-called *delay-and-sum beamformer* [24, 29], where the residual noise $v[n]$

$$v[n] = \frac{1}{N}\sum_{i=0}^{N-1} v_i[n + D_i] \tag{10.65}$$

has a variance that decreases as the number of microphones N increases, since the noises $v_i[n + D_i]$ are uncorrelated.

Equation (10.64) requires estimation of the delays D_i. To attenuate the additive noise $v[n]$, it is not necessary to identify the absolute delays, but rather the delays relative to one reference microphone (for example, the center microphone). It can be shown that the maximum likelihood solution consists in maximizing the energy of $\tilde{y}[n]$ in Eq. (10.64), which is the sum of cross-correlations:

$$D_i = \arg\max_{D_i}\left(\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} R_{ij}[D_i - D_j]\right) \qquad 0 \le i < N \tag{10.66}$$

This approach assumes that we know nothing about the geometry of the microphone placement. In fact, given a point source and assuming no reflections, we can compute the delay based on the distance between the source and the microphone. The use of geometry allows us to reduce the number of parameters to estimate from $(N - 1)$ to a maximum of 3, in case we desire to estimate the exact location. This location is often described in spherical coordinates (φ, θ, ρ) with φ being the direction of arrival, θ the elevation angle, and ρ the distance to the reference microphone, as shown in Figure 10.21.

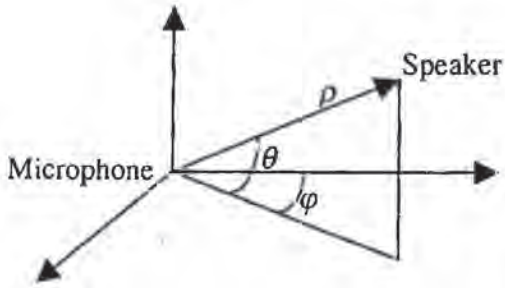


Figure 10.21 Spherical coordinates (φ, θ, ρ) with φ being the direction of arrival, θ the elevation angle, and ρ the distance to the reference microphone.

While 2-D and 3-D microphone configurations can be used, which would allow us to determine not just the steering angle φ, but also distance to the origin ρ and azimuth θ, linear microphone arrays are the most widely used configurations because they are the simplest. In a linear array all the microphones are placed on a line (see Figure 10.22). In this case, we cannot determine the elevation angle θ. To determine both φ and ρ we need at least two microphones in the array.

If the microphones are relatively close to each other compared to the distance to the source, the angle of arrival φ is approximately the same for all signals. With this assumption, the normalized delay $\bar{D}_i$ with respect to the reference microphone is given by

$$\bar{D}_i = -a_i \sin(\varphi)/c \tag{10.67}$$

where a_i is the y-axis coordinate in Figure 10.22 for microphone i, where the reference microphone has $a_0 = 0$ and also $\bar{D}_0 = 0$.

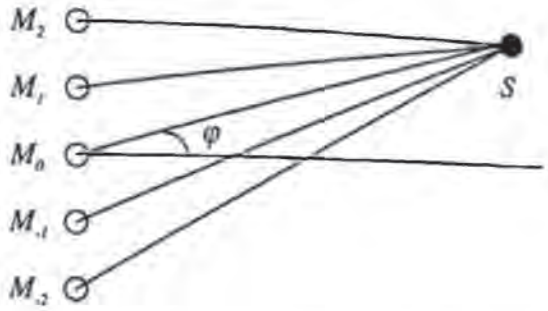


Figure 10.22 Linear microphone array (five microphones). The source signal arrives at each microphone with a different delay, which allows us to find the correct angle of arrival.

With approximation, we define $\bar{D}_i(\varphi)$, the relative delay of the signal at microphone i to the reference microphone, as a function of the direction of arrival angle φ and independent of ρ. The optimal direction of arrival φ is then that which maximizes the energy of the estimated signal $\tilde{x}[n]$ over a set of samples

$$\begin{aligned}\varphi &= \arg\max_{\varphi} \sum_n \left(\frac{1}{N} \sum_{i=0}^{N-1} y_i[n + \bar{D}_i(\varphi)] \right)^2 \\ &= \arg\max_{\varphi} \sum_n \left(\frac{1}{N} \sum_{i=0}^{N-1} y_i[n - \frac{a_i}{c} \sin(\varphi)] \right)^2\end{aligned} \tag{10.68}$$

The term *beamforming* entails that this array favors a specific direction of arrival φ and that sources arriving from other directions are not in phase and therefore are attenuated. Since the source can move over time, maximization of Eq. (10.68) can be done in an adaptive fashion.

As the beam is steered away from the broadside, the system exhibits a reduction in spatial discrimination because the beam pattern broadens. Furthermore, beamwidth varies with frequency, so an array has an approximate bandwidth given by the upper f_u and lower f_l frequencies

$$\begin{aligned}f_u &= \frac{c}{d \max_{\varphi,\varphi'} |\cos\varphi - \cos\varphi'|} \\ f_l &= \frac{f_u}{N}\end{aligned} \tag{10.69}$$

with d being the sensor spacing, φ' the steering angle measured with respect to the axis of the array, and φ the direction of the source. For a desired range of $\pm 30^\circ$ and five sensors spaced 5 cm apart, the range is approximately 880 to 4400 Hz. We see in Figure 10.23 that at very low frequencies the response is essentially omnidirectional, since the microphone spacing is small compared to the large wavelength. At high frequencies more lobes start appearing, and the array steers toward not only the preferred direction but others as well. For speech signals, the upshot is that we either need a lot of microphones to provide a directional polar pattern at low frequencies, or we need them to be spread far enough apart, or both.

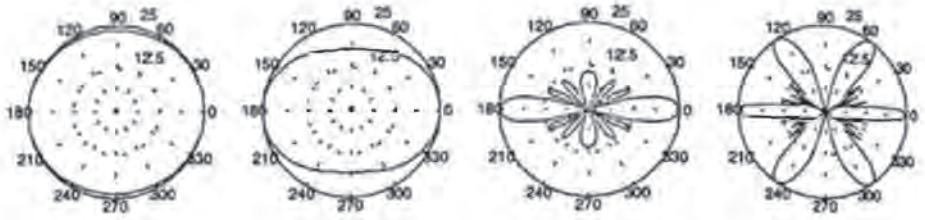


Figure 10.23 Polar pattern of a microphone array with steering angle of $\varphi' = 0$, five microphones spaced 5 cm apart for 400, 880, 4400, and 8000 Hz from left to right, respectively, for a source located at 5 m.

The polar pattern in Figure 10.23 was computed as follows:

$$P(f, r, \varphi) = \sum_{i=1}^{N} \frac{e^{-j2\pi f\left[a_i \sin\varphi' + \left|re^{j\varphi} - ja_i\right|\right]/c}}{\left|re^{j\varphi} - ja_i\right|} \tag{10.70}$$

though the sensors could be spaced nonuniformly, as in Figure 10.24, allowing for better behavior across the frequency spectrum.

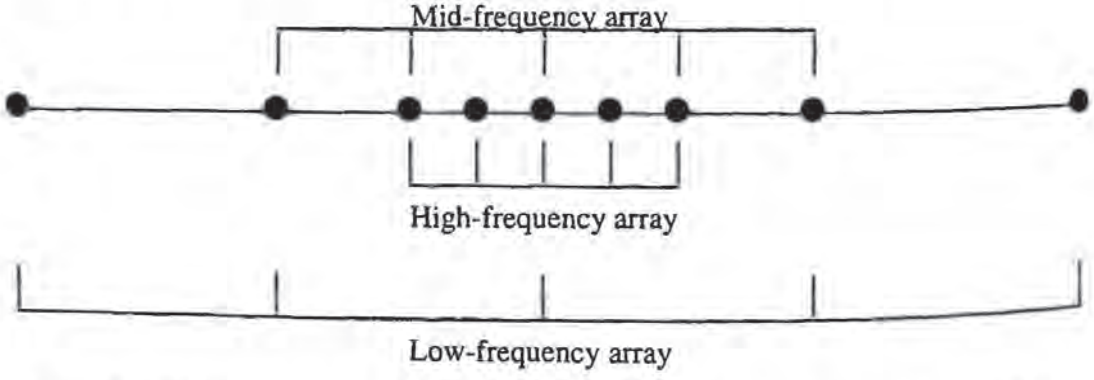


Figure 10.24 Nonuniform linear microphone array containing three subarrays for the high, mid, and low frequencies.

Once a microphone array has been steered towars a direction φ', it attenuates noise source coming from other directions. The beamwidth depends not only on the frequency of the signal, but also on the steering direction. If the beam is steered toward a direction φ', then the direction of the source for which the beam response fall to half its power has been found empirically to be

$$\varphi_{3dB}(f) = \cos^{-1}\left\{\cos\varphi' \pm \frac{K}{Ndf}\right\} \tag{10.71}$$

with K being a constant. Equation (10.71) shows that the smaller the array, the wider the beam, and that lower frequencies yield wider beams also. Figure 10.25 shows that the bandwidth of the array when steering toward a 30° direction is lower than when steering at 0°.

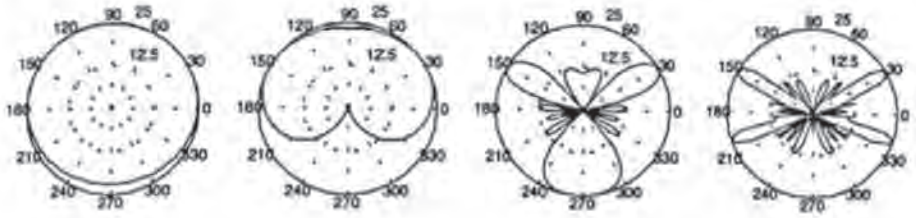


Figure 10.25 Polar pattern of a microphone array with steering angle of $\varphi' = 30°$, five microphones spaced 5 cm apart for 400, 880, 3000, and 4400 Hz from left to right, respectively, for a source located at 5 m.

Microphone arrays have been shown to improve recognition accuracy when the microphones and the speaker are far apart [51]. Several companies are commercializing microphone arrays for teleconferencing or speech recognition applications.

Only in anechoic chambers does the assumption in Eq. (10.61) hold, since in practice many reflections take place, which are also different for different microphones. In addition, the assumption of a common direction of arrival for all microphones may not hold either. For this case of reverberant environments, single beamformers typically fail. While computing the direction of arrival is much more difficult in this case, the SNR can still be improved.

Let's define the desired signal $d[n]$ as that obtained in the reference microphone. We can estimate the vector $\mathbf{H}[n] = \{h_{11},\cdots,h_{1L},h_{21},\cdots,h_{2L},\cdots,h_{(N-1)1},\cdots,h_{(N-1)L}\}$ for the $(N\text{-}1)$ L-tap filters that minimizes the error array [25]

$$e[n] = d[n] - \mathbf{H}[n]\mathbf{Y}[n] \tag{10.72}$$

where the $(N-1)$ microphone signals are represented in the vector

$$\mathbf{Y}[n] = \{y_1[n],\cdots,y_1[n-L-1],y_2[n],\cdots,y_2[n-L-1],\cdots,y_{N-1}[n],\cdots,y_{N-1}[n-L-1]\}$$

The filter coefficients $\mathbf{G}[n]$ can be estimated through the adaptive filtering techniques described in Section 10.3. The clean signal is then estimated as

$$\hat{x}[n] = \frac{1}{2}\left(d[n] + \mathbf{H}[n]\mathbf{Y}[n]\right) \tag{10.73}$$

This last method does not assume anything about the geometry of the microphone array.

10.4.2. Blind Source Separation

The problem of separating the desired speech from interfering sources, the *cocktail party effect* [15], has been one of the holy grails in signal processing. *Blind source separation* (BSS) is a set of techniques that assume no information about the mixing process or the sources, apart from their mutual statistical independence, hence is termed blind. *Independent component analysis* (ICA), developed in the last few years [19, 38], is a set of techniques to solve the BSS problem that estimate a set of linear filters to separate the mixed signals under the assumption that the original sources are statistically independent.

Let's first consider *instantaneous mixing*. Let's assume that R microphone signals $y_i[n]$, denoted by $\mathbf{y}[n] = \left(y_1[n], y_2[n], \cdots, y_R[n]\right)$, are obtained by a linear combination of R unobserved source signals $x_i[n]$, denoted by $\mathbf{x}[n] = \left(x_1[n], x_2[n], \cdots, x_R[n]\right)$:

$$\mathbf{y}[n] = \mathbf{G}\mathbf{x}[n] \tag{10.74}$$

for all n, with $\mathbf{G}$ being the R x R mixing matrix. This mixing is termed instantaneous, since the sensor signals at time n depend on the sources at the same, but no earlier, time point. Had the mixing matrix been given, its inverse could have been applied to the sensor signals to recover the sources by $\mathbf{x}[n] = \mathbf{G}^{-1}\mathbf{y}[n]$. In the absence of any information about the mixing, the blind separation problem consists of estimating a separating matrix $\mathbf{H} = \mathbf{G}^{-1}$ from the observed microphone signals alone. The source signals can then be recovered by

$$\mathbf{x}[n] = \mathbf{H}\mathbf{y}[n] \tag{10.75}$$

We'll use here the probabilistic formulation of ICA, though alternate frameworks for ICA have been derived also [18]. Let $p_{\mathbf{x}}(\mathbf{x}[n])$ be the probability density function (pdf) of the source signals, so that the pdf of microphone signals $\mathbf{y}[n]$ is given by

$$p_{\mathbf{y}}(\mathbf{y}[n]) = \mid \mathbf{H} \mid p_{\mathbf{x}}(\mathbf{H}\mathbf{y}[n]) \tag{10.76}$$

and if we furthermore assume the sources $\mathbf{x}[n]$ are independent from themselves in time, $\mathbf{x}[n+i]$ $i \neq 0$, then the joint probability is given by

$$p_{\mathbf{y}}(\mathbf{y}[0], \mathbf{y}[1], \cdots, \mathbf{y}[N-1]) = \prod_{n=0}^{N-1} p_{\mathbf{y}}(\mathbf{y}[n]) = \mid \mathbf{H} \mid^{N} \prod_{n=0}^{N-1} p_{\mathbf{x}}(\mathbf{H}\mathbf{y}[n]) \tag{10.77}$$

whose normalized log-likelihood is given by

$$\Psi = \frac{1}{N}\ln p_{\mathbf{y}}(\mathbf{y}[0],\mathbf{y}[1],\cdots,\mathbf{y}[N-1]) = \ln|\mathbf{H}| + \frac{1}{N}\sum_{n=0}^{N-1}\ln p_{\mathbf{x}}(\mathbf{H}\mathbf{y}[n]) \tag{10.78}$$

It can be shown that

$$\frac{\partial \ln|\mathbf{H}|}{\partial \mathbf{H}} = \left(\mathbf{H}^T\right)^{-1} \tag{10.79}$$

so that that the gradient of Ψ [38] in Eq. (10.78) is given by

$$\frac{\partial \Psi}{\partial \mathbf{H}} = \left(\mathbf{H}^T\right)^{-1} + \frac{1}{N}\sum_{n=0}^{N-1}\phi(\mathbf{H}\mathbf{y}[n])\left(\mathbf{y}[n]\right)^T \tag{10.80}$$

where $\phi(\mathbf{x})$ is expressed as

$$\phi(\mathbf{x}) = \frac{\partial \ln p_{\mathbf{x}}(\mathbf{x})}{\partial \mathbf{x}} \tag{10.81}$$

If we further assume the distribution is a zero mean Gaussian distribution with standard deviation σ, then Eq. (10.81) results in

$$\phi(\mathbf{x}) = -\frac{\mathbf{x}}{\sigma^2} \tag{10.82}$$

which inserted into Eq. (10.80) yields

$$\frac{\partial \Psi}{\partial \mathbf{H}} = \left(\mathbf{H}^T\right)^{-1} - \frac{\mathbf{H}}{\sigma^2}\left(\frac{1}{N}\sum_{n=0}^{N-1}\mathbf{y}[n]\left(\mathbf{y}[n]\right)^T\right) = \left(\mathbf{H}^T\right)^{-1} - \frac{\mathbf{H}}{\sigma^2}\mathbf{R} \tag{10.83}$$

with $\mathbf{R}$ being the matrix of cross-correlations, i.e.,

$$R_{ij} = \frac{1}{N}\sum_{n=0}^{N-1} y_i[n]y_j[n] \tag{10.84}$$

Setting Eq. (10.83) to $\mathbf{0}$ results in maximization of Eq. (10.78) under the Gaussian assumption:

$$\mathbf{H}^T\mathbf{H} = \sigma^2\mathbf{R}^{-1} \tag{10.85}$$

which can be solved using the Cholesky decomposition described in Chapter 6.

Since σ is generally not known, it can be shown from Eq. (10.85) that the sources can be recovered only to within a scaling factor [17]. Scaling is in general not a big problem, since speech recognition systems perform automatic gain control (AGC). Moreover, the sources can be recovered to within a permutation. To see this, let's define a two-dimensional matrix A

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{10.86}$$

which is orthogonal:

$$\mathbf{A}^T\mathbf{A} = \mathbf{I} \tag{10.87}$$

If $\mathbf{H}$ is a solution of Eq. (10.85), then $\mathbf{AH}$ is also a solution. Thus, a permutation of the sources yields the same correlation matrix in Eq. (10.84). Although we have shown it only under the Gaussian assumption, separation up to a scaling factor and source permutation is a general result in blind source separation [17].

Unfortunately, the Gaussian assumption does not guarantee separation. To see this, we can define a two-dimensional rotation matrix $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{10.88}$$

which is also orthogonal, so that if $\mathbf{H}$ is a solution of Eq. (10.85), then $\mathbf{AH}$ is also a solution.

The Gaussian assumption entails considering only second-order statistics, and to ensure separation we could consider higher-order statistics. Since speech signals do not follow a Gaussian distribution, we could use a Laplacian distribution, as we saw in Chapter 7:

$$p_x(x) = \frac{\beta}{2} e^{-\beta|x|} \tag{10.89}$$

which, using Eq. (10.81), results in

$$\phi(x) = \begin{cases} -\beta & x > 0 \\ \beta & x < 0 \end{cases} \tag{10.90}$$

and thus a nonlinear function of $\mathbf{H}$ for Eq. (10.80). Since a closed-form solution is not possible, a common solution in this case is gradient descent, where the gradient is given by

$$\frac{\partial \Psi}{\partial \mathbf{H}_n} = \left(\mathbf{H}_n^T\right)^{-1} + \phi(\mathbf{H}_n\mathbf{y}[n])(\mathbf{y}[n])^T \tag{10.91}$$

and the update formula by

$$\mathbf{H}_{n+1} = \mathbf{H}_n - \varepsilon \frac{\partial \Psi}{\partial \mathbf{H}_n} = \mathbf{H}_n - \varepsilon\left[\left(\mathbf{H}_n^T\right)^{-1} + \phi(\mathbf{H}_n\mathbf{y}[n])(\mathbf{y}[n])^T\right] \tag{10.92}$$

which is the so-called *infomax* rule [10].

Often the nonlinearity in Eq. (10.90) is replaced by a sigmoid[6] function:

$$\phi(x) = -\beta \tanh(\beta x) \tag{10.93}$$

which implies a density function

$$p_x(x) = \frac{\beta}{2\pi \cosh(\beta x)} \tag{10.94}$$

The sigmoid converges to the Laplacian as $\beta \to \infty$. Nonlinear functions in Eqs. (10.90) and (10.93) can be expanded in Taylor series so that all the moments of the observed signals are used and not just the second order, as in the case of the Gaussian assumption. These nonlinearities have been shown to be more effective in separating the sources. The use of more accurate density functions for $p_x(\mathbf{x})$, such as a mixture of Gaussians [9], also results in nonlinear $\phi(\mathbf{x})$ functions that have shown better separation.

A problem of Eq. (10.92) is that it requires a matrix inversion at every iteration. The so-called *natural gradient* [7] was suggested to avoid this, also providing faster convergence. To do this we can multiply the gradient of Eq. (10.91) by a positive definite matrix, the inverse of the Fisher's information matrix $\mathbf{H}_n^T\mathbf{H}_n$, for example, to whiten the signal:

$$\mathbf{H}_{n+1} = \mathbf{H}_n - \varepsilon \frac{\partial \Psi}{\partial \mathbf{H}} \mathbf{H}_n^T \mathbf{H}_n \tag{10.95}$$

which, combined with Eq. (10.91), results in

$$\mathbf{H}_{n+1} = \mathbf{H}_n - \varepsilon \left[\mathbf{I} + \phi(\hat{\mathbf{x}}[n])(\hat{\mathbf{x}}[n])^T \right] \mathbf{H}_n \tag{10.96}$$

where the estimated sources are given by

$$\hat{\mathbf{x}}[n] = \mathbf{H}_n \mathbf{y}[n] \tag{10.97}$$

Notice the similarity of this approach to the RLS algorithm of Section 10.3.5. Similarly to most Newton-Raphson methods, the convergence of this approach is quadratic instead of linear as long as we are close enough to the maximum.

Another way of overcoming the lack of separation under the independent Gaussian assumption is to make use of temporal information, which we know is important for speech signals. If the model of Eq. (10.74) is extended to contain additive noise

$$\mathbf{y}[n] = \mathbf{G}\mathbf{x}[n] + \mathbf{v}[n] \tag{10.98}$$

[6] The sigmoid function can be expressed in terms of the hyperbolic tangent $\tanh(x) = \sinh(x)/\cosh(x)$, where $\sinh(x) = (e^x - e^{-x})/2$ and $\cosh(x) = (e^x + e^{-x})/2$.

we can compute the autocorrelation of $\mathbf{y}[n]$ as

$$\mathbf{R}_y[n]=\mathbf{G}\mathbf{R}_x[n]\mathbf{G}^T+\mathbf{R}_v[n] \tag{10.99}$$

or, after some manipulation,

$$\mathbf{R}_x[n]=\mathbf{H}(\mathbf{R}_y[n]-\mathbf{R}_v[n])\mathbf{H}^T \tag{10.100}$$

which we know must be diagonal because the sources $\mathbf{x}$ are independent, and thus $\mathbf{H}$ can be estimated to minimize the squared error of the off-diagonal terms of $\mathbf{R}_x[n]$ for several values of n [11]. Equation (10.100) is a generalization of Eq. (10.85) when considering temporal correlation and additive noise.

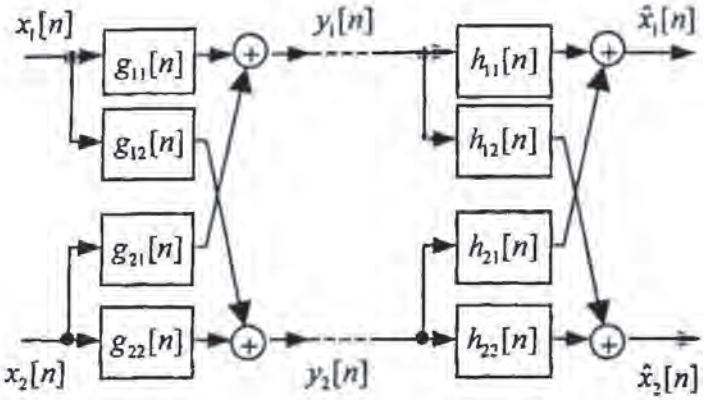


Figure 10.26 Convolutional model for the case of two microphones.

The case of instantaneous mixing is not realistic, as we need to consider the transfer functions between the sources and the microphones created by the room acoustics. It can be shown that the reconstruction filters $h_{ij}[n]$ in Figure 10.26 will completely recover the original signals $x_i[n]$ *if and only if* their z-transforms are the inverse of the z-transforms of the mixing filters $g_{ij}[n]$:

$$\begin{aligned}\begin{pmatrix} H_{11}(z) & H_{12}(z) \\ H_{21}(z) & H_{22}(z) \end{pmatrix} &= \begin{pmatrix} G_{11}(z) & G_{12}(z) \\ G_{21}(z) & G_{22}(z) \end{pmatrix}^{-1} \\ &= \frac{1}{G_{11}(z)G_{22}(z)-G_{12}(z)G_{21}(z)} \begin{pmatrix} G_{11}(z) & G_{12}(z) \\ G_{21}(z) & G_{22}(z) \end{pmatrix}\end{aligned} \tag{10.101}$$

If the matrix in Eq. (10.101) is not invertible, separability is impossible. This can happen if both microphones pick up the same signal, which could happen if either the two mi-

crophones are too close to each other or the two sources are too close to each other. It's reasonable to assume the mixing filters $g_{ij}[n]$ to be FIR filters, whose length will generally depend on the reverberation time, which in turn depends on the room size, microphone position, wall absorbance, and so on. In general this means that the reconstruction filters $h_{ij}[n]$ have an infinite impulse response. In addition, the filters $g_{ij}[n]$ may have zeroes outside the unit circle, so that perfect reconstruction filters would need to have poles outside the unit circle. For this reason it is not possible, in general, to recover the original signals exactly.

In practice, it's convenient to assume such filters to be FIR of length q, which means that the original signals $x_1[n]$ and $x_2[n]$, will not be recovered exactly. Thus the problem consists in estimating the reconstruction filters $h_{ij}[n]$ directly from the microphone signals $y_1[n]$ and $y_2[n]$, so that the estimated signals $\hat{x}_i[n]$ are as close as possible to the original signals. Often we are satisfied if the resulting signals are separated, even if they contain some amount of reverberation.

An approach commonly used to combat this problem consists of taking a filterbank and assuming instantaneous mixing within each filter [38]. This approach can separate real sources much more effectively, but it suffers from the problem of permutations, which in this case is more severe because frequencies from different sources can be mixed together. To avoid this, we may need a probabilistic model of the sources that takes into account correlations across frequencies [3]. Another problem occurs when the number of sources is larger than the number of microphones.

10.5. ENVIRONMENT COMPENSATION PREPROCESSING

The goal of this section is to present a number of techniques used to *clean up* the signal of additive noise and/or channel distortions prior to the speech recognition system. Although the techniques presented here are developed for the case of one microphone, they can be generalized to the case where several microphones are available using the approaches described in Section 10.4. These techniques can also be used to *enhance* the signal captured with a speakerphone or a desktop microphone in teleconferencing applications.

Since the use of human auditory system is so robust to changes in acoustical environment, many researchers have attempted to develop signal processing schemes that mimic the functional organization of the peripheral auditory system [27, 49]. The PLP cepstrum described in Chapter 6 has also been shown to be very effective in combating noise and channel distortions [60].

Another alternative is to consider the feature vector as an integral part of the recognizer, and thus researchers have investigated its design so as to maximize recognition accuracy, as discussed in Chapter 9. Such approaches include LDA [34] and neural networks [45]. These discriminatively trained features can also be optimized to operate better under noisy conditions, thus possibly beating the standard mel-cepstrum, especially when several independent features are combined [50]. The mel-cepstrum is the most popular feature vector for speech recognition. In this context we present a number of techniques that have been proposed over the years to compensate for the effects of additive noise and channel distortions on the cepstrum.

10.5.1. Spectral Subtraction

The basic assumption in this section is that the desired clean signal $x[m]$ has been corrupted by additive noise $n[m]$:

$$y[m] = x[m] + n[m] \tag{10.102}$$

and that both $x[m]$ and $n[m]$ are statistically independent, so that the power spectrum of the output $y[m]$ can be approximated as the sum of the power spectra:

$$|Y(f)|^2 \approx |X(f)|^2 + |N(f)|^2 \tag{10.103}$$

with equality if we take expected values, as the expected value of the cross term vanishes (see Section 10.1.3).

Although we don't know $|N(f)|^2$, we can obtain an estimate using the average periodogram over M frames that are known to be just noise (i.e., when no signal is present) as long as the noise is stationary

$$|\hat{N}(f)|^2 = \frac{1}{M}\sum_{i=0}^{M-1}|Y_i(f)|^2 \tag{10.104}$$

Spectral subtraction supplies an intuitive estimate for $|X(f)|$ using Eqs. (10.103) and (10.104) as

$$|\hat{X}(f)|^2 = |Y(f)|^2 - |\hat{N}(f)|^2 = |Y(f)|^2\left(1 - \frac{1}{SNR(f)}\right) \tag{10.105}$$

where we have defined the frequency-dependent signal-to-noise ratio $SNR(f)$ as

$$SNR(f) = \frac{|Y(f)|^2}{|\hat{N}(f)|^2} \tag{10.106}$$

Equation (10.105) describes the magnitude of the Fourier transform but not the phase. This is not a problem if we are interested in computing the mel-cepstrum as discussed in Chapter 6. We can just modify the magnitude and keep the original phase of $Y(f)$ using a filter $H_{ss}(f)$:

$$\hat{X}(f) = Y(f)H_{ss}(f) \tag{10.107}$$

where, according to Eq. (10.105), $H_{ss}(f)$ is given by

$$H_{ss}(f) = \sqrt{1 - \frac{1}{SNR(f)}} \tag{10.108}$$

Since $\left|\hat{X}(f)\right|^2$ is a power spectral density, it has to be positive, and therefore

$$SNR(f) \geq 1 \tag{10.109}$$

but we have no guarantee that $SNR(f)$, as computed by Eq. (10.106), satisfies Eq. (10.109). In fact, it is easy to see that noise frames do not comply. To enforce this constraint, Boll [13] suggested modifying Eq. (10.108) as follows:

$$H_{ss}(f) = \sqrt{\max\left(1 - \frac{1}{SNR(f)}, a\right)} \tag{10.110}$$

with $a \geq 0$, so that the quantity within the square root is always positive, and where $f_{ss}(x)$ is given by

$$f_{ss}(x) = \sqrt{\max\left(1 - \frac{1}{x}, a\right)} \tag{10.111}$$

It is useful to express $SNR(f)$ in dB so that

$$\overline{x} = 10 \log_{10} SNR \tag{10.112}$$

and the gain of the filter in Eq. (10.111) also in dB:

$$g_{ss}(\overline{x}) = 20 \log_{10} f_{ss}(\overline{x}) \tag{10.113}$$

Using Eqs. (10.111) and (10.112), we can express Eq. (10.113) by

$$g_{ss}(\overline{x}) = \max\left(10 \log_{10}\left(1 - 10^{-\overline{x}/10}\right), -A\right) \tag{10.114}$$

after expressing the attenuation a in dB:

$$a = 10^{-A/10} \tag{10.115}$$

Equation (10.114) is plotted in Figure 10.27 for A = 10 dB.

The spectral subtraction rule in Eq. (10.111) is quite intuitive. To implement it we can do a short-time analysis, as shown in Chapter 6, by using overlapping windowed segments, zero-padding, computing the FFT, modifying the magnitude spectrum, taking the inverse FFT, and adding the resulting windows.

This implementation results in output speech that has significantly less noise, though it exhibits what is called *musical noise* [12]. This is caused by frequency bands f for which $|Y(f)|^2 \approx \left|\hat{N}(f)\right|^2$. As shown in Figure 10.27, a frequency f_0 for which $|Y(f_0)|^2 < \left|\hat{N}(f_0)\right|^2$ is attenuated by A dB, whereas a neighboring frequency f_1, where $|Y(f_1)|^2 > \left|\hat{N}(f_1)\right|^2$, has a much smaller attenuation. These rapid changes with frequency introduce tones at varying frequencies that appear and disappear rapidly.

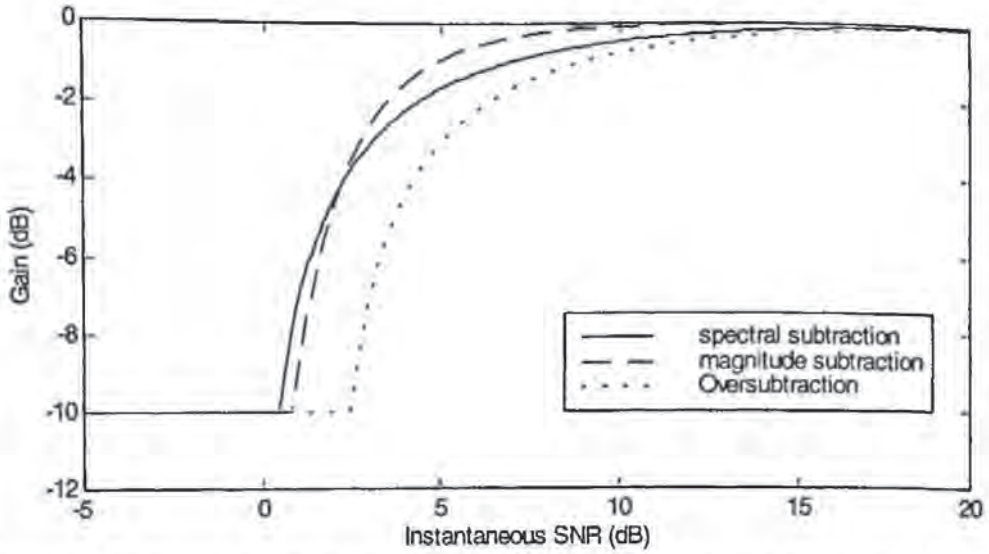


Figure 10.27 Magnitude of the spectral subtraction filter gain as a function of the input instantaneous SNR for A = 10 dB, for the spectral subtraction of Eq. (10.114), magnitude subtraction of Eq. (10.118), and oversubtraction of Eq. (10.119) with β = 2 dB.

The main reason for the presence of musical noise is that the estimates of $SNR(f)$ through Eqs. (10.104) and (10.106) are poor. This is partly because $SNR(f)$ is computed independently for each frequency, whereas we know that $SNR(f_0)$ and $SNR(f_1)$ are correlated if f_0 and f_1 are close to each other. Thus, one possibility is to smooth the filter in Eq. (10.114) over frequency. This approach suppresses a smaller amount of noise, but it does not distort the signal as much, and thus may be preferred by listeners. Similarly, smoothing over time

$$SNR(f,t)=\gamma SNR(f,t-1)+(1-\gamma)\frac{|Y(f)|^2}{|\hat{N}(f)|^2} \tag{10.116}$$

can also be done to reduce the distortion, at the expense of a smaller noise attenuation. Smoothing over both time and frequency can be done to obtain more accurate SNR measurements and thus less distortion. As shown in Figure 10.28, use of spectral subtraction can reduce the error rate.

Additionally, the attenuation A can be made a function of frequency. This is useful when we want to suppress more noise at one frequency than another, which is a tradeoff between noise reduction and nonlinear distortion of speech.

Other enhancements to the basic algorithm have been proposed to reduce the musical noise. Sometimes Eq. (10.111) is generalized to

$$f_{ms}(x)=\left(\max\left(1-\frac{1}{x^{\alpha/2}},a\right)\right)^{1/\alpha} \tag{10.117}$$

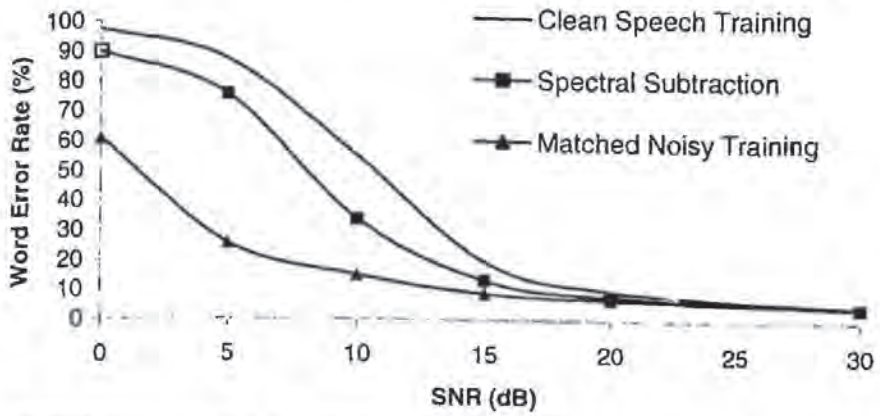


Figure 10.28 Word error rate as a function of SNR (dB) using Whisper on the *Wall Street Journal* 5000-word dictation task. White noise was added at different SNRs. The solid line represents the baseline system trained with clean speech, the line with squares the use of spectral subtraction with the previous clean HMMs. They are compared to a system trained on the same speech with the same SNR as the speech tested on.

where $\alpha = 2$ corresponds to the *power spectral subtraction* rule in Eq. (10.111), and $\alpha = 1$ corresponds to the *magnitude subtraction* rule (plotted in Figure 10.27 for A = 10 dB):

$$g_{ms}(\bar{x}) = \max\left(20\log_{10}\left(1-10^{-\bar{x}/5}\right), -A\right) \tag{10.118}$$

Another variation, called oversubtraction, consists of multiplying the estimate of the noise power spectral density $|\hat{N}(f)|^2$ in Eq. (10.104) by a constant $10^{\beta/10}$, where $\beta > 0$, which causes the power spectral subtraction rule in Eq. (10.114) to be transformed to another function

$$g_{ms}(\bar{x}) = \max\left(10\log_{10}\left(1-10^{-(\bar{x}-\beta)/10}\right), -A\right) \tag{10.119}$$

This causes $|Y(f)|^2 < |\hat{N}(f)|^2$ to occur more often than $|Y(f)|^2 > |\hat{N}(f)|^2$ for frames for which $|Y(f)|^2 \approx |\hat{N}(f)|^2$, and thus reduces the musical noise.

10.5.2. Frequency-Domain MMSE from Stereo Data

You have seen that several possible functions, such as Eqs. (10.114), (10.118), or (10.119), can be used to attenuate the noise, and it is not clear that any one of them is better than the others, since each has been obtained through different assumptions. This opens the possibility of estimating the curve $g(\bar{x})$ using a different criterion, and, thus, different approximations than those used in Section 10.5.1.

One interesting possibility occurs when we have pairs of stereo utterances that have been recorded simultaneously in noise-free conditions in one channel and noisy conditions

in the other channel. In this case, we can estimate $f(x)$ using a minimum mean squared criterion (Porter and Boll [47], Ephraim and Malah [23]), so that

$$\hat{f}(x) = \arg\min_{f(x)} \left\{ \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left(X_i(f_j) - f\left(SNR(f_j)\right) Y_i(f_j) \right)^2 \right\} \tag{10.120}$$

or $g(x)$ as

$$\hat{g}(x) = \arg\min_{g(x)} \left\{ \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left(10\log_{10}\left|X_i(f_j)\right|^2 - g\left(SNR(f_j)\right) - 10\log_{10}\left|Y_i(f_j)\right|^2 \right)^2 \right\} \tag{10.121}$$

which can be solved by discretizing $f(x)$ and $g(x)$ into several bins and summing over all M frequencies and N frames. This approach results in a curve that is smoother and thus offers less musical noise and lower distortion. Stereo utterances of noise-free and noisy speech are needed to estimate $f(x)$ and $g(x)$ through Eqs. (10.120) and (10.121) for any given acoustical environment and can be collected with two microphones, or the noisy speech can be obtained by adding to the clean speech artificial noise from the testing environment.

Another generalization of this approach is to use a different function $f(x)$ or $g(x)$ for every frequency [2] as shown in Figure 10.29. This also allows for a lower squared error at the expense of having to store more data tables. In the experiments of Figure 10.29, we note that more subtraction is needed at lower frequencies than at higher frequencies in this case.

If such stereo data is available to estimate these curves, it makes the enhanced speech sound better [23] than does spectral subtraction. When used in speech recognition systems, it also leads to higher accuracies [2].

10.5.3. Wiener Filtering

Let's reformulate Eq. (10.102) from the statistical point of view. The process $\mathbf{y}[n]$ is the sum of random process $\mathbf{x}[n]$ and the additive noise $\mathbf{v}[n]$ process:

$$\mathbf{y}[n] = \mathbf{x}[n] + \mathbf{v}[n] \tag{10.122}$$

We wish to find a linear estimate $\hat{\mathbf{x}}[n]$ in terms of the process $\mathbf{y}[n]$:

$$\hat{\mathbf{x}}[n] = \sum_{m=-\infty}^{\infty} h[m]\mathbf{y}[n-m] \tag{10.123}$$

which is the result of a linear time-invariant filtering operation. The MMSE estimate of $h[n]$ in Eq. (10.123) minimizes the squared error

$$E\left\{ \left[\mathbf{x}[n] - \sum_{m=-\infty}^{\infty} h[m]\mathbf{y}[n-m] \right]^2 \right\} \tag{10.124}$$

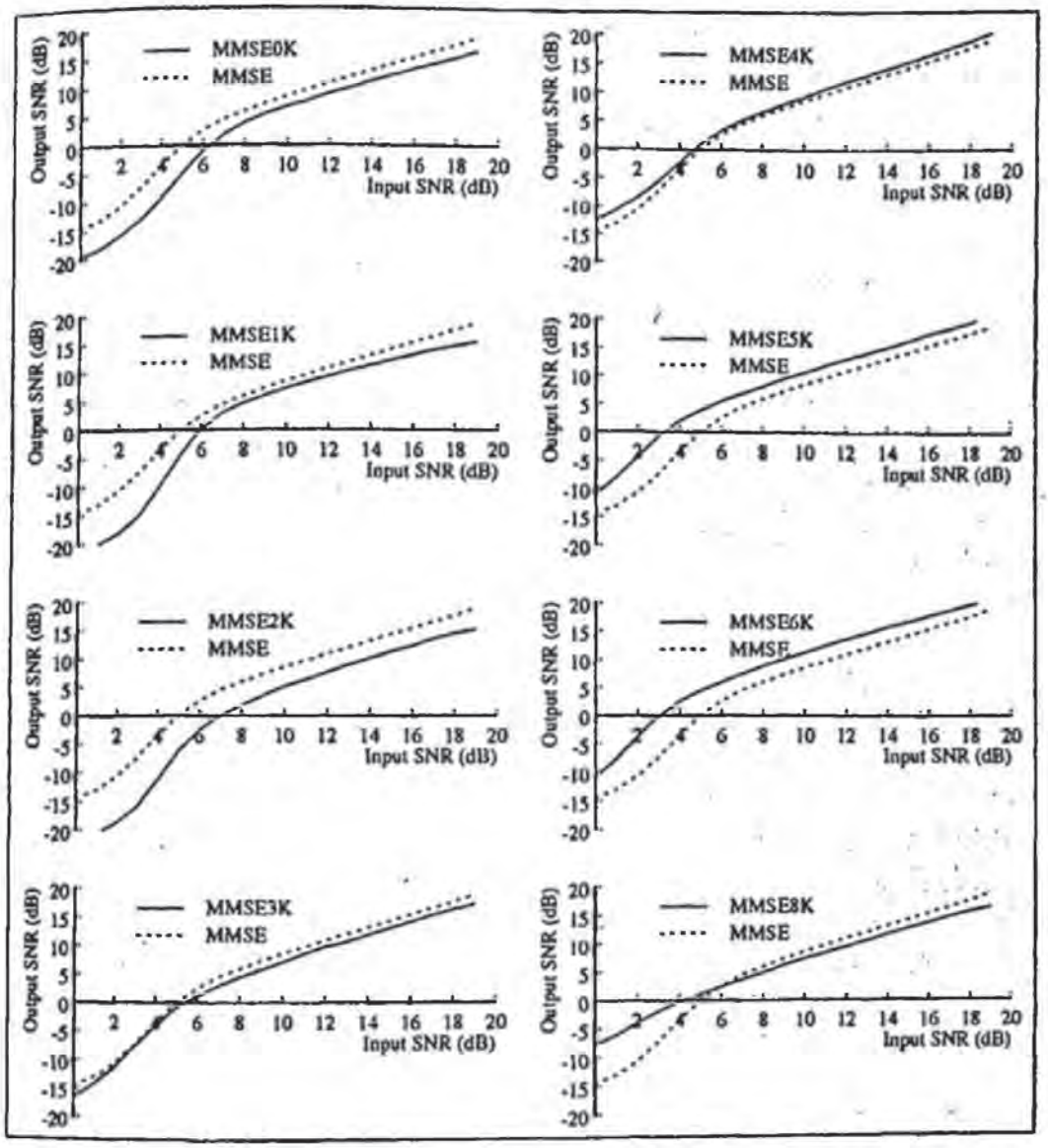


Figure 10.29 Empirical curves for input-to-output instantaneous SNR. Eight different curves for 0, 1, 2, 3, 4, 5, 6, 7 and 8 kHz are obtained following Eq. (10.121) [2] using speech recorded simultaneously from a close-talking microphone and a desktop microphone.

which results in the famous *Wiener-Hopf* equation

$$R_{sy}[l] = \sum_{m=-\infty}^{\infty} h[m] R_{yy}[l-m] \tag{10.125}$$

so that, taking Fourier transforms, the resulting filter can be expressed in the frequency domain as

$$H(f) = \frac{S_{xy}(f)}{S_{yy}(f)} \tag{10.126}$$

If the signal $\mathbf{x}[n]$ and the noise $\mathbf{v}[n]$ are orthogonal, which is often the case, then

$$S_{xy}(f) = S_{xx}(f) \text{ and } S_{yy}(f) = S_{xx}(f) + S_{vv}(f) \tag{10.127}$$

so that Eq. (10.126) is given by

$$H(f) = \frac{S_{xx}(f)}{S_{xx}(f) + S_{vv}(f)} \tag{10.128}$$

Equation (10.128) is called the *noncausal Wiener filter*. This can be realized only if we know the power spectra of both the noise and the signal. Of course, if $S_{xx}(f)$ and $S_{vv}(f)$ do not overlap, then $H(f) = 1$ in the band of the signal and $H(f) = 0$ in the band of the noise.

In practice, $S_{xx}(f)$ is unknown. If it were known, we could compute its mel-cepstrum, which would coincide exactly with the mel-cepstrum before noise addition. To solve this chicken-and-egg problem, we need some kind of model. Ephraim [22] proposed the use of an HMM where, if we know what state the current frame falls under, we can use its mean spectrum as $S_{xx}(f)$. In practice we do not know what state each frame falls into either, so he proposed to weigh the filters for each state by the a posterior probability that the frame falls into each state. This algorithm, when used in speech enhancement, results in gains of 7 dB or more.

A causal version of the Wiener filter can also be derived. A dynamical state model algorithm called the Kalman filter (see [42] for details) is also an extension of the Wiener filter.

10.5.4. Cepstral Mean Normalization (CMN)

Different microphones have different transfer functions, and even the same microphone has a varying transfer function depending on the distance to the microphone and the room acoustics. In this section we describe a powerful and simple technique that is designed to handle convolutional distortions and, thus, increases the robustness of speech recognition systems to unknown linear filtering operations.

Given a signal $x[n]$, we compute its cepstrum through short-time analysis, resulting in a set of T cepstral vectors $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{T-1}\}$. Its sample mean $\bar{\mathbf{x}}$ is given by

$$\bar{\mathbf{x}} = \frac{1}{T}\sum_{t=0}^{T-1} \mathbf{x}_t \tag{10.129}$$

Cepstral mean normalization (CMN) (Atal [8]) consists of subtracting $\bar{\mathbf{x}}$ from each vector $\mathbf{x}_t$ to obtain the normalized cepstrum vector $\hat{\mathbf{x}}_t$:

$$\hat{\mathbf{x}}_t = \mathbf{x}_t - \bar{\mathbf{x}} \tag{10.130}$$

Let's now consider a signal $y[n]$, which is the output of passing $x[n]$ through a filter $h[n]$. We can compute another sequence of cepstrum vectors $\mathbf{Y} = \{\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{T-1}\}$. Now let's further define a vector $\mathbf{h}$ as

$$\mathbf{h} = \mathbf{C}\left(\ln|H(\omega_0)|^2 \quad \cdots \quad \ln|H(\omega_M)|^2\right) \tag{10.131}$$

where $\mathbf{C}$ is the DCT matrix. We can see that

$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{h} \tag{10.132}$$

and thus the sample mean $\bar{\mathbf{y}}_t$ equals

$$\bar{\mathbf{y}} = \frac{1}{T}\sum_{t=0}^{T-1}\mathbf{y}_t = \frac{1}{T}\sum_{t=0}^{T-1}(\mathbf{x}_t + \mathbf{h}) = \bar{\mathbf{x}} + \mathbf{h} \tag{10.133}$$

and its normalized cepstrum is given by

$$\hat{\mathbf{y}}_t = \mathbf{y}_t - \bar{\mathbf{y}}_t = \hat{\mathbf{x}}_t \tag{10.134}$$

which indicates that cepstral mean normalization is immune to linear filtering operations. This procedure is performed on every utterance for both training and testing. Intuitively, the mean vector $\bar{\mathbf{x}}$ conveys the spectral characteristics of the current microphone and room acoustics. In the limit, when $T \to \infty$ for each utterance, we should expect means from utterances from the same recording environment to be the same. Use of CMN to the cepstrum vectors does not modify the delta or delta-delta cepstrum.

Let's analyze the effect of CMN on a short utterance. Assume that our utterance contains a single phoneme, say /s/. The mean $\bar{\mathbf{x}}$ will be very similar to the frames in this phoneme, since /s/ is quite stationary. Thus, after normalization, $\hat{\mathbf{x}}_t \approx 0$. A similar result will happen for other fricatives, which means that it would be impossible to distinguish these ultrashort utterances, and the error rate will be very high. If the utterance contains more than one phoneme but is still short, this problem is not insurmountable, but the confusion among phonemes is still higher than if no CMN had been applied. Empirically, it has been found that this procedure does not degrade the recognition rate on utterances from the same acoustical environment, as long as they are longer than 2–4 seconds. Yet the method provides significant robustness against linear filtering operations. In fact, for telephone recordings, where each call has a different frequency response, the use of CMN has been shown to provide as much as 30% relative decrease in error rate. When a system is trained on one microphone and tested on another, CMN can provide significant robustness.

Interestingly enough, it has been found in practice that the error rate for utterances within the same environment is actually somewhat lower, too. This is surprising, given that

there is no mismatch in channel conditions. One explanation is that, even for the same microphone and room acoustics, the distance between the mouth and the microphone varies for different speakers, which causes slightly different transfer functions, as we studied in Section 10.2. In addition, the cepstral mean characterizes not only the channel transfer function, but also the average frequency response of different speakers. By removing the long-term speaker average, CMN can act as sort of speaker normalization.

One drawback of CMN is it does not discriminate silence and voice in computing the utterance mean. An extension to CMN consists in computing different means for noise and speech [5]:

$$\begin{aligned} \mathbf{h}^{(j+1)} &= \frac{1}{N_s}\sum_{t\in q_s}\mathbf{x}_t - \mathbf{m}_s \\ \mathbf{n}^{(j+1)} &= \frac{1}{N_n}\sum_{t\in q_n}\mathbf{x}_t - \mathbf{m}_n \end{aligned} \tag{10.135}$$

i.e., the difference between the average vector for speech frames in the utterance and the average vector $\mathbf{m}_s$ for speech frames in the training data, and similarly for the noise frames $\mathbf{m}_n$. Speech/noise discrimination could be done by classifying frames into speech frames and noise frames, computing the average cepstra for each, and subtracting them from the average in the training data. This procedure works well as long as the speech/noise classification is accurate. It's best done by the recognizer, since other speech detection algorithms can fail in high background noise (see Section 10.6.2). To avoid errors in transitions between speech and noise, delta and delta-delta can be computed prior to this speech/noise mean normalization so that they are unaffected. As shown in Figure 10.30, this algorithm has been shown to improve robustness not only to varying channels but also to noise.

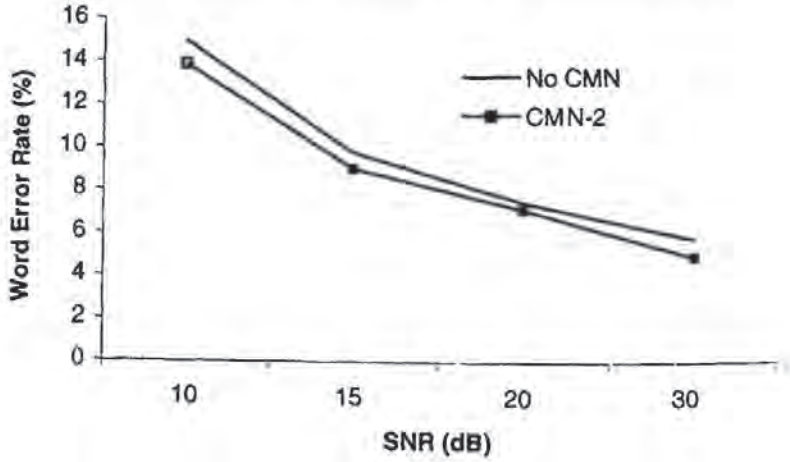


Figure 10.30 Word error rate as a function of SNR (dB) for both no CMN and CMN-2 [5]. White noise was added at different SNRs and the system was trained with speech with the same SNR. The Whisper system is used on the 5000-word *Wall Street Journal* task using a bigram language model.

10.5.5. Real-Time Cepstral Normalization

CMN requires the complete utterance to compute the cepstral mean; thus, it cannot be used in a real-time system, and an approximation needs to be used. In this section we discuss a modified version of CMN that can address this problem, as well as a set of techniques called RASTA that attempt to do the same thing.

We can interpret CMN as the operation of subtracting a low-pass filter $d[n]$, where all the T coefficients are identical and equal $1/T$, which is a high-pass filter with a cutoff frequency ω_c that is arbitrarily close to 0. This interpretation indicates that we can implement other types of high-pass filters. One that has been found to work well in practice is the exponential filter, so the cepstral mean $\bar{\mathbf{x}}_t$ is a function of time

$$\bar{\mathbf{x}}_t = \alpha \mathbf{x}_t + (1-\alpha)\bar{\mathbf{x}}_{t-1} \tag{10.136}$$

where α is chosen so that the filter has a time constant[7] of at least 5 seconds of speech.

Other types of filters have been proposed in the literature. In fact, a popular approach consists of an IIR bandpass filter with the transfer function:

$$H(z) = 0.1z^4 * \frac{2 + z^{-1} - z^{-3} - 2z^{-4}}{1 - 0.98z^{-1}} \tag{10.137}$$

which is used in the so-called relative spectral processing or RASTA [32]. As in CMN, the high-pass portion of the filter is expected to alleviate the effect of convolutional noise introduced in the channel. The low-pass filtering helps to smooth some of the fast frame-to-frame spectral changes present. Empirically, it has been shown that the RASTA filter behaves similarly to the real-time implementation of CMN, albeit with a slightly higher error rate. Both the RASTA filter and real-time implementations of CMN require the filter to be properly initialized. Otherwise, the first utterance may use an incorrect cepstral mean. The original derivation of RASTA includes a few stages prior to the bandpass filter, and this filter is performed on the spectral energies, not the cepstrum.

10.5.6. The Use of Gaussian Mixture Models

Algorithms such as spectral subtraction of Section 10.5.1 or the frequency-domain MMSE of Section 10.5.2 implicitly assume that different frequencies are uncorrelated from each other. Because of that, the spectrum of the enhanced signal may exhibit abrupt changes across frequency and not look like spectra of real speech signals. Using the model of the

[7] The time constant τ of a low-pass filter is defined as the value for which the output is cut in half. For an exponential filter of parameter α and sampling rate F_s, $\alpha = \ln 2/(TF_s)$.

environment of Section 10.1.3, we can express the clean-speech cepstral vector $\mathbf{x}$ as a function of the observed noisy cepstral vector $\mathbf{y}$ as

$$\mathbf{x} = \mathbf{y} - \mathbf{h} - \mathbf{C}\ln\left(1 - e^{\mathbf{C}^{-1}(\mathbf{n}-\mathbf{y})}\right) \tag{10.138}$$

where the noise cepstral vector $\mathbf{n}$ is a random vector. The MMSE estimate of $\mathbf{x}$ is given by

$$\hat{\mathbf{x}}_{MMSE} = E\{\mathbf{x} \mid \mathbf{y}\} = \mathbf{y} - \mathbf{h} - \mathbf{C}E\left\{\ln\left(1 - e^{\mathbf{C}^{-1}(\mathbf{n}-\mathbf{y})}\right) \mid \mathbf{y}\right\} \tag{10.139}$$

where the expectation uses the distribution of $\mathbf{n}$. Solution to Eq. (10.139) results in a nonlinear function which can be learned, for example, with a neural network [53].

A popular model to attack this problem consists in modeling the probability distribution of the noisy speech $\mathbf{y}$ as a mixture of K Gaussians:

$$p(\mathbf{y}) = \sum_{k=0}^{K-1} p(\mathbf{y} \mid k)P[k] = \sum_{k=0}^{K-1} \mathrm{N}(\mathbf{y}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)P[k] \tag{10.140}$$

where $P[k]$ is the prior probability of each Gaussian component k. If $\mathbf{x}$ and $\mathbf{y}$ are jointly Gaussian within class k, then $p(\mathbf{x} \mid \mathbf{y}, k)$ is also Gaussian [42] with mean:

$$E\{\mathbf{x} \mid \mathbf{y}, k\} = \boldsymbol{\mu}_{\mathbf{x}}^k + \boldsymbol{\Sigma}_{\mathbf{xy}}^k \left(\boldsymbol{\Sigma}_{\mathbf{y}}^k\right)^{-1} (\mathbf{y} - \boldsymbol{\mu}_{\mathbf{y}}^k) = \mathbf{C}_k\mathbf{y} + \mathbf{r}_k \tag{10.141}$$

so that the joint distribution of $\mathbf{x}$ and $\mathbf{y}$ is given by

$$\begin{aligned} p(\mathbf{x},\mathbf{y}) &= \sum_{k=0}^{K-1} p(\mathbf{x},\mathbf{y} \mid k)P[k] = \sum_{k=0}^{K-1} p(\mathbf{x} \mid \mathbf{y}, k)p(\mathbf{y} \mid k)P[k] \\ &= \sum_{k=0}^{K-1} \mathrm{N}(\mathbf{x}, \mathbf{C}_k\mathbf{y} + \mathbf{r}_k, \boldsymbol{\Gamma}_k)\mathrm{N}(\mathbf{y}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)P[k] \end{aligned} \tag{10.142}$$

where $\mathbf{r}_k$ is called the *correction vector*, $\mathbf{C}_k$ is the *rotation matrix*, and the matrix $\boldsymbol{\Gamma}_k$ tells us how uncertain we are about the compensation.

A maximum likelihood estimate of $\mathbf{x}$ maximizes the joint probability in Eq. (10.142). Assuming the Gaussians do not overlap very much (as in the FCDCN algorithm [2]):

$$\hat{\mathbf{x}}_{ML} \approx \arg\max_k p(\mathbf{x},\mathbf{y},k) = \arg\max_k \mathrm{N}(\mathbf{y}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\mathrm{N}(\mathbf{x}, \mathbf{C}_k\mathbf{y} + \mathbf{r}_k, \boldsymbol{\Gamma}_k)P[k] \tag{10.143}$$

whose solution is

$$\hat{\mathbf{x}}_{ML} = \mathbf{C}_{\hat{k}}\mathbf{y} + \mathbf{r}_{\hat{k}} \tag{10.144}$$

where

$$\hat{k} = \arg\max_{k} \mathrm{N}(\mathbf{y}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) P[k] \tag{10.145}$$

It is often more robust to compute the MMSE estimate of $\mathbf{x}$ (as in the CDCN [2] and RATZ [43] algorithms):

$$\hat{\mathbf{x}}_{MMSE} = E\{\mathbf{x} \mid \mathbf{y}\} = \sum_{k=0}^{K-1} p(k \mid \mathbf{y}) E\{\mathbf{x} \mid \mathbf{y}, k\} = \sum_{k=0}^{K-1} p(k \mid \mathbf{y})(\mathbf{C}_k \mathbf{y} + \mathbf{r}_k) \tag{10.146}$$

as a weighted sum for all mixture components, where the posterior probability $p(k \mid \mathbf{y})$ is given by

$$p(k \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid k) P[k]}{\sum_{k=0}^{K-1} p(\mathbf{y} \mid k) P[k]} \tag{10.147}$$

where the rotation matrix $\mathbf{C}_k$ in Eq. (10.144) can be replaced by $\mathbf{I}$ with a modest degradation in performance in return for faster computation [21].

A number of different algorithms [2, 43] have been proposed that vary in how the parameters $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, $\mathbf{r}_k$, and $\boldsymbol{\Gamma}_k$ are estimated. If stereo recordings are available from both the clean signal and the noisy signal, then we can estimate $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ by fitting a mixture Gaussian model to $\mathbf{y}$ as described in Chapter 3. Then $\mathbf{C}_k$, $\mathbf{r}_k$ and $\boldsymbol{\Gamma}_k$ can be estimated directly by linear regression of $\mathbf{x}$ and $\mathbf{y}$. The FCDCN algorithm [2, 6] is a variant of this approach when it is assumed that $\boldsymbol{\Sigma}_k = \sigma^2\mathbf{I}$, $\boldsymbol{\Gamma}_k = \gamma^2\mathbf{I}$, and $\mathbf{C}_k = \mathbf{I}$, so that $\boldsymbol{\mu}_k$ and $\mathbf{r}_k$ are estimated through a VQ procedure and $\mathbf{r}_k$ is the average difference $(\mathbf{y} - \mathbf{x})$ for vectors $\mathbf{y}$ that belong to mixture component k. An enhancement is to use the instantaneous SNR of a frame, defined as the difference between the log-energy of that frame and the average log-energy of the background noise. It is advantageous to use different correction vectors for different instantaneous SNR levels. The log-energy can be replaced by the zeroth-order cepstral coefficient with little change in recognition accuracy.

Often, stereo recordings are not available and we need other means of estimating parameters μ_k, Σ_k, $\mathbf{r}_k$, and Γ_k. CDCN [6] is one such algorithm that has a model of the environment as described in Section 10.1.3, which defines a nonlinear relationship between $\mathbf{x}$, $\mathbf{y}$ and the environmental parameters $\mathbf{n}$ and $\mathbf{h}$ for the noise and channel. This method also uses an MMSE approach where the correction vector is a weighted average of the correction vectors for all classes. An extension of CDCN using a vector Taylor series approximation [44] for that nonlinear function has been shown to offer improved results. Other methods that do not require stereo recordings or a model of the environment are presented in [43].

10.6. ENVIRONMENTAL MODEL ADAPTATION

We describe a number of techniques that achieve compensation by adapting the HMM to the noisy conditions. The most straightforward method is to retrain the whole HMM with the speech from the new acoustical environment. Another option is to apply standard adaptive techniques discussed in Chapter 9 to the case of environment adaptation. We consider a model of the environment that allows constrained adaptation methods for more efficient adaptation in comparison to the general techniques.

10.6.1. Retraining on Corrupted Speech

If there is a mismatch between acoustical environments, it is sensible to retrain the HMM. This is done in practice for telephone speech where only telephone speech, and no clean high-bandwidth speech, is used in the training phase.

Unfortunately, training a large-vocabulary speech recognizer requires a very large amount of data, which is often not available for a specific noisy condition. For example, it is difficult to collect a large amount of training data in a car driving at 50 mph, whereas it is much easier to record it at idle speed. Having a small amount of matched-conditions training data could be worse than a large amount of mismatched-conditions training data. Often we want to adapt our model given a relatively small sample of speech from the new acoustical environment.

One option is to take a noise waveform from the new environment, add it to all the utterances in our database, and retrain the system. If the noise characteristics are known ahead of time, this method allows us to adapt the model to the new environment with a relatively small amount of data from the new environment, yet use a large amount of training data. Figure 10.31 shows the benefit of this approach over a system trained on clean speech for the case of additive white noise.. If the target acoustical environment also has a different channel, we can also filter all the utterances in the training data prior to retraining. This method allows us to adapt the model to the new environment with a relatively small amount of data from the new environment.

If the noise sample is available offline, this simple technique can provide good results at no cost during recognition. Otherwise the noise addition and model retraining would need

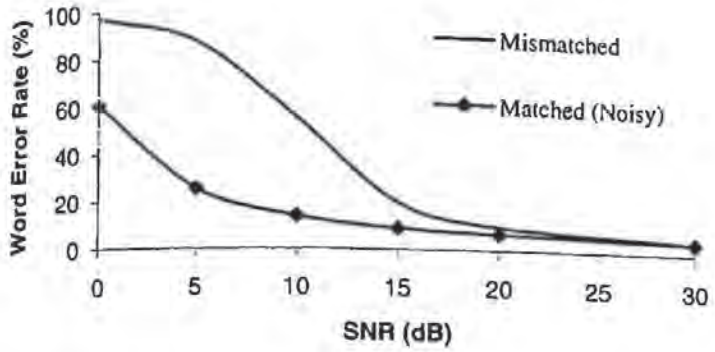


Figure 10.31 Word error rate as a function of the testing data SNR (dB) for Whisper trained on clean data and a system trained on noisy data at the same SNR as the testing set as in Figure 10.30. White noise at different SNRs is added.

to occur at runtime. This is feasible for speaker-dependent small-vocabulary systems where the training data can be kept in memory and where the retraining time can be small, but it is generally not feasible for large-vocabulary speaker-independent systems because of memory and computational limitations.

One possibility is to create a number of artificial acoustical environments by corrupting our clean database with noise samples of varying levels and types, as well as varying channels. Then all those waveforms from multiple acoustical environments can be used in training. This is called *multistyle training* [39], since our training data comes from different conditions. Because of the diversity of the training data, the resulting recognizer is more robust to varying noise conditions. In Figure 10.32 we see that, though generally the error-rate curve is above the matched-condition curve, particularly for clean speech, multistyle training does not require knowledge of the specific noise level and thus is a viable alternative to the theoretical lower bound of matched conditions.

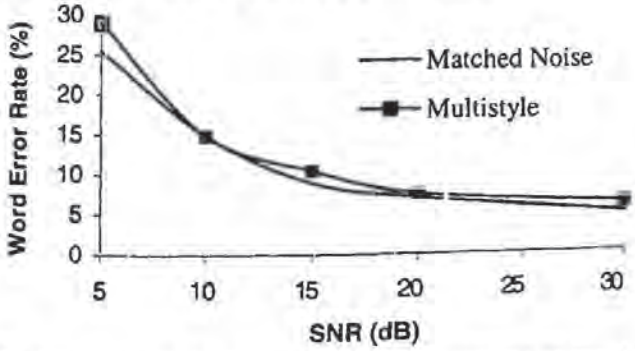


Figure 10.32 Word error rates of multistyle training compared to matched-noise training as a function of the SNR in dB for additive white noise. Whisper is trained as in Figure 10.30. The error rate of multistyle training is between 12% (for low SNR) and 25% (for high SNR) higher in relative terms than that of matched-condition training. Nonetheless, multistyle training does better than a system trained on clean data for all conditions other than clean speech.

10.6.2. Model Adaptation

We can also use the standard adaptation methods used for speaker adaptation, such as MAP or MLLR described in Chapter 9. Since MAP is an unstructured method, it can offer results similar to those of matched conditions, but it requires a significant amount of adaptation data. MLLR can achieve reasonable performance with about a minute of speech for minor mismatches [41]. For severe mismatches, MLLR also requires a large number of transformations, which, in turn, require a larger amount of adaptation data as discussed in Chapter 9.

Let's analyze the case of a single MLLR transform, where the affine transformation is simply a bias. In this case the MLLR transform consists only of a vector $\mathbf{h}$ that, as in the case of CMN described in Section 10.5.4, can be estimated from a single utterance. Instead of estimating $\mathbf{h}$ as the average cepstral mean, this method estimates $\mathbf{h}$ as the maximum likelihood estimate, given a set of sample vectors $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{T-1}\}$ and an HMM model λ [48], and it is a version of the EM algorithm where all the vector means are tied together (see Algorithm 10.2). This procedure for estimating the cepstral bias has a very slight reduction in error rates over CMN, although the improvement is larger for short utterances [48].

ALGORITHM 10.2: *MLE SIGNAL BIAS REMOVAL*

Step 1: Initialize $\mathbf{h}^{(0)} = \mathbf{0}$ at iteration $j = 0$

Step 2: Obtain model $\lambda^{(j)}$ by updating the means from $\mathbf{m}_k$ to $\mathbf{m}_k + \mathbf{h}^{(j)}$, for all Gaussians k.

Step 3: Run recognition with model $\lambda^{(j)}$ on the current utterance and determine a state segmentation $\theta[t]$ for each frame t.

Step 4: Estimate $\mathbf{h}^{(j+1)}$ as the vector that maximizes the likelihood, which, using covariance matrices $\boldsymbol{\Sigma}_k$, is given by:

$$\mathbf{h}^{(j+1)} = \left(\sum_{t=0}^{T-1} \boldsymbol{\Sigma}_{\theta[t]}^{-1}\right)^{-1} \sum_{t=0}^{T-1} \boldsymbol{\Sigma}_{\theta[t]}^{-1} \left(\mathbf{x}_t - \mathbf{m}_{\theta[t]}\right) \tag{10.148}$$

Step 5: If converged, stop; otherwise, increment j and go to Step 2. In practice two iterations are often sufficient.

If both additive noise and linear filtering are applied, the cepstrum for the noise and that for most speech frames are affected differently. The *speech/noise mean normalization* [5] algorithm can be extended similarly, as shown in Algorithm 10.3. The idea is to estimate a vector $\bar{\mathbf{n}}$ and $\bar{\mathbf{h}}$, such that all the Gaussians associated to the noise model are shifted by $\bar{\mathbf{n}}$, and all remaining Gaussians are shifted by $\bar{\mathbf{h}}$.

We can make Eq. (10.150) more efficient by tying all the covariance matrices. This transforms Eq. (10.150) into

$$\begin{aligned} \mathbf{h}^{(j+1)} &= \frac{1}{N_s} \sum_{t \in q_s} \mathbf{x}_t - \mathbf{m}_s \\ \mathbf{n}^{(j+1)} &= \frac{1}{N_n} \sum_{t \in q_n} \mathbf{x}_t - \mathbf{m}_n \end{aligned} \tag{10.149}$$

i.e., the difference between the average vector for speech frames in the utterance and the average vector $\mathbf{m}_s$ for speech frames in the training data, and similarly for the noise frames $\mathbf{m}_n$. This is essentially the same equation as in the speech-noise cepstral mean normalization described in Section 10.5.4. The difference is that the speech/noise discrimination is done by the recognizer instead of by a separate classifier. This method is more accurate in high-background-noise conditions where traditional speech/noise classifiers can fail. As a compromise, a codebook with considerably fewer Gaussians than a recognizer can be used to estimate $\bar{\mathbf{n}}$ and $\bar{\mathbf{h}}$.

ALGORITHM 10.3: *SPEECH/NOISE MEAN NORMALIZATION*

Step 1: Initialize $\mathbf{h}^{(0)} = \mathbf{0}$, $\mathbf{n}^{(0)} = 0$ at iteration $j = 0$

Step 2: Obtain model $\lambda^{(j)}$ by updating the means of speech Gaussians from $\mathbf{m}_k$ to $\mathbf{m}_k + \mathbf{h}^{(j)}$, and of noise Gaussians from $\mathbf{m}_l$ to $\mathbf{m}_l + \mathbf{n}^{(j)}$.

Step 3: Run recognition with model $\lambda^{(j)}$ on the current utterance and determine a state segmentation $\theta[t]$ for each frame t.

Step 4: Estimate $\mathbf{h}^{(j+1)}$ and $\mathbf{n}^{(j+1)}$ as the vectors that maximize the likelihood for speech frames ($t \in q_s$) and noise frames ($t \in q_n$), respectively:

$$\mathbf{h}^{(j+1)} = \left(\sum_{t \in q_s} \Sigma_{\theta[t]}^{-1} \right)^{-1} \sum_{t \in q_s} \Sigma_{\theta[t]}^{-1} \left(\mathbf{x}_t - \mathbf{m}_{\theta[t]} \right)$$

$$\mathbf{n}^{(j+1)} = \left(\sum_{t \in q_n} \Sigma_{\theta[t]}^{-1} \right)^{-1} \sum_{t \in q_n} \Sigma_{\theta[t]}^{-1} \left(\mathbf{x}_t - \mathbf{m}_{\theta[t]} \right) \qquad (10.150)$$

Step 5: If converged, stop; otherwise, increment j and go to Step 2.

10.6.3. Parallel Model Combination

By using the clean-speech models and a noise model, we can approximate the distributions obtained by training a HMM with corrupted speech. The memory requirements for the algorithm are then significantly reduced, as the training data is not needed online. *Parallel model combination* (PMC) is a method to obtain the distribution of noisy speech given the distribution of clean speech and noise as mixture of Gaussians. As discussed in Section 10.1.3, if the clean-speech cepstrum follows a Gaussian distribution and the noise cepstrum follows another Gaussian distribution, the noisy speech has a distribution that is no longer Gaussian. The PMC method nevertheless makes the assumption that the resulting distribution is Gaussian whose mean and covariance matrix are the mean and covariance matrix of the resulting non-Gaussian distribution. If it is assumed that the distribution of clean speech is a mixture of N Gaussians, and the distribution of the noise is a mixture of M Gaussians, the distribution of the noisy speech contains NM Gaussians. The feature vector is often composed of the cepstrum, delta cepstrum, and delta-delta cepstrum. The model combination can be seen in Figure 10.33.

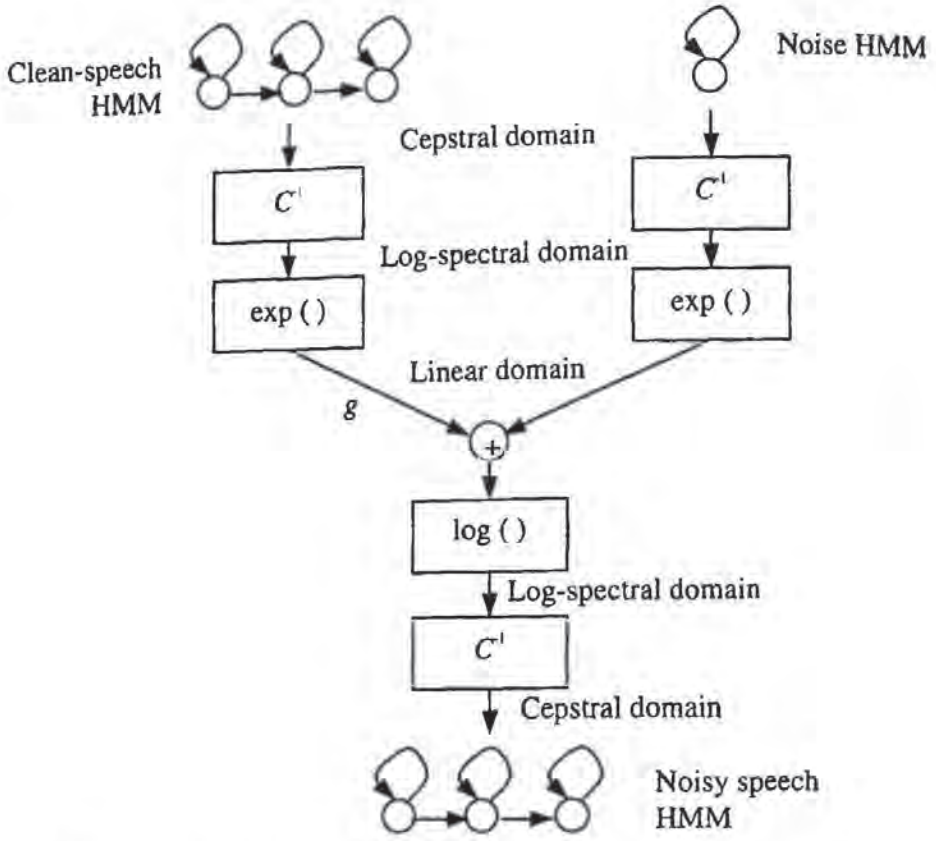


Figure 10.33 Parallel model combination for the case of one-state noise HMM.

If the mean and covariance matrix of the cepstral noise vector $\mathbf{n}$ are given by $\boldsymbol{\mu}_n^c$ and $\boldsymbol{\Sigma}_n^c$, respectively, we first compute the mean and covariance matrix in the log-spectral domain:

$$\begin{aligned}\boldsymbol{\mu}_n^l &= \mathbf{C}^{-1}\boldsymbol{\mu}_n^c \\ \boldsymbol{\Sigma}_n^l &= \mathbf{C}^{-1}\boldsymbol{\Sigma}_n^c(\mathbf{C}^{-1})^T\end{aligned} \tag{10.151}$$

In the linear domain $\mathbf{N} = e^{\mathbf{n}}$, the distribution is lognormal, whose mean vector $\boldsymbol{\mu}_N$ and covariance matrix $\boldsymbol{\Sigma}_N$ can be shown (see Chapter 3) to be given by

$$\begin{aligned}\boldsymbol{\mu}_N[i] &= \exp\left\{\boldsymbol{\mu}_n^l[i] + \boldsymbol{\Sigma}_n^l[i,i]/2\right\} \\ \boldsymbol{\Sigma}_N[i,j] &= \boldsymbol{\mu}_N[i]\boldsymbol{\mu}_N[j]\left(\exp\left\{\boldsymbol{\Sigma}_n^l[i,j]\right\} - 1\right)\end{aligned} \tag{10.152}$$

with expressions similar to Eqs. (10.151) and (10.152) for the mean and covariance matrix of $\mathbf{X}$.

Using the model of the environment with no filter is equivalent to obtaining a random linear spectral vector $\mathbf{Y}$ given by (see Figure 10.33)

$$\mathbf{Y} = \mathbf{X} + \mathbf{N} \tag{10.153}$$

and, since $\mathbf{X}$ and $\mathbf{N}$ are independent, we can obtain the mean and covariance matrix of $\mathbf{Y}$ as

$$\begin{aligned} \mu_{\mathbf{Y}} &= \mu_{\mathbf{X}} + \mu_{\mathbf{N}} \\ \Sigma_{\mathbf{Y}} &= \Sigma_{\mathbf{X}} + \Sigma_{\mathbf{N}} \end{aligned} \tag{10.154}$$

Although the sum of two lognormal distributions is not lognormal, the popular *log-normal approximation* [26] consists in assuming that $\mathbf{Y}$ is lognormal. In this case we can apply the inverse formulae of Eq. (10.152) to obtain the mean and covariance matrix in the log-spectral domain:

$$\begin{aligned} \Sigma_{\mathbf{y}}^{l}[i,j] &= \ln\left\{\frac{\Sigma_{\mathbf{Y}}[i,j]}{\mu_{\mathbf{Y}}[i]\mu_{\mathbf{Y}}[j]} + 1\right\} \\ \mu_{\mathbf{y}}^{l}[i] &= \ln\mu_{\mathbf{Y}}[i] - \frac{1}{2}\ln\left\{\frac{\Sigma_{\mathbf{Y}}[i,j]}{\mu_{\mathbf{Y}}[i]\mu_{\mathbf{Y}}[j]} + 1\right\} \end{aligned} \tag{10.155}$$

and finally return to the cepstrum domain applying the inverse of Eq. (10.151):

$$\begin{aligned} \mu_{\mathbf{y}}^{c} &= \mathbf{C}\mu_{\mathbf{y}}^{l} \\ \Sigma_{\mathbf{y}}^{c} &= \mathbf{C}\Sigma_{\mathbf{y}}^{l}\mathbf{C}^{T} \end{aligned} \tag{10.156}$$

The lognormal approximation cannot be used directly for the delta and delta-delta cepstrum. Another variant that can be used in this case and is more accurate than the lognormal approximation is the *data-driven parallel model combination* (DPMC) [26], which uses Monte Carlo simulation to draw random cepstrum vectors from both the clean-speech HMM and noise distribution to create cepstrum of the noisy speech by applying Eqs. (10.20) and (10.21) to each sample point. These composite cepstrum vectors are not kept in memory, only their means and covariance matrices are, therefore reducing the required memory though still requiring a significant amount of computation. The number of vectors drawn from the distribution was at least 100 in [26]. A way of reducing the number of random vectors needed to obtain good Monte Carlo simulations is proposed in [56]. A version of PMC using numerical integration, which is very computationally expensive, yielded the best results.

Figure 10.34 and Figure 10.35 compare the values estimated through the lognormal approximation to the true value, where for simplicity we deal with scalars. Thus x, n, and y represent the log-spectral energies of the clean signal, noise, and noisy signal, respectively, for a given frequency. Assuming x and n to be Gaussian with means μ_x and μ_n and variances σ_x and σ_n respectively, we see that the lognormal approximation is accurate when the standard deviations σ_x and σ_n are small.

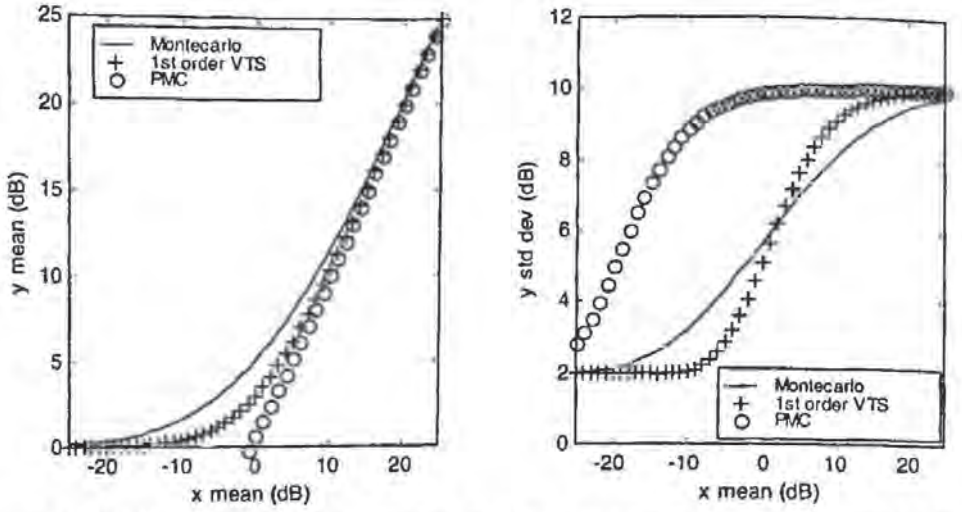


Figure 10.34 Means and standard deviation of noisy log-spectrum y in dB according to Eq. (10.165). The distribution of the noise log-spectrum n is Gaussian with mean 0 dB and standard deviation 2 dB. The distribution of the clean log-spectrum x is Gaussian, having a standard deviation of 10 dB and a mean varying from –25 to 25 dB. Both the mean and the standard deviation of y are more accurate in first-order VTS than in PMC.

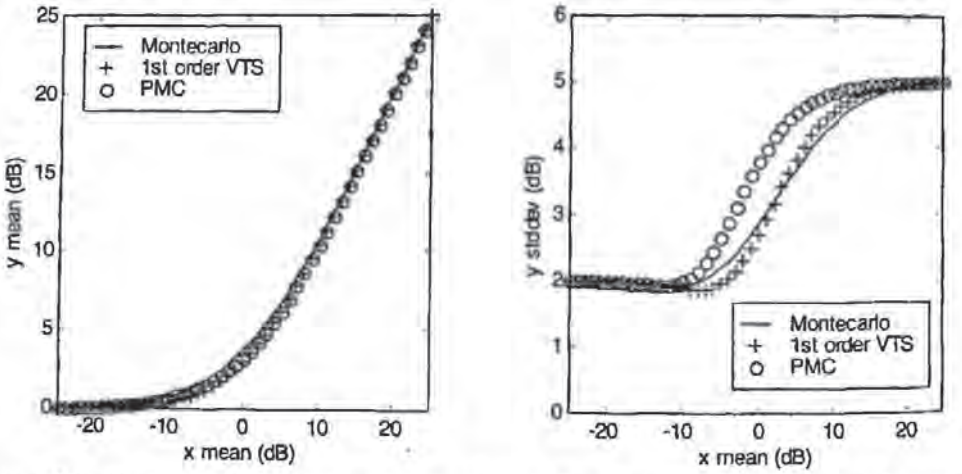


Figure 10.35 Means and standard deviation of noisy log-spectrum y in dB according to Eq. (10.165). The distribution of the noise log-spectrum n is Gaussian with mean 0 dB and standard deviation of 2 dB. The distribution of the clean log-spectrum x is Gaussian with a standard deviation of 5 dB and a mean varying from –25 dB to 25 dB. The mean of y is well estimated in both PMC and first-order VTS. The standard deviation of y is more accurate in first-order VTS than in PMC.

10.6.4. Vector Taylor Series

The model of the acoustical environment described in Section 10.1.3 describes the relationship between the cepstral vectors $\mathbf{x}$, $\mathbf{n}$, and $\mathbf{y}$ of the clean speech, noise, and noisy speech, respectively:

$$\mathbf{y} = \mathbf{x} + \mathbf{h} + \mathbf{g}(\mathbf{n} - \mathbf{x} - \mathbf{h}) \tag{10.157}$$

where $\mathbf{h}$ is the cepstrum of the filter, and the nonlinear function $\mathbf{g}(\mathbf{z})$ is given by

$$\mathbf{g}(\mathbf{z}) = \mathbf{C}\ln\left(1 + e^{\mathbf{C}^{-1}\mathbf{z}}\right) \tag{10.158}$$

Moreno [44] suggests the use of Taylor series to approximate the nonlinearity in Eq. (10.158), though he applies it in the spectral instead of the cepstral domain. We follow that approach to compute the mean and covariance matrix of $\mathbf{y}$ [4].

Assume that $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{n}$ are Gaussian random vectors with means $\boldsymbol{\mu}_x$, $\boldsymbol{\mu}_h$, and $\boldsymbol{\mu}_n$ and covariance matrices $\boldsymbol{\Sigma}_x$, $\boldsymbol{\Sigma}_h$, and $\boldsymbol{\Sigma}_n$, respectively, and furthermore that $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{n}$ are independent. After algebraic manipulation it can be shown that the Jacobian of Eq. (10.157) with respect to $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{n}$ evaluated at $\boldsymbol{\mu} = \boldsymbol{\mu}_n - \boldsymbol{\mu}_x - \boldsymbol{\mu}_h$ can be expressed as

$$\begin{aligned} \left.\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right|_{(\boldsymbol{\mu}_n, \boldsymbol{\mu}_x, \boldsymbol{\mu}_h)} &= \left.\frac{\partial \mathbf{y}}{\partial \mathbf{h}}\right|_{(\boldsymbol{\mu}_n, \boldsymbol{\mu}_x, \boldsymbol{\mu}_h)} = \mathbf{A} \\ \left.\frac{\partial \mathbf{y}}{\partial \mathbf{n}}\right|_{(\boldsymbol{\mu}_n, \boldsymbol{\mu}_x, \boldsymbol{\mu}_h)} &= \mathbf{I} - \mathbf{A} \end{aligned} \tag{10.159}$$

where the matrix $\mathbf{A}$ is given by

$$\mathbf{A} = \mathbf{C}\mathbf{F}\mathbf{C}^{-1} \tag{10.160}$$

and $\mathbf{F}$ is a diagonal matrix whose elements are given by vector $\mathbf{f}(\boldsymbol{\mu})$, which in turn is given by

$$\mathbf{f}(\boldsymbol{\mu}) = \frac{1}{1 + e^{\mathbf{C}^{-1}\boldsymbol{\mu}}} \tag{10.161}$$

Using Eq. (10.159) we can then approximate Eq. (10.157) by a first-order Taylor series expansion around $(\boldsymbol{\mu}_n, \boldsymbol{\mu}_x, \boldsymbol{\mu}_h)$ as

$$\begin{aligned} \mathbf{y} \approx\ & \boldsymbol{\mu}_x + \boldsymbol{\mu}_h + \mathbf{g}(\boldsymbol{\mu}_n - \boldsymbol{\mu}_x - \boldsymbol{\mu}_h) \\ & + \mathbf{A}(\mathbf{x} - \boldsymbol{\mu}_x) + \mathbf{A}(\mathbf{h} - \boldsymbol{\mu}_h) + (\mathbf{I} - \mathbf{A})(\mathbf{n} - \boldsymbol{\mu}_n) \end{aligned} \tag{10.162}$$

The mean of $\mathbf{y}$, $\boldsymbol{\mu}_y$, can be obtained from Eq. (10.162) as

$$\boldsymbol{\mu}_y \approx \boldsymbol{\mu}_x + \boldsymbol{\mu}_h + \mathbf{g}(\boldsymbol{\mu}_n - \boldsymbol{\mu}_x - \boldsymbol{\mu}_h) \tag{10.163}$$

and its covariance matrix $\boldsymbol{\Sigma}_y$ by

$$\boldsymbol{\Sigma}_y \approx \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T + \mathbf{A}\boldsymbol{\Sigma}_h\mathbf{A}^T + (\mathbf{I}-\mathbf{A})\boldsymbol{\Sigma}_n(\mathbf{I}-\mathbf{A})^T \tag{10.164}$$

so that even if $\boldsymbol{\Sigma}_x$, $\boldsymbol{\Sigma}_h$, and $\boldsymbol{\Sigma}_n$ are diagonal, $\boldsymbol{\Sigma}_y$ is no longer diagonal. Nonetheless, we can assume it to be diagonal, because this way we can transform a clean HMM to a corrupted HMM that has the same functional form and use a decoder that has been optimized for diagonal covariance matrices.

It is difficult to visualize how good the approximation is, given the nonlinearity involved. To provide some insight, let's consider the frequency-domain version of Eqs. (10.157) and (10.158) when no filtering is done:

$$y = x + \ln\left(1+\exp\left(n-x\right)\right) \tag{10.165}$$

where x, n, and y represent the log-spectral energies of the clean signal, noise, and noisy signal, respectively, for a given frequency. In Figure 10.34 we show the mean and standard deviation of the noisy log-spectral energy y in dB as a function of the mean of the clean log-spectral energy x with a standard deviation of 10 dB. The log-spectral energy of the noise n is Gaussian with mean 0 dB and standard deviation 2 dB. We compare the correct values obtained through Monte Carlo simulation (or DPMC) with the values obtained through the lognormal approximation of Section 10.6.3 and the first-order VTS approximation described here. We see that the VTS approximation is more accurate than the lognormal approximation for the mean and especially for the standard deviation of y, assuming the model of the environment described by Eq. (10.165).

Figure 10.35 is similar to Figure 10.34 except that the standard deviation of the clean log-energy x is only 5 dB, a more realistic number in speech recognition systems. In this case, both the lognormal approximation and the first-order VTS approximation are good estimates of the mean of y, though the standard deviation estimated through the lognormal approximation in PMC is not as good as that obtained through first-order VTS, again assuming the model of the environment described by Eq. (10.165). The overestimate of the variance in the lognormal approximation might, however, be useful if the model of the environment is not accurate.

To compute the means and covariance matrices of the delta and delta-delta parameters, let's take the derivative of the approximation of $\mathbf{y}$ in Eq. (10.162) with respect to time:

$$\frac{\partial \mathbf{y}}{\partial t} \approx \mathbf{A}\frac{\partial \mathbf{x}}{\partial t} \tag{10.166}$$

so that the delta-cepstrum computed through $\Delta\mathbf{x}_t = \mathbf{x}_{t+2} - \mathbf{x}_{t-2}$, is related to the derivative [28] by

$$\Delta \mathbf{x} \approx 4\frac{\partial \mathbf{x}_t}{\partial t} \tag{10.167}$$

so that

$$\mu_{\Delta y} \approx \mathbf{A}\mu_{\Delta x} \tag{10.168}$$

and similarly

$$\Sigma_{\Delta y} \approx \mathbf{A}\Sigma_{\Delta x}\mathbf{A}^T + (\mathbf{I}-\mathbf{A})\Sigma_{\Delta n}(\mathbf{I}-\mathbf{A})^T \tag{10.169}$$

where we assumed that $\mathbf{h}$ is constant within an utterance, so that $\Delta \mathbf{h} = 0$.

Similarly, for the delta-delta cepstrum, the mean is given by

$$\mu_{\Delta^2 y} \approx \mathbf{A}\mu_{\Delta^2 x} \tag{10.170}$$

and the covariance matrix by

$$\Sigma_{\Delta^2 y} \approx \mathbf{A}\Sigma_{\Delta^2 x}\mathbf{A}^T + (\mathbf{I}-\mathbf{A})\Sigma_{\Delta^2 n}(\mathbf{I}-\mathbf{A})^T \tag{10.171}$$

where we again assumed that $\mathbf{h}$ is constant within an utterance, so that $\Delta^2 \mathbf{h} = 0$.

Equations (10.163), (10.168), and (10.170) resemble the MLLR adaptation formulae of Chapter 9 for the means, though in this case the matrix is different for each Gaussian and is heavily constrained.

We are interested in estimating the environmental parameters μ_n, μ_h, and Σ_n, given a set of T observation frames $\mathbf{y}_t$. This estimation can be done iteratively using the EM algorithm on Eq. (10.162). If the noise process is stationary, $\Sigma_{\Delta n}$ could be approximated, assuming independence between $\mathbf{n}_{t+2}$ and $\mathbf{n}_{t-2}$, by $\Sigma_{\Delta n} = 2\Sigma_n$. Similarly, $\Sigma_{\Delta^2 n}$ could be approximated, assuming independence between $\Delta \mathbf{n}_{t+1}$ and $\Delta \mathbf{n}_{t-1}$, by $\Sigma_{\Delta^2 n} = 4\Sigma_n$. If the noise process is not stationary, it is best to estimate $\Sigma_{\Delta n}$ and $\Sigma_{\Delta^2 n}$ from input data directly.

If the distribution of $\mathbf{x}$ is a mixture of N Gaussians, each Gaussian is transformed according to the equations above. If the distribution of $\mathbf{n}$ is also a mixture of M Gaussians, the composite distribution has NM Gaussians. While this increases the number of Gaussians, the decoder is still functionally the same as for clean speech. Because normally you do not want to alter the number of Gaussians of the system when you do noise adaptation, it is often assumed that $\mathbf{n}$ is a single Gaussian.

10.6.5. Retraining on Compensated Features

We have discussed adapting the HMM to the new acoustical environment using the standard front-end features, in most cases the mel-cepstrum. Section 10.5 dealt with cleaning the noisy feature without retraining the HMMs. It's logical to consider a combination of both, where the features are cleaned to remove noise and channel effects and then the HMMs are retrained to take into account that this processing stage is not perfect. This idea is illustrated

in Figure 10.36, where we compare the word error rate of the standard matched-noise-condition training with the matched-noise-condition training after it has been compensated by a variant of the mixture Gaussian algorithms described in Section 10.5.6 [21]. An improvement is obtained by retraining on compensated features, which beats the unprocessed matched-condition training.

The low error rates of both curves in Figure 10.36 are hard to obtain in practice, because they assume we know exactly what the noise level and type are ahead of time, which in general is hard to do. On the other hand, this could be combined with the multistyle training discussed in Section 10.6.1 or with a set of clustered models discussed in Chapter 9.

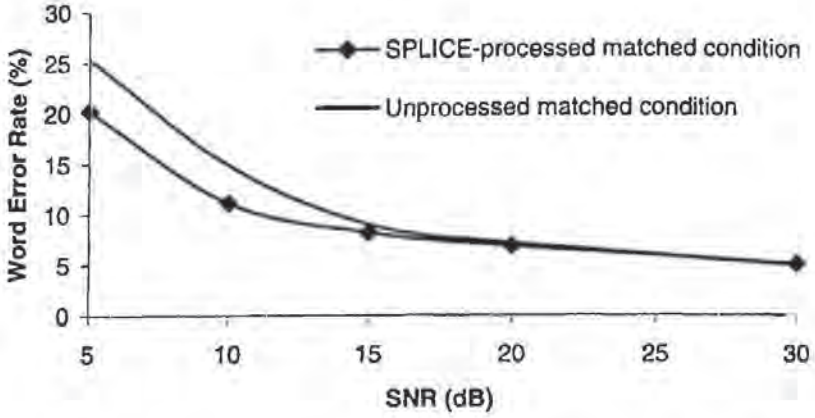


Figure 10.36 Word error rates of matched-noise training without feature preprocessing and with the SPLICE algorithm [21] as a function of the SNR in dB for additive white noise. Whisper is trained as in Figure 10.30. Error rate with the mixture Gaussian model is up to 30% lower than that of standard noisy matched conditions for low SNRs while it is about the same for high SNRs.

10.7. MODELING NONSTATIONARY NOISE

The previous sections deal mostly with stationary noise. In practice, there are many nonstationary noises that often match a random word in the system's lexicon better than the silence model. In this case, the benefit of using speech recognition vanishes quickly.

The most typical types of noise present in desktop applications are mouth noise (lip smacks, throat clearings, coughs, nasal clearings, heavy breathing, uhms and uhs, etc), computer noise (keyboard typing, microphone adjustment, computer fan, disk head seeking, etc.), and office noise (phone rings, paper rustles, shutting door, interfering speakers, etc.). We can use a simple method that has been successful in speech recognition [57], as shown in Algorithm 10.4. This method consists of adding noise words modeled with HMMs to absorb these nonstationary noises.

In practice, updating the transcription turns out to be important, because human labelers often miss short noises that the system can uncover. Since the noise training data are often limited in terms of coverage, some noises can be easily matched to short word models, such as: *if, two*. Due to the unique characteristics of noise rejection, we often need to further augment confidence measures such as those described in Chapter 9. In practice, we need an additional classifier to provide more detailed discrimination between speech and noise. We can use a two-level classifier for this purpose. The ratio between the *all-speech model* score (fully connected context-independent phone models) and the *all-noise model score* (fully connected silence and noise phone models) can be used.

Another approach [55] consists of having an HMM for noise with several states to deal with nonstationary noises. The decoder needs to conduct a three-dimensional Viterbi search which evaluates at each frame every possible speech state as well as every possible noise state to achieve the *speech/noise decomposition* (see Figure 10.37). The computational complexity of such an approach is very large, though it can handle nonstationary noises quite well in theory.

ALGORITHM 10.4: *EXPLICIT NOISE MODELING*

Step 1: Augmenting the vocabulary with *noise words* (such as ++SMACK++), each composed of a single *noise phoneme* (such as +SMACK+), which are thus modeled with a single HMM. These noise words have to be labeled in the transcriptions so that they can be trained.
Step 2: Training noise models, as well as the other models, using the standard HMM training procedure.
Step 3: Updating the transcription. To do that, convert the transcription into a network, where the noise words can be optionally inserted between each word in the original transcription. A forced alignment segmentation is then conducted with the current HMM optional noise words inserted. The segmentation with the highest likelihood is selected, thus yielding an optimal transcription.
Step 4: If converged, stop; otherwise go to Step 2.

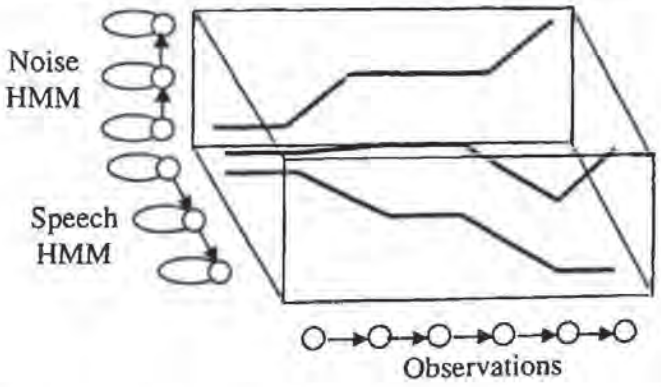


Figure 10.37 Speech noise decomposition and a three-dimensional Viterbi decoder.

10.8. HISTORICAL PERSPECTIVE AND FURTHER READING

This chapter contains a number of diverse topics that are often described in different fields; no single reference covers it all. For further reading on adaptive filtering, you can check the books by Widrow and Stearns [59] and Haykin [30]. Theodoridis and Bellanger provide [54] a good summary of adaptive filtering, and Breining et al. [16] a good summary of echo-canceling techniques. Lee [38] has a good summary of independent component analysis for blind source separation. Deller et al. [20] provide a number of techniques for speech enhancement. Juang [35] and Junqua [37] survey techniques used in improving the robustness of speech recognition systems to noise. Acero [2] compares a number of feature transformation techniques in the cepstral domain and introduces the model of the environment used in this chapter.

Adaptive filtering theory emerged early in the 1900s. The Wiener and LMS filters were derived by Wiener and Widrow in 1919 and 1960, respectively. Norbert Wiener joined the MIT faculty in 1919 and made profound contributions to generalized harmonic analysis, the famous Wiener-Hopf equation, and the resulting Wiener filter. The LMS algorithm was developed by Widrow and his colleagues at Stanford University in the early 1960s.

From a practical point of view, the use of gradient microphones (Olsen [46]) has proven to be one of the more important contributions to increased robustness. Directional microphones are commonplace today in most speech recognition systems.

Boll [13] first suggested the use of spectral subtraction. This has been the cornerstone for noise suppression, and many systems nowadays still use a variant of Boll's original algorithm.

The Cepstral mean normalization algorithm was proposed by Atal [8] in 1974, although it wasn't until the early 1990s that it became commonplace in most speech recognition systems evaluated in the DARPA speech programs [33]. Hermansky proposed PLP [31] in 1990. The work of Rich Stern's robustness group at CMU (especially the Ph.D. thesis work of Acero [1] and Moreno [43]) and the Ph.D. thesis of Gales [26] also represented advances in the understanding of the effect of noise in the cepstrum.

Bell and Sejnowski [10] gave the field of independent component analysis a boost in 1995 with their infomax rule. The field of source separation is a promising alternative to improve the robustness of speech recognition systems when more than one microphone is available.

REFERENCES

[1] Acero, A., Acoustical and Environmental Robustness in Automatic Speech Recognition, PhD Thesis in Electrical and Computer Engineering 1990, Carnegie Mellon University, Pittsburgh.

[2] Acero, A., Acoustical and Environmental Robustness in Automatic Speech Recognition, 1993, Boston, Kluwer Academic Publishers.

[3] Acero, A., S. Altschuler, and L. Wu, "Speech/Noise Separation Using Two Microphones and a VQ Model of Speech Signals," Int. Conf. on Spoken Language Processing, 2000, Beijing, China.

[4] Acero, A., et al., "HMM Adaptation Using Vector Taylor Series for Noisy Speech Recognition," Int. Conf. on Spoken Language Processing, 2000, Beijing, China.

[5] Acero, A. and X.D. Huang, "Augmented Cepstral Normalization for Robust Speech Recognition," Proc. of the IEEE Workshop on Automatic Speech Recognition, 1995, Snowbird, UT.

[6] Acero, A. and R. Stern, "Environmental Robustness in Automatic Speech Recognition," Int. Conf. on Acoustics, Speech and Signal Processing, 1990, Albuquerque, NM, pp. 849-852.

[7] Amari, S., A. Cichocki, and H.H. Yang, eds. A New Learning Algorithm for Blind Separation, Advances in Neural Information Processing Systems, 1996, Cambridge, MA, MIT Press.

[8] Atal, B.S., "Effectiveness of Linear Prediction Characteristics of the Speech Wave for Automatic Speaker Identification and Verification," Journal of the Acoustical Society of America, 1974, **55**(6), pp. 1304-1312.

[9] Attias, H., "Independent Factor Analysis," Neural Computation, 1998, **11**, pp. 803-851.

[10] Bell, A.J. and T.J. Sejnowski, "An Information Maximisation Approach to Blind Separation and Blind Deconvolution," Neural Computation, 1995, **7**(6), pp. 1129-1159.

[11] Belouchrani, A., et al., "A Blind Source Separation Technique Using Second Order Statistics," IEEE Trans. on Signal Processing, 1997, **45**(2), pp. 434-444.

[12] Berouti, M., R. Schwartz, and J. Makhoul, "Enhancement of Speech Corrupted by Acoustic Noise," Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 1979, pp. 208-211.

[13] Boll, S.F., "Suppression of Acoustic Noise in Speech Using Spectral Subtraction," IEEE Trans. on Acoustics, Speech and Signal Processing, 1979, **27**(Apr.), pp. 113-120.

[14] Boll, S.F. and D.C. Pulsipher, "Suppression of Acoustic Noise in Speech Using Two Microphone Adaptive Noise Cancellation," IEEE Trans. on Acoustics Speech and Signal Processing, 1980, **28**(December), pp. 751-753.

[15] Bregman, A.S., Auditory Scene Analysis, 1990, Cambridge MA, MIT Press.

[16] Breining, C., Acoustic Echo Control, in IEEE Signal Processing Magazine, 1999. pp. 42-69.

[17] Cardoso, J., "Blind Signal Separation: Statistical Principles," Proc. of the IEEE, 1998, **9**(10), pp. 2009-2025.

[18] Cardoso, J.F., "Infomax and Maximum Likelihood for Blind Source Separation," IEEE Signal Processing Letters, 1997, **4**, pp. 112-114.

[19] Comon, P., "Independent Component Analysis: A New Concept," Signal Processing, 1994, **36**, pp. 287-314.

[20] Deller, J.R., J.H.L. Hansen, and J.G. Proakis, Discrete-Time Processing of Speech Signals. 2000. IEEE Press.

[21] Deng, L., et al., "Large-Vocabulary Speech Recognition Under Adverse Acoustic Environments," Int. Conf. on Spoken Language Processing, 2000, Beijing, China.

[22] Ephraim, Y., "Statistical Model-Based Speech Enhancement System," Proc. of the IEEE. 1992, **80**(1), pp. 1526-1555.

[23] Ephraim, Y. and D. Malah, "Speech Enhancement Using Minimum Mean Square Error Short Time Spectral Amplitude Estimator," IEEE Trans. on Acoustics, Speech and Signal Processing, 1984, **32**(6), pp. 1109-1121.

[24] Flanagan, J.L., et al., "Computer-Steered Microphone Arrays for Sound Transduction in Large Rooms," Journal of the Acoustical Society of America, 1985, **78**(5), pp. 1508-1518.

[25] Frost, O.L., "An Algorithm for Linearly Constrained Adaptive Array Processing," Proc. of the IEEE, 1972, **60**(8), pp. 926-935.

[26] Gales, M.J., Model Based Techniques for Noise Robust Speech Recognition, PhD Thesis in Engineering Department, 1995, Cambridge University.

[27] Ghitza, O., "Robustness against Noise: The Role of Timing-Synchrony Measurement," Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 1987, pp. 2372-2375.

[28] Gopinath, R.A., et al., "Robust Speech Recognition in Noise—Performance of the IBM Continuous Speech Recognizer on the ARPA Noise Spoke Task," Proc. ARPA Workshop on Spoken Language Systems Technology, 1995, pp. 127-133.

[29] Griffiths, L.J. and C.W. Jim, "An Alternative Approach to Linearly Constrained Adaptive Beamforming," IEEE Trans. on Antennas and Propagation, 1982, **30**(1), pp. 27-34.

[30] Haykin, S., Adaptive Filter Theory, 2nd ed, 1996, Upper Saddle River, NJ, Prentice-Hall.

[31] Hermansky, H., "Perceptual Linear Predictive (PLP) Analysis of Speech," Journal of the Acoustical Society of America, 1990, **87**(4), pp. 1738-1752.

[32] Hermansky, H. and N. Morgan, "RASTA Processing of Speech," IEEE Trans. on Speech and Audio Processing, 1994, **2**(4), pp. 578-589.

[33] Huang, X.D., et al., "The SPHINX-II Speech Recognition System: An Overview," Computer Speech and Language, 1993, pp. 137-148.

[34] Hunt, M. and C. Lefebre, "A Comparison of Several Acoustic Representations for Speech Recognition with Degraded and Undegraded Speech," Int. Conf. on Acoustic, Speech and Signal Processing, 1989, pp. 262-265.

[35] Juang, B.H., "Speech Recognition in Adverse Environments," Computer Speech and Language, 1991, **5**, pp. 275-294.

[36] Junqua, J.C., "The Lombard Reflex and Its Role in Human Listeners and Automatic Speech Recognition," Journal of the Acoustical Society of America, 1993, **93**(1), pp. 510-524.

[37] Junqua, J.C. and J.P. Haton, Robustness in Automatic Speech Recognition, 1996, Kluwer Academic Publishers.

[38] Lee, T.W., Independent Component Analysis: Theory and Applications, 1998, Kluwer Academic Publishers.

[39] Lippmann, R.P., E.A. Martin, and D.P. Paul, "Multi-Style Training for Robust Isolated-Word Speech Recognition," Int. Conf. on Acoustics, Speech and Signal Processing, 1987, Dallas, TX, pp. 709-712.

[40] Lombard, E., "Le Signe de l'élévation de la Voix," Ann. Maladies Oreille, Larynx, Nez, Pharynx, 1911, **37**, pp. 101-119.

[41] Matassoni, M., M. Omologo, and D. Giuliani, "Hands-Free Speech Recognition Using a Filtered Clean Corpus and Incremental HMM Adaptation," Proc. Int. Conf. on Acoustics, Speech and Signal Processing, 2000, Istanbul, Turkey, pp. 1407-1410.

[42] Mendel, J.M., Lessons in Estimation Theory for Signal Processing, Communications, and Control, 1995, Upper Saddle River, NJ, Prentice Hall.

[43] Moreno, P., Speech Recognition in Noisy Environments, PhD Thesis in Electrical and Computer Engineering 1996, Carnegie Mellon University, Pittsburgh.

[44] Moreno, P.J., B. Raj, and R.M. Stern, "A Vector Taylor Series Approach for Environment Independent Speech Recognition," Int. Conf. on Acoustics, Speech and Signal Processing, 1996, Atlanta, pp. 733-736.

[45] Morgan, N. and H. Bourlard, Continuous Speech Recognition: An Introduction to Hybrid HMM/Connectionist Approach, in IEEE Signal Processing Magazine, 1995, pp. 25-42.

[46] Olsen, H.F., "Gradient Microphones," Journal of the Acoustical Society of America, 1946, 17,(3), pp. 192-198.

[47] Porter, J.E. and S.F. Boll, "Optimal Estimators for Spectral Restoration of Noisy Speech," Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 1984, San Diego, CA, pp. 18.A.2.1-4.

[48] Rahim, M.G. and B.H. Juang, "Signal Bias Removal by Maximum Likelihood Estimation for Robust Telephone Speech Recognition," IEEE Trans. on Speech and Audio Processing, 1996, **4**(1), pp. 19-30.

[49] Seneff, S., "A Joint Synchrony/Mean-Rate Model of Auditory Speech Processing," Journal of Phonetics, 1988, **16**(1), pp. 55-76.

[50] Sharma, S., et al., "Feature Extraction Using Non-Linear Transformation for Robust Speech Recognition on the Aurora Database," Int. Conf. on Acoustics, Speech and Signal Processing, 2000, Istanbul, Turkey, pp. 1117-1120.

[51] Sullivan, T.M. and R.M. Stern, "Multi-Microphone Correlation-Based Processing for Robust Speech Recognition," Int. Conf. on Acoustics, Speech and Signal Processing, 1993, Minneapolis, pp. 2091-2094.

[52] Suzuki, Y., et al., "An Optimum Computer-Generated Pulse Signal Suitable for the Measurement of Very Long Impulse Responses," Journal of the Acoustical Society of America, 1995, **97**(2), pp. 1119-1123.

[53] Tamura, S. and A. Waibel, "Noise Reduction Using Connectionist Models," Int. Conf. on Acoustics, Speech and Signal Processing, 1988, New York, pp. 553-556.

[54] Theodoridis, S. and M.G. Bellanger, Adaptive Filters and Acoustic Echo Control, in IEEE Signal Processing Magazine, 1999, pp. 12-41.

[55] Varga, A.P. and R.K. Moore, "Hidden Markov Model Decomposition of Speech and Noise," Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 1990 pp. 845-848.

[56] Wan, E.A., R.V.D. Merwe, and A.T. Nelson, "Dual Estimation and the Unscented Transformation" in Advances in Neural Information Processing Systems, S.A. Solla, T.K. Leen, and K.R. Muller, eds. 2000, Cambridge, MA, MIT Press, pp. 666-672.

[57] Ward, W., "Modeling Non-Verbal Sounds for Speech Recognition," Proc. Speech and Natural Language Workshop, 1989, Cape Cod, MA, Morgan Kauffman, pp. 311-318.

[58] Widrow, B. and M.E. Hoff, "Adaptive Switching Algorithms," IRE Wescon Convention Record, 1960, pp. 96-104.

[59] Widrow, B. and S.D. Stearns, Adaptive Signal Processing, 1985, Upper Saddle River, NJ, Prentice Hall.

[60] Woodland, P.C., "Improving Environmental Robustness in Large Vocabulary Speech Recognition," Int. Conf. on Acoustics, Speech and Signal Processing, 1996, Atlanta, Georgia, pp. 65-68.

CHAPTER 11

Language Modeling

Acoustic pattern matching, as discussed in Chapter 9, and knowledge about language are equally important in recognizing and understanding natural speech. Lexical knowledge (i.e., vocabulary definition and word pronunciation) is required, as are the syntax and semantics of the language (the rules that determine what sequences of words are grammatically well-formed and meaningful). In addition, knowledge of the pragmatics of language (the structure of extended discourse, and what people are likely to say in particular contexts) can be important to achieving the goal of spoken language understanding systems. In practical speech recognition, it may be impossible to separate the use of these different levels of knowledge, since they are often tightly integrated.

In this chapter we review the basic concept of Chomsky's formal language theory and the probabilistic language model. For the formal language model, two things are fundamental: the grammar and the parsing algorithm. The *grammar* is a formal specification of the permissible structures for the language. The *parsing* technique is the method of analyzing the sentence to see if its structure is compliant with the grammar. With the advent of bodies

of text (*corpora*) that have had their structures hand-annotated, it is now possible to generalize the formal grammar to include accurate probabilities. Furthermore, the probabilistic relationship among a sequence of words can be directly derived and modeled from the corpora with the so-called stochastic language models, such as *n*-gram, avoiding the need to create broad coverage formal grammars. Stochastic language models play a critical role in building a working spoken language system, and we discuss a number of important issues associated with them.

11.1. FORMAL LANGUAGE THEORY

In constructing a syntactic grammar for a language, it is important to consider the generality, the selectivity, and the understandability of the grammar. The *generality* and *selectivity* basically determine the range of sentences the grammar accepts and rejects. The *understandability* is important, since it is up to the authors of the system to create and maintain the grammar. For SLU systems described in Chapter 17, we need to have a grammar that covers and generalizes to most of the typical sentences for an application. The system also needs to distinguish the kind of sentences for different actions in a given application. Without understandability, it is almost impossible to improve a practical SLU system since it typically involves a large number of developers to maintain and refine the grammar.

The most common way of representing the grammatical structure of a sentence, "*Mary loves that person,*" is by using a tree, as illustrated in Figure 11.1. The node labeled *S* is the parent node of the nodes labeled *NP* and *VP* for noun phrase and verb phrase, respectively. The *VP* node is the parent node of node *V*—for verb. Each leaf is associated with the word

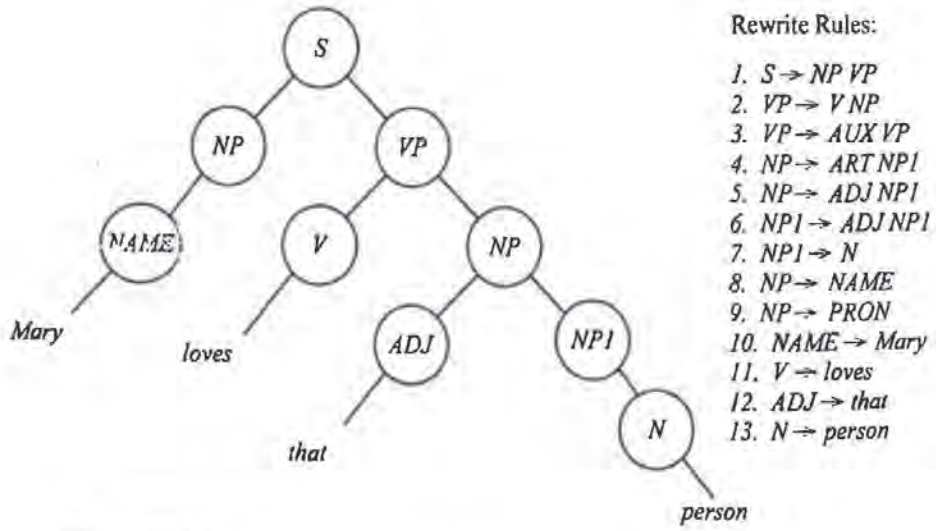


Figure 11.1 A tree representation of a sentence and its corresponding grammar.

in the sentence to be analyzed. To construct such a tree for a sentence, we must know the structure of the language so that a set of rewrite rules can be used to describe what tree structures are allowable. These rules, as illustrated in Figure 11.1, determine that a certain symbol may be expanded in the tree by a sequence of symbols. The grammatical structure helps in determining the meaning of the sentence. It tells us that *that* in the sentence modifies *person*. "Mary loves *that* person."

11.1.1. Chomsky Hierarchy

In Chomsky's *formal language theory* [1, 14, 15], a grammar is defined as $G = (V, T, P, S)$, where V and T are finite sets of *non-terminals* and *terminals*, respectively. V contains all the *non-terminal* symbols. We often use upper-case symbols to denote them. In the example discussed here, *S, NP, NP1, VP, NAME, ADJ, N*, and *V* are non-terminal symbols. The *terminal* set T contains *Mary, loves, that*, and *person*, which are often denoted with lower-case symbols. P is a finite set of *production (rewrite) rules*, as illustrated in the rewrite rules in Figure 11.1. S is a special non-terminal, called the *start symbol*.

The language to be analyzed is essentially a string of terminal symbols, such as "*Mary loves that person.*" It is produced by applying production rules sequentially to the start symbol. The production rule is of the form $\alpha \rightarrow \beta$, where α and β are arbitrary strings of grammar symbols V and T, and the α must not be empty. In formal language theory, four major languages and their associated grammars are hierarchically structured. They are referred to as the Chomsky hierarchy [1] as defined in Table 11.1. There are four kinds of automata that can accept the languages produced by these four types of grammars. Among these automata, the finite-state automaton is not only the mathematical device used to implement the regular grammar but also one of the most significant tools in computational linguistics. Variations of automata such as finite-state transducers, hidden Markov models, and n-gram models are important examples in spoken language processing.

These grammatical formulations can be compared according to their generative capacity, i.e., the range that the formalism can cover. While there is evidence that natural languages are at least weakly context sensitive, the context-sensitive requirements are rare in practice. The context-free grammar (CFG) is a very important structure for dealing with both machine language and natural language. CFGs are not only powerful enough to describe most of the structure in spoken language,[1] but also restrictive enough to have efficient parsers to analyze natural sentences. Since CFGs offer a good compromise between parsing efficiency and power in representing the structure of the language, they have been widely applied to natural language processing. Alternatively, regular grammars, as represented with a finite-state machine, can be applied to more restricted applications. Since finite-state grammars are a subset of the more general context-free grammar, we focus our discussion on context-free grammars only, although the parsing algorithm for finite-state grammars can be more efficient.

[1] The effort to prove natural languages are not context-free is summarized in Pullum and Gazdar [54].

Table 11.1 Chomsky hierarchy and the corresponding machine that accepts the language.

Types	Constraints	Automata
Phrase structure grammar	$\alpha \rightarrow \beta$. This is the most general grammar.	Turing machine
Context-sensitive grammar	A subset of the phrase structure grammar. $\|\alpha\| \leq \|\beta\|$, where |.| indicates the length of the string.	Linear bounded automata
Context-free grammar (CFG)	A subset of the context sensitive grammar. The production rule is $A \rightarrow \beta$, where A is a non-terminal. This production rule is shown to be equivalent to Chomsky normal form: $A \rightarrow w$ and $A \rightarrow BC$, where w is a terminal and B, C are non-terminals.	Push down automata
Regular grammar	A subset of the CFG. The production rule is expressed as: $A \rightarrow w$ and $A \rightarrow wB$.	Finite-state automata

As discussed in Section 11.1.2, a parsing algorithm offers a procedure that searches through various ways of combining grammatical rules to find a combination that generates a tree to illustrate the structure of the input sentence, which is similar to the search problem in speech recognition. The result of the parsing algorithm is a parse tree,[2] which can be regarded as a record of the CFG rules that account for the structure of the sentence. In other words, if we parse the sentence, working either top-down from S or bottom-up from each word, we automatically derive something that is similar to the tree representation, as illustrated in Figure 11.1.

A push-down automaton is also called a *recursive transition network* (RTN), which is an alternative formalism to describe context-free grammars. A transition network consists of nodes and labeled arcs. One of the nodes is specified as the initial state S. Starting at the initial state, we traverse an arc if the current word in the sentence is in the category on the arc. If the arc is followed, the current word is updated to the next word. A phrase can be parsed if there is a path from the starting node to a *pop* arc that indicates a complete parse for all the words in the phrase. Simple transition networks without recursion are often called *finite-state machines* (FSM). Finite-state machines are equivalent in expressive power to regular grammars and, thus, are not powerful enough to describe all languages that can be described by CFGs. Chapter 12 has a more detailed discussion on RTNs and FSMs used in speech recognition.

[2] The result can be more than one parse tree since natural language sentences are often ambiguous. In practice, a parsing algorithm should not only consider all the possible parse trees but also provide a ranking among them, as discussed in Chapter 17.

11.1.2. Chart Parsing for Context-Free Grammars

Since Chomsky introduced the notion of context-free grammars in the 1950s, a vast literature has arisen on the parsing algorithms. Most parsing algorithms were developed in computer science to analyze programming languages that are not ambiguous in the way that spoken language is [1, 32]. We discuss only the most relevant materials that are fundamental to building spoken language systems, namely the chart parser for the context-free grammar. This algorithm has been widely used in state-of-the-art spoken language understanding systems.

11.1.2.1. Top Down or Bottom Up?

Parsing is a special case of the search problem generally encountered in speech recognition. A parsing algorithm offers a procedure that searches through various ways of combining grammatical rules to find a combination that generates a tree to describe the structure of the input sentence, as illustrated in Figure 11.1. The search procedure can start from the root of the tree with the *S* symbol, attempting to rewrite it into a sequence of terminal symbols that matches the words in the input sentence, which is based on *goal-directed search*. Alternatively, the search procedure can start from the words in the input sentence and identify a word sequence that matches some non-terminal symbol. The bottom-up procedure can be repeated with partially parsed symbols until the root of the tree or the start symbol *S* is identified. This *data-directed search* has been widely used in practical SLU systems.

A top-down approach starts with the *S* symbol, then searches through different ways to rewrite the symbols until the input sentence is generated, or until all possibilities have been examined. A grammar is said to accept a sentence if there is a sequence of rules that allow us to rewrite the start symbol into the sentence. For the grammar in Figure 11.1, a sequence of rewrite rules can be illustrated as follows:

```
S
→ NP VP (rewriting S using S→NP)
→NAME VP (rewriting NP using NP→NAME)
→Mary VP (rewriting NAME using NAME→Mary)
...
→Mary loves that person (rewriting N using N→person)
```

Alternatively, we can take a bottom-up approach to start with the words in the input sentence and use the rewrite rules backward to reduce the sequence of symbols until it becomes S. The left-hand side of each rule is used to rewrite the symbol on the right-hand side as follows:

```
→NAME loves that person (rewriting Mary using NAME→Mary)
→NAME V that person (rewriting loves using V→loves)
...
→NP VP
→S (rewriting NP using S→NP VP)
```

A parsing algorithm must systematically explore every possible state that represents the intermediate node in the parsing tree. If a mistake occurs early on in choosing the rule that rewrites *S*, the intermediate parser results can be quite wasteful if the number of rules becomes large.

The main difference between top-down and bottom-up parsers is the way the grammar rules are used. For example, consider the rule *NP→ADJ NP1*. In a top-down approach, the rule is used to identify an *NP* by looking for the sequence *ADJ NP1*. Top-down parsing can be very predictive. A phrase or a word may be ambiguous in isolation. The top-down approach may prevent some ungrammatical combinations from consideration. It never wastes time exploring trees that cannot result in an *S*. On the other hand, it may predict many different constituents that do not have a match to the input sentence and rebuild large constituents again and again. For example, when the grammar is *left-recursive* (i.e., it contains a non-terminal category that has a derivation that includes itself anywhere along its leftmost branch), the top-down approach can lead a top-down, depth-first left-to-right parser to recursively expand the same non-terminal over again in exactly the same way. This causes an infinite expansion of trees. In contrast, a bottom-up parser takes a sequence *ADJ NP1* and identifies it as an *NP* according to the rule. The basic operation in bottom-up parsing is to take a sequence of symbols and match it to the right-hand side of the rules. It checks the input only once, and only builds each constituent exactly once. However, it may build up trees that have no hope of leading to *S* since it never suggests trees that are not at least locally grounded in the actual input. Since bottom-up parsing is similar to top-down parsing in terms of overall performance and is particularly suitable for robust spoken language processing as described in Chapter 17, we use the bottom-up method as our example to understand the key concept in the next section.

11.1.2.2. Bottom-Up Chart Parsing

As a standard search procedure, the state of the search consists of a symbol list, starting with the words in the sentence. Successor states can be generated by exploring all possible ways to replace a sequence of symbols that matches the right-hand side of a grammar rule with its left-hand side symbol. A simple-minded solution enumerates all the possible matches, leading to prohibitively expensive computational complexity. To avoid this problem, it is necessary to store partially parsed results of the matching, thereby eliminating duplicate work. This is the same technique that has been widely used in dynamic programming, as described in Chapter 8. Since chart parsing does not need to be from left to right, it is more efficient than the graph search algorithm discussed in Chapter 12, which can be used to parse the input sentence from left to right.

A data structure, called a *chart,* is used to allow the parser to store the partial results of the matching. The chart data structure maintains not only the records of all the constituents derived from the sentence so far in the parse tree, but also the records of rules that have matched partially but are still incomplete. These are called *active arcs*. Here, matches are always considered from the point of view of some *active constituents*, which represent the

subparts that the input sentence can be divided into according to the rewrite rules. Active constituents are stored in a data structure called an *agenda*. To find grammar rules that match a string involving the active constituent, we need to identify rules that start with the active constituent or rules that have already been started by earlier active constituents and require the current constituent to complete the rule or to extend the rule. The basic operation of a chart-based parser involves combining these partially matched records (active arcs) with a completed constituent to form either a new completed constituent or a new partially matched (but incomplete) constituent that is an extension of the original partially matched constituent. Just like the graph search algorithm, we can use either a depth-first or breadth-first search strategy, depending on how the agenda is implemented. If we use probabilities or other heuristics, we take the best-first strategy discussed in Chapter 12 to select constituents from the agenda. The chart-parser process is defined more precisely in Algorithm 11.1. It is possible to combine both top-down and bottom-up. The major difference is how the constituents are used.

ALGORITHM 11.1: *A BOTTOM-UP CHART PARSER*

Step1: Initialization: Define a list called chart to store active arcs, and a list called an agenda to store active constituents until they are added to the chart.
Step 2: Repeat: Repeat Step 2 to 7 until there is no input left.
Step 3: Push and pop the agenda: If the agenda is empty, look up the interpretations of the next word in the input and push them to the agenda. Pop a constituent *C* from the agenda. If *C* corresponds to position from w_i to w_j of the input sentence, we denote it *C[i,j]*.
Step 4: Add *C* to the chart: Insert *C[i,j]* into the chart.
Step 5: Add key-marked active arcs to the chart: For each rule in the grammar of the form $X \rightarrow C\ Y$, add to the chart an active arc (partially matched constituent) of the form $X[i,j] \rightarrow °CY$, where ° denotes the critical position called the key that indicates that everything before ° has been seen, but things after ° are yet to be matched (incomplete constituent).
Step 6: Move ° forward: For any active arc of the form $X[1,j] \rightarrow Y...°C...Z$ (everything before w_i) in the chart, add a new active arc of the form $X[1,j] \rightarrow Y...C°...Z$ to the chart.
Step 7: Add new constituents to the agenda: For any active arc of the form $X[1,l] \rightarrow Y...°C$, add a new constituent of type *X[1,j]* to the agenda.
Step 8: Exit: If *S*[1,*n*] is in the chart, where *n* is the length of the input sentence, we can exit successfully unless we want to find all possible interpretations of the sentence. The chart may contain many *S* structures covering the entire set of positions.

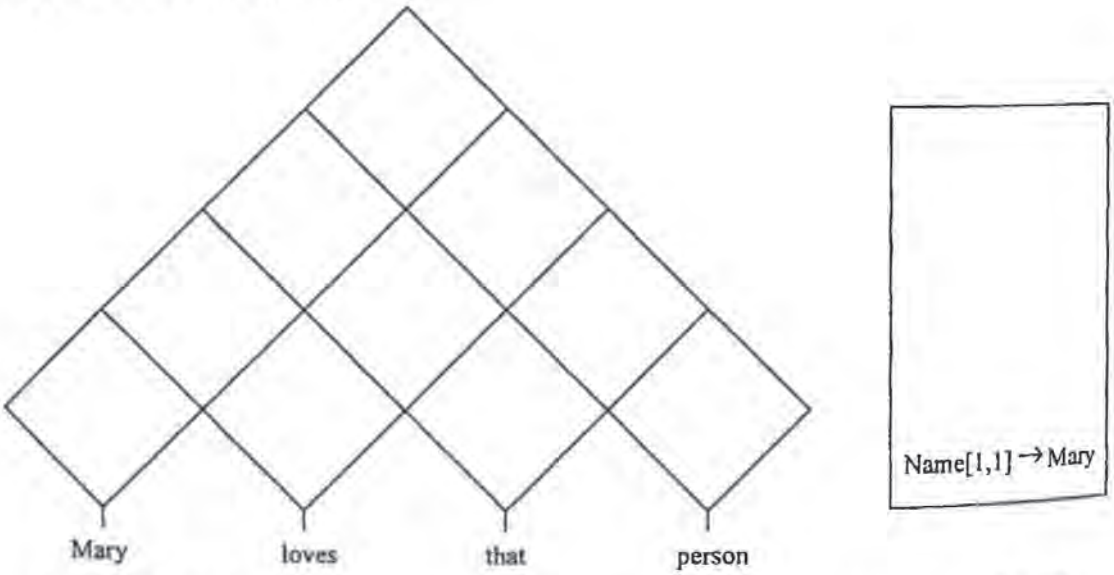
Let us look at an example to see how the chart parser parses the sentence *Mary loves that person* using the grammar specified in Figure 11.1. We first create the chart and agenda data structure as illustrated in Figure 11.2 (a), in which the leaves of the tree-like chart data structure corresponds to the position of each input word. The parent of each block in the chart covers from the position of the left child's corresponding starting word position to the right child's corresponding ending word position. Thus, the root block in the chart covers the whole sentence from the first word *Mary* to the last word *person*. The chart parser scans

through the input words to match against possible rewrite rules in the grammar. For the first word, the rule *Name→Mary* can be matched, so it is added to the agenda according to Step 3 in Algorithm 11.1. In Step 4, *Name→Mary* is added to the chart from the agenda. After the word Mary is processed, we have *Name→Mary, NP→Name*, and *S→NP°VP* in the chart, as illustrated in Figure 11.2 (b). *NP°VP* in the chart indicates that ° has reached the point at which everything before ° has been matched (in this case *Mary* matched *NP*) but everything after ° is yet to be parsed. The completed parsed chart is illustrated in Figure 11.2 (c).

A parser may assign one or more parsed structures to the sentence in the language it defines. If any sentence is assigned more than one such structure, the grammar is said to be ambiguous. Spoken language is, of course, ambiguous by nature.[3] For example, we can have a sentence like *Mary sold the student bags*. It is unclear whether *student* should be the modifier for *bags* or whether it means that *Mary* sold the *bags* to *the student*.

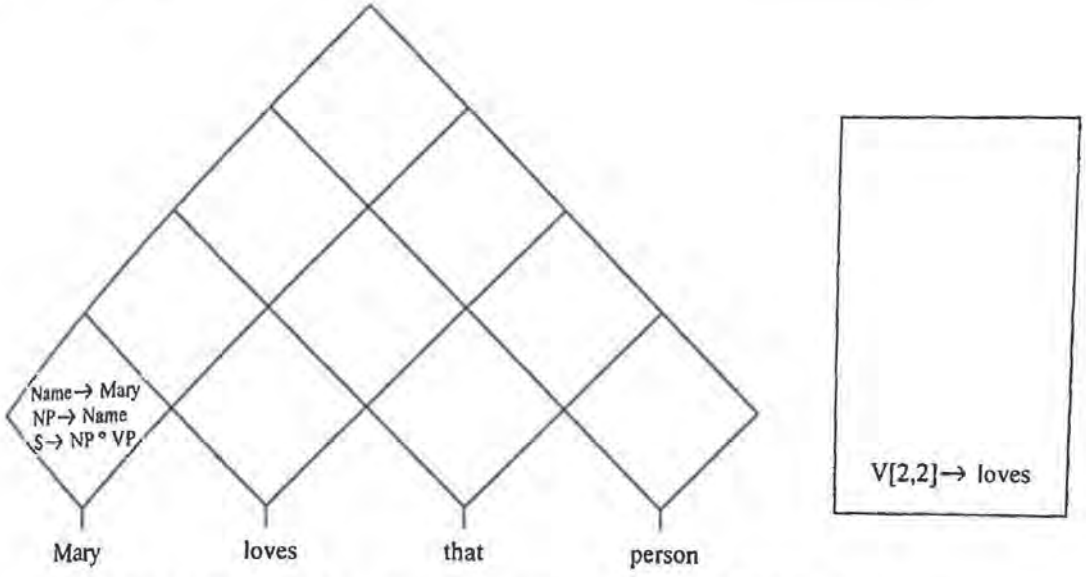
Chart parsers can be fairly efficient simply because the same constituent is never constructed more than once. In the worst case, the chart parser builds every possible constituent between every possible pair of positions, leading to the worst-case computational complexity of $O(n^3)$, where n is the length of the input sentence. This is still far more efficient than a straightforward brute-force search.

In many practical tasks, we need only a partial parse or shallow parse of the input sentence. You can use cascades of finite-state automata instead of CFGs. Relying on simple finite-state automata rather than full parsing makes such systems more efficient, although finite-state systems cannot model certain kinds of recursive rules, so that efficiency is traded for a certain lack of coverage.

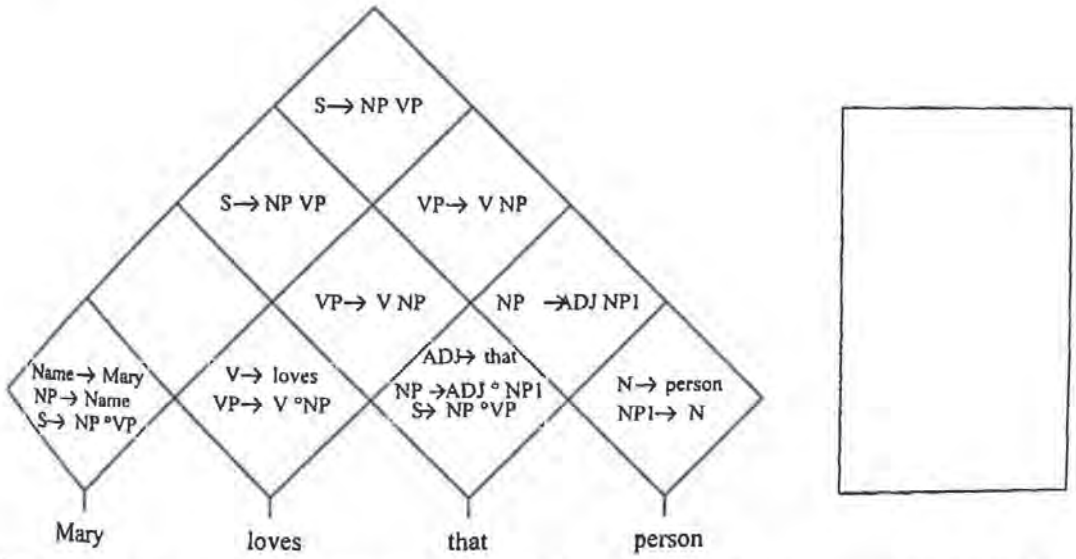


(a) The chart is illustrated on the left, and the agenda is on the right. The agenda now has one rule in it according to Step 3, since the agenda is empty.

[3] The same parse tree can also mean multiple things, so a parse tree itself does not define meaning. "*Mary loves that person*" could be sarcastic and mean something different.



(b) After Mary, the chart now has rules *Name→Mary, NP→Name*, and *S→NP°VP*.



(c) The chart after the whole sentence is parsed. S→ NP VP covers the whole sentence, indicating that the sentence is parsed successfully by the grammar.

Figure 11.2 An example of a chart parser with the grammar illustrated in Figure 11.1. Parts (a) and (b) show the initial chart and agenda to parse the first word; part (c) shows the chart after the sentence is completely parsed.

11.2. STOCHASTIC LANGUAGE MODELS

Stochastic language models (SLM) take a probabilistic viewpoint of language modeling. We need to accurately estimate the probability $P(\mathbf{W})$ for a given word sequence $\mathbf{W} = w_1 w_2 ... w_n$. In the formal language theory discussed in Section 11.1, $P(\mathbf{W})$ can be regarded as 1 or 0 if the word sequence is accepted or rejected, respectively, by the grammar. This may be inappropriate for spoken language systems, since the grammar itself is unlikely to have a complete coverage, not to mention that spoken language is often ungrammatical in real conversational applications.

The key goal of SLM is to provide adequate probabilistic information so that likely word sequences should have a higher probability. This not only makes speech recognition more accurate but also helps to dramatically constrain the search space for speech recognition (see Chapters 12 and 13). Notice that SLM can have a wide coverage on all the possible word sequences, since probabilities are used to differentiate different word sequences. The most widely used SLM is the so call n-gram model discussed in this chapter. In fact, the CFG can be augmented as the bridge between the n-gram and the formal grammar if we can incorporate probabilities into the production rules, as discussed in the next section.

11.2.1. Probabilistic Context-Free Grammars

The CFG can be augmented with probability for each production rule. The advantages of probabilistic CFGs (PCFGs) lie in their ability to more accurately capture the embedded usage structure of spoken language to minimize syntactic ambiguity. The use of probability becomes increasingly important to discriminate many competing choices when the number of rules is large.

In the PCFG, we have to address the parallel problems we discussed for HMMs in Chapter 8. The *recognition problem* is concerned with the computation of the probability of the start symbol S generating the word sequence $\mathbf{W} = w_1, w_2, ... w_T$, given the grammar G:

$$P(S \Rightarrow \mathbf{W}|G) \tag{11.1}$$

where $\Rightarrow$ denotes a derivation sequence consisting of one or more steps. This is equivalent to the chart parser augmented with probabilities, as discussed in Section 11.1.2.2.

The *training problem* is concerned with determining a set of rules G based on the training corpus and estimating the probability of each rule. If the set of rules is fixed, the simplest approach to deriving these probabilities is to count the number of times each rule is used in a corpus containing parsed sentences. We denote the probability of a rule $A \rightarrow \alpha$ by $P(A \rightarrow \alpha|G)$. For instance, if there are m rules for left-hand side non-terminal node A: $A \rightarrow \alpha_1, A \rightarrow \alpha_2, ... A \rightarrow \alpha_m$, we can estimate the probability of these rules as follows:

$$P(A \rightarrow \alpha_j | G) = C(A \rightarrow \alpha_j) / \sum_{i=1}^{m} C(A \rightarrow \alpha_i) \tag{11.2}$$

where $C(.)$ denotes the number of times each rule is used.

When you have hand-annotated corpora, you can use the maximum likelihood estimation as illustrated by Eq. (11.2) to derive the probabilities. When you don't have hand-annotated corpora, you can extend the EM algorithm (see Chapter 4) to derive these probabilities. The algorithm is also known as the *inside-outside* algorithm. As we discussed in Chapter 8, you can develop algorithms similar to the Viterbi algorithm to find the most likely parse tree that could have generated the sequence of words $P(\mathbf{W})$ after these probabilities are estimated.

We can make certain independence assumptions about rule usage. Namely, we assume that the probability of a constituent being derived by a rule is independent of how the constituent is used as a subconstituent. For instance, we assume that the probabilities of *NP* rules are the same no matter whether the *NP* is used for the subject or the object of a verb, although the assumptions are not valid in many cases. More specifically, let the word sequence $\mathbf{W}=w_1 w_2 \ldots w_T$ be generated by a PCFG G, with rules in Chomsky normal form as discussed in Section 11.1.1:

$$A_i \rightarrow A_m A_n \text{ and } A_i \rightarrow w_l \tag{11.3}$$

where A_m and A_n are two possible non-terminals that expand A_i at different locations. The probability for these rules must satisfy the following constraint:

$$\sum_{m,n} P(A_i \rightarrow A_m A_n \mid G) + \sum_{l} P(A_i \rightarrow w_l \mid G) = 1, \quad \text{for all } i \tag{11.4}$$

Equation (11.4) simply means that all non-terminals can generate either pairs of non-terminal symbols or a single terminal symbol, and all these production rules should satisfy the probability constraint. Analogous to the HMM forward and backward probabilities discussed in Chapter 8, we can define the inside and outside probabilities to facilitate the estimation of these probabilities from the training data.

A non-terminal symbol A_i can generate a sequence of words $w_j w_{j+1} \ldots w_k$; we define the probability of $Inside(j, A_i, k) = P(A_i \Rightarrow w_j w_{j+1} \ldots w_k \mid G)$ as the *inside constituent probability*, since it assigns a probability to the word sequence inside the constituent. The inside probability can be computed recursively. When only one word is emitted, the transition rule of the form $A_i \rightarrow w_m$ applies. When there is more than one word, rules of the form $A_i \rightarrow A_m A_n$ must apply. The inside probability of $inside(j, A_i, k)$ can be expressed recursively as follows:

$$\begin{aligned} inside(j, A_i, k) &= P(A_i \Rightarrow w_j w_{j+1} \ldots w_k) \\ &= \sum_{n,m} \sum_{l=j}^{k-1} P(A_i \rightarrow A_m A_n) P(A_m \Rightarrow w_j \ldots w_l) P(A_n \Rightarrow w_{l+1} \ldots w_k) \\ &= \sum_{n,m} \sum_{l=j}^{k-1} P(A_l \rightarrow A_m A_n) inside(j, A_m, l) inside(l+1, A_n, k) \end{aligned} \tag{11.5}$$

The inside probability is the sum of the probabilities of all derivations for the section over the span of j to k. One possible derivation of the form can be drawn as a parse tree shown in Figure 11.3.

Another useful probability is the *outside* probability for a non-terminal node A_i covering w_s to w_t, in which they can be derived from the start symbol S, as illustrated in Figure 11.4, together with the rest of the words in the sentence:

$$outside(s, A_i, t) = P(S \Rightarrow w_1 \ldots w_{s-1}\ A_i\ w_{t+1} \ldots w_T) \tag{11.6}$$

After the inside probabilities are computed bottom-up, we can compute the outside probabilities top-down. For each non-terminal symbol A_j, there are one of two possible configurations $A_m \rightarrow A_n A_j$ or $A_m \rightarrow A_j A_n$ as illustrated in Figure 11.5. Thus, we need to consider all the possible derivations of these two forms as follows:

$$\begin{aligned} outside(s, A_j, t) &= P(S \Rightarrow w_1 \ldots w_{s-1}\ A_j\ w_{t+1} \ldots w_T) \\ &= \sum_{m,n} \left\{ \begin{array}{l} \sum_{l=1}^{s-1} P(A_m \rightarrow A_n A_j) P(A_n \Rightarrow w_l \ldots w_{s-1}) P(S \Rightarrow w_1 \ldots w_{l-1}\ A_m\ w_{t+1} \ldots w_T) + \\ + \sum_{l=t+1}^{T} P(A_m \rightarrow A_j A_n) P(A_n \Rightarrow w_{t+1} \ldots w_l) P(S \Rightarrow w_1 \ldots w_{s-1}\ A_m\ w_{t+1} \ldots w_T) \end{array} \right\} \\ &= \sum_{m,n} \left\{ \begin{array}{l} \sum_{l=1}^{s-1} P(A_m \rightarrow A_n A_j) inside(l, A_n, s-1) outside(l, A_m, t) + \\ + \sum_{l=t+1}^{T} P(A_m \rightarrow A_j A_n) inside(t+1, A_n, l) outside(s, A_m, l) \end{array} \right\} \end{aligned} \tag{11.7}$$

The inside and outside probabilities are used to compute the sentence probability as follows:

$$P(S \Rightarrow w_1 \ldots w_T) = \sum_j inside(s, A_j, t) outside(s, A_j, t) \qquad \text{for any } s \le t \tag{11.8}$$

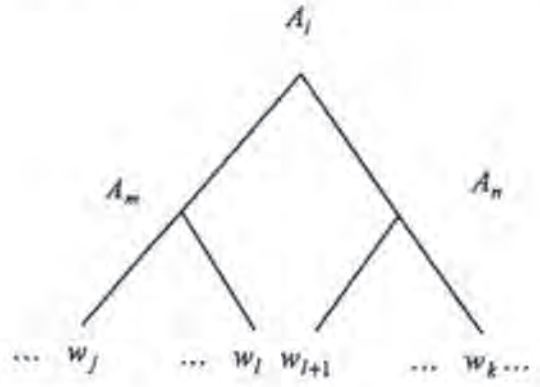


Figure 11.3 Inside probability is computed recursively as sum of all the derivations.

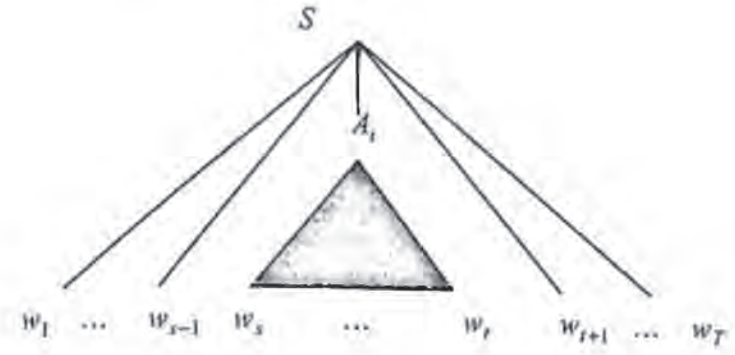


Figure 11.4 Definition of the outside probability.

Since $outside(1, A_i, T)$ is equal to 1 for the starting symbol only, the probability for the whole sentence can be conveniently computed using the inside probability alone as

$$P(S \Rightarrow \mathbf{W}|G) = inside(1, S, T) \tag{11.9}$$

We are interested in the probability that a particular rule, $A_i \rightarrow A_m A_n$ is used to cover a span $w_s \ldots w_t$, given the sentence and the grammar:

$$\xi(i,m,n,s,t) = P(A_i \Rightarrow w_s \ldots w_t, A_i \rightarrow A_m A_n \mid S \Rightarrow \mathbf{W}, G)$$
$$= \frac{1}{P(S \Rightarrow \mathbf{W} \mid G)} \sum_{k=s}^{t-1} P(A_i \rightarrow A_m A_n \mid G) inside(s, A_m, k) inside(k+1, A_n, t) outside(s, A_i, t) \tag{11.10}$$

These conditional probabilities form the basis of the inside-outside algorithm, which is similar to the forward-backward algorithm discussed in Chapter 8. We can start with some initial probability estimates. For each sentence of training data, we determine the inside and outside probabilities in order to compute, for each production rule, how likely it is that the production rule is used as part of the derivation of that sentence. This gives us the number of counts for each production rule in each sentence. Summing these counts across sentences gives us an estimate of the total number of times each production rule is used to produce the

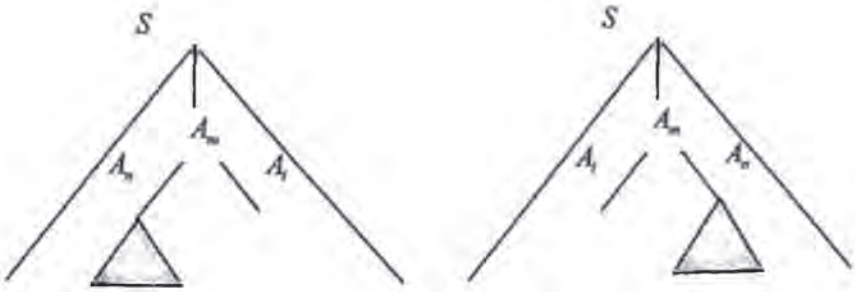


Figure 11.5 Two possible configurations for a non-terminal node A_i.

sentences in the training corpus. Dividing by the total counts of productions used for each non-terminal gives us a new estimate of the probability of the production in the MLE framework. For example, we have:

$$P(A_i \rightarrow A_m A_n | G) = \frac{\sum_{s=1}^{T-1} \sum_{t=s+1}^{T} \xi(i,m,n,s,t)}{\sum_{m,n} \sum_{s=1}^{T-1} \sum_{t=s+1}^{T} \xi(i,m,n,s,t)} \tag{11.11}$$

In a similar manner, we can estimate $P(A_i \rightarrow w_m | G)$. It is also possible to let the inside-outside algorithm formulate all the possible grammar production rules so that we can select rules with sufficient probability values. If there is no constraint, we may have too many *greedy symbols* that serve as possible non-terminals. In addition, the algorithm is guaranteed only to find a local maximum. It is often necessary to use prior knowledge about the task and the grammar to impose strong constraints to avoid these two problems. The chart parser discussed in Section 11.1.2 can be modified to accommodate PCFGs [29, 45].

One problem with the PCFG is that it assumes that the expansion of any one non-terminal is independent of the expansion of other non-terminals. Thus each PCFG rule probability is multiplied together without considering the location of the node in the parse tree. This is against our intuition since there is a strong tendency toward the context-dependent expansion. Another problem is its lack of sensitivity to words, although lexical information plays an important role in selecting the correct parsing of an ambiguous prepositional phrase attachment. In the PCFG, lexical information can only be represented via the probability of pre-terminal nodes, such as verb or noun, to be expanded lexically. You can add lexical dependencies to PCFGs and make PCFG probabilities more sensitive to surrounding syntactic structure [6, 11, 19, 31, 45].

11.2.2. *N*-gram Language Models

As covered earlier, a language model can be formulated as a probability distribution $P(\mathbf{W})$ over word strings $\mathbf{W}$ that reflects how frequently a string $\mathbf{W}$ occurs as a sentence. For example, for a language model describing spoken language, we might have *P(hi)* = 0.01, since perhaps one out of every hundred sentences a person speaks is *hi*. On the other hand, we would have *P(lid gallops Changsha pop)* = 0, since it is extremely unlikely anyone would utter such a strange string.

$P(\mathbf{W})$ can be decomposed as

$$\begin{aligned} P(\mathbf{W}) &= P(w_1, w_2, \ldots, w_n) \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\cdots P(w_n|w_1, w_2, \ldots, w_{n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_1, w_2, \ldots, w_{i-1}) \end{aligned} \tag{11.12}$$

where $P(w_i|w_1, w_2, \ldots, w_{i-1})$ is the probability that w_i will follow, given that the word sequence $w_1, w_2, \ldots, w_{i-1}$ was presented previously. In Eq. (11.12), the choice of w_i thus depends on the entire past history of the input. For a vocabulary of size v there are v^{i-1} different histories and so, to specify $P(w_i|w_1, w_2, \ldots, w_{i-1})$ completely, v^i values would have to be estimated. In reality, the probabilities $P(w_i|w_1, w_2, \ldots, w_{i-1})$ are impossible to estimate for even moderate values of i, since most histories $w_1, w_2, \ldots, w_{i-1}$ are unique or have occurred only a few times. A practical solution to the above problems is to assume that $P(w_i|w_1, w_2, \ldots, w_{i-1})$ depends only on some equivalence classes. The equivalence class can be simply based on the several previous words $w_{i-N+1}, w_{i-N+2}, \ldots, w_{i-1}$. This leads to an *n-gram* language model. If the word depends on the previous two words, we have a *trigram*: $P(w_i|w_{i-2}, w_{i-1})$. Similarly, we can have *unigram*: $P(w_i)$, or *bigram*: $P(w_i|w_{i-1})$ language models. The trigram is particularly powerful, as most words have a strong dependence on the previous two words, and it can be estimated reasonably well with an attainable corpus.

In bigram models, we make the approximation that the probability of a word depends only on the identity of the immediately preceding word. To make $P(w_i|w_{i-1})$ meaningful for $i = 1$, we pad the *beginning of the sentence* with a distinguished token <s>; that is, we pretend $w_0 = $ <s>. In addition, to make the sum of the probabilities of all strings equal 1, it is necessary to place a distinguished token </s> at the *end of the sentence*. For example, to calculate *P(Mary loves that person)* we would take

$$P(\textit{Mary loves that person}) =$$
$$P(\textit{Mary}|\langle s\rangle)P(\textit{loves}|\textit{Mary})P(\textit{that}|\textit{loves})P(\textit{person}|\textit{that})P(\langle/s\rangle|\textit{person})$$

To estimate $P(w_i|w_{i-1})$, the frequency with which the word w_i occurs given that the last word is w_{i-1}, we simply count how often the sequence (w_{i-1}, w_i) occurs in some text and normalize the count by the number of times w_{i-1} occurs.

In general, for a trigram model, the probability of a word depends on the two preceding words. The trigram can be estimated by observing the frequencies or counts of the word pair $C(w_{i-2}, w_{i-1})$ and triplet $C(w_{i-2}, w_{i-1}, w_i)$ as follows:

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})} \tag{11.13}$$

The text available for building a model is called a training corpus. For *n*-gram models, the amount of training data used is typically many millions of words. The estimate of Eq. (11.13) is based on the maximum likelihood principle, because this assignment of probabilities yields the trigram model that assigns the highest probability to the training data of all possible trigram models.

We sometimes refer to the value n of an n-gram model as its order. This terminology comes from the area of Markov models, of which n-gram models are an instance. In particular, an n-gram model can be interpreted as a Markov model of order n-1.

Consider a small example. Let our training data S be comprised of the three sentences *John read her book. I read a different book. John read a book by Mulan*. and let us calculate *P(John read a book)* for the maximum likelihood bigram model. We have

$$P(John|<s>)=\frac{C(<s>,John)}{C(<s>)}=\frac{2}{3}$$

$$P(read|John)=\frac{C(John,read)}{C(John)}=\frac{2}{2}$$

$$P(a|read)=\frac{C(read,a)}{C(read)}=\frac{2}{3}$$

$$P(book|a)=\frac{C(a,book)}{C(a)}=\frac{1}{2}$$

$$P(</s>|book)=\frac{C(book,</s>)}{C(book)}=\frac{2}{3}$$

These trigram probabilities help us estimate the probability for the sentence as:

$$\begin{aligned}&P(John\ read\ a\ book)\\&=P(John|<s>)P(read|John)P(a|read)P(book|a)P(</s>|book)\\&\approx 0.148\end{aligned}\qquad(11.14)$$

If these three sentences are all the data we have available to use in training our language model, the model is unlikely to generalize well to new sentences. For example, the sentence "*Mulan read her book*" should have a reasonable probability, but the trigram will give it a zero probability simply because we do not have a reliable estimate for *P(read|Mulan)*.

Unlike linguistics, grammaticality is not a strong constraint in the *n*-gram language model. Even though the string is ungrammatical, we may still assign it a high probability if *n* is small.

11.3. Complexity Measure of Language Models

Language can be thought of as an information source whose outputs are words w_i belonging to the vocabulary of the language. The most common metric for evaluating a language model is the word recognition error rate, which requires the participation of a speech recognition system. Alternatively, we can measure the probability that the language model assigns to test word strings without involving speech recognition systems. This is the derivative measure of cross-entropy known as test-set *perplexity*.

The measure of cross-entropy is discussed in Chapter 3. Given a language model that assigns probability $P(\mathbf{W})$ to a word sequence $\mathbf{W}$, we can derive a compression algorithm that encodes the text $\mathbf{W}$ using $-\log_2 P(\mathbf{W})$ bits. The cross-entropy $H(\mathbf{W})$ of a model

$P(w_i|w_{i-n+1}\cdots w_{i-1})$ on data $\mathbf{W}$, with a sufficiently long word sequence, can be simply approximated as

$$H(\mathbf{W}) = -\frac{1}{N_w}\log_2 P(\mathbf{W}) \tag{11.15}$$

where N_w is the length of the text $\mathbf{W}$ measured in words.

The perplexity $PP(\mathbf{W})$ of a language model $P(\mathbf{W})$ is defined as the reciprocal of the (geometric) average probability assigned by the model to each word in the test set $\mathbf{W}$. This is a measure, related to cross-entropy, known as test-set perplexity:

$$PP(\mathbf{W}) = 2^{H(\mathbf{W})} \tag{11.16}$$

The perplexity can be roughly interpreted as the geometric mean of the branching factor of the text when presented to the language model. The perplexity defined in Eq. (11.16) has two key parameters: a language model and a word sequence. The test-set[4] perplexity evaluates the generalization capability of the language model. The training-set perplexity measures how the language model fits the training data, like the likelihood. It is generally true that lower perplexity correlates with better recognition performance. This is because the perplexity is essentially a statistically weighted word branching measure on the test set. The higher the perplexity, the more branches the speech recognizer needs to consider statistically.

While the perplexity [Eqs. (11.16) and (11.15)] is easy to calculate for the n-gram [Eq. (11.12)], it is slightly more complicated to compute for a probabilistic CFG. We can first parse the word sequence and use Eq. (11.9) to compute $P(\mathbf{W})$ for the test-set perplexity. The perplexity can also be applied to nonstochastic models such as CFGs. We can assume they have a uniform distribution in computing $P(\mathbf{W})$.

A language with higher perplexity means that the number of words branching from a previous word is larger on average. In this sense, perplexity is an indication of the complexity of the language if we have an accurate estimate of $P(\mathbf{W})$. For a given language, the difference between the perplexity of a language model and the true perplexity of the language is an indication of the quality of the model. The perplexity of a particular language model can change dramatically in terms of the vocabulary size, the number of states of grammar rules, and the estimated probabilities. A language model with perplexity X has roughly the same difficulty as another language model in which every word can be followed by X different words with equal probabilities. Therefore, in the task of continuous digit recognition, the perplexity is 10. Clearly, lower perplexity will generally have less confusion in recognition. Typical perplexities yielded by n-gram models on English text range from about 50 to almost 1000 (corresponding to cross-entropies from about 6 to 10 bits/word), depending on the type of text. In the task of 5,000-word continuous speech recognition for the *Wall Street Journal*, the test-set perplexities of the trigram grammar and the bigram grammar are re-

[4] We often distinguish between the word sequence from the unseen test data and that from the training data to derive the language model.

ported to be about 128 and 176 respectively.[5] In the tasks of 2000-word conversational Air Travel Information System (ATIS), the test-set perplexity of the word trigram model is typically less than 20.

Since perplexity does not take into account acoustic confusability, we eventually have to measure speech recognition accuracy. For example, if the vocabulary of a speech recognizer contains the E-set of English alphabet: *B, C, D, E, G, P*,and *T*, we can define a CFG that has a low perplexity value of 7. Such a low perplexity does not guarantee we will have good recognition performance, because of the intrinsic acoustic confusability of the E-set.

11.4. *N*-GRAM SMOOTHING

One of the key problems in *n*-gram modeling is the inherent data sparseness of real training data. If the training corpus is not large enough, many actually possible word successions may not be well observed, leading to many extremely small probabilities. For example, with several-million-word collections of English text, more than 50% of trigrams occur only once, and more than 80% of trigrams occur less than five times. Smoothing is critical to make estimated probabilities robust for unseen data. If we consider the sentence *Mulan read a book* in the example we discussed in Section 11.2.2, we have:

$$P(read|\, Mulan) = \frac{C(Mulan, read)}{\sum_{w} C(Mulan, w)} = \frac{0}{1}$$

giving us $P(Mulan\ read\ a\ book) = 0$.

Obviously, this is an underestimate for the probability of "*Mulan read a book*" since there is *some* probability that the sentence occurs in some test set. To show why it is important to give this probability a nonzero value, we turn to the primary application for language models, speech recognition. In speech recognition, if $P(\mathbf{W})$ is zero, the string **W** will never be considered as a possible transcription, regardless of how unambiguous the acoustic signal is. Thus, whenever a string W such that $P(\mathbf{W}) = 0$ occurs during a speech recognition task, an error will be made. Assigning all strings a nonzero probability helps prevent errors in speech recognition. This is the core issue of smoothing. Smoothing techniques adjust the maximum likelihood estimate of probabilities to produce more robust probabilities for unseen data, although the likelihood for the training data may be hurt slightly.

The name smoothing comes from the fact that these techniques tend to make distributions flatter, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Not only do smoothing methods generally prevent zero probabilities,

[5] Some experimental results show that the test-set perplexities for different languages are comparable. For example, French, English, Italian and German have a bigram test-set perplexity in the range of 95 to 133 for newspaper corpora. Italian has a much higher perplexity reduction (a factor of 2) from bigram to trigram because of the high number of function words. The trigram perplexity of Italian is among the lowest in these languages [34].

but they also attempt to improve the accuracy of the model as a whole. Whenever a probability is estimated from few counts, smoothing has the potential to significantly improve the estimation so that it has better generalization capability.

To give an example, one simple smoothing technique is to pretend each bigram occurs once more than it actually does, yielding

$$P(w_i|w_{i-1}) = \frac{1+C(w_{i-1},w_i)}{\sum_{w_i}(1+C(w_{i-1},w_i))} = \frac{1+C(w_{i-1},w_i)}{V+\sum_{w_i}C(w_{i-1},w_i)} \tag{11.17}$$

where V is the size of the vocabulary. In practice, vocabularies are typically fixed to be tens of thousands of words or less. All words not in the vocabulary are mapped to a single word, usually called the *unknown word*. Let us reconsider the previous example using this new distribution, and let us take our vocabulary V to be the set of all words occurring in the training data S, so that we have $V = 11$ (with both <s> and </s>).

For the sentence *John read a book*, we now have

$$\begin{aligned} &P(John\ read\ a\ book) \\ &= P(John|</s>)P(read|John)P(a|read)P(book|a)P(</s>|book) \\ &\approx 0.00035 \end{aligned} \tag{11.18}$$

In other words, we estimate that the sentence *John read a book* occurs about once every three thousand sentences. This is more reasonable than the maximum likelihood estimate of 0.148 of Eq. (11.14). For the sentence *Mulan read a book*, we have

$$\begin{aligned} &P(Mulan\ read\ a\ book) \\ &= P(Mulan|</s>)P(read|Mulan)P(a|read)P(book|a)P(</s>|book) \\ &\approx 0.000084 \end{aligned} \tag{11.19}$$

Again, this is more reasonable than the zero probability assigned by the maximum likelihood model. In general, most existing smoothing algorithms can be described with the following equation:

$$\begin{aligned} &P_{smooth}(w_i \mid w_{i-n+1}...w_{i-1}) \\ &= \begin{cases} \alpha(w_i \mid w_{i-n+1}...w_{i-1}) & \text{if } C(w_{i-n+1}...w_i)>0 \\ \gamma(w_{i-n+1}...w_{i-1})P_{smooth}(w_i \mid w_{i-n+2}...w_{i-1}) & \text{if } C(w_{i-n+1}...w_i)=0 \end{cases} \end{aligned} \tag{11.20}$$

That is, if an n-gram has a nonzero count we use the distribution $\alpha(w_i|w_{i-n+1}...w_{i-1})$. Otherwise, we backoff to the lower-order n-gram distribution $P_{smooth}(w_i|w_{i-n+2}...w_{i-1})$, where the scaling factor $\gamma(w_{i-n+1}...w_{i-1})$ is chosen to make the conditional distribution sum to one. We refer to algorithms that fall directly in this framework as *backoff models*.

Several other smoothing algorithms are expressed as the linear interpolation of higher- and lower-order n-gram models as:

$$P_{smooth}(w_i \mid w_{i-n+1} \ldots w_{i-1}) = \lambda P_{ML}(w_i \mid w_{i-n+1} \ldots w_{i-1}) + (1-\lambda) P_{smooth}(w_i \mid w_{i-n+2} \ldots w_{i-1}) \tag{11.21}$$

where λ is the interpolation weight that depends on $w_{i-n+1} \ldots w_{i-1}$. We refer to models of this form as interpolated models.

The key difference between backoff and interpolated models is that for the probability of n-grams with nonzero counts, interpolated models use information from lower-order distributions while backoff models do not. In both backoff and interpolated models, lower-order distributions are used in determining the probability of n-grams with zero counts. Now, we discuss several backoff and interpolated smoothing methods. Performance comparison of these techniques in real speech recognition applications is discussed in Section 11.4.4.

11.4.1. Deleted Interpolation Smoothing

Consider the case of constructing a bigram model on training data where we have that *C(enliven you)* = 0 *and C(enliven thou)* = 0. Then, according to both additive smoothing of Eq. (11.17), we have *P(you|enliven) = P(thou|enliven)*. However, intuitively we should have *P(you|enliven) > P(thou|enliven)*, because the word *you* is much more common than the word *thou* in modern English. To capture this behavior, we can interpolate the bigram model with a unigram model. A unigram model conditions the probability of a word on no other words, and just reflects the frequency of that word in text. We can linearly interpolate a bigram model and a unigram model as follows:

$$P_I(w_i \mid w_{i-1}) = \lambda P(w_i \mid w_{i-1}) + (1-\lambda) P(w_i) \tag{11.22}$$

where $0 \le \lambda \le 1$. Because *P(you|enliven) = P(thou|enliven)*=0 while presumably *P(you) > P(thou)*, we will have that $P_I(you \mid enliven) > P_I(thou \mid enliven)$ as desired.

In general, it is useful to interpolate higher-order n-gram models with lower-order n-gram models, because when there is insufficient data to estimate a probability in the higher-order model, the lower-order model can often provide useful information. An elegant way of performing this interpolation is given as follows

$$P_I(w_i \mid w_{i-n+1} \cdots w_{i-1}) = \lambda_{w_{i-n+1} \ldots w_{i-1}} P(w_i \mid w_{i-n+1} \cdots w_{i-1}) + (1-\lambda_{w_{i-n+1} \ldots w_{i-1}}) P_I(w_i \mid w_{i-n+2} \cdots w_{i-1}) \tag{11.23}$$

That is, the nth-order smoothed model is defined *recursively* as a linear interpolation between the nth-order maximum likelihood model and the $(n-1)$th-order smoothed model. To end the recursion, we can take the smoothed first-order model to be the maximum likeli-

hood distribution (unigram), or we can take the smoothed zeroth-order model to be the uniform distribution. Given a fixed $P(w_i|w_{i-n+1}\cdots w_{i-1})$, it is possible to search efficiently for the interpolation parameters using the deleted interpolation method discussed in Chapter 9.

Notice that the optimal $\lambda_{w_{i-n+1}\cdots w_{i-1}}$ is different for different histories $w_{i-n+1}\cdots w_{i-1}$. For example, for a context we have seen thousands of times, a high λ will be suitable, since the higher-order distribution is very reliable; for a history that has occurred only once, a lower λ is appropriate. Training each parameter $\lambda_{w_{i-n+1}\cdots w_{i-1}}$ independently can be harmful; we need an enormous amount of data to train so many independent parameters accurately. One possibility is to divide the $\lambda_{w_{i-n+1}\cdots w_{i-1}}$ into a moderate number of partitions or buckets, constraining all $\lambda_{w_{i-n+1}\cdots w_{i-1}}$ in the same bucket to have the same value, thereby reducing the number of independent parameters to be estimated. Ideally, we should tie together those $\lambda_{w_{i-n+1}\cdots w_{i-1}}$ that we have a prior reason to believe should have similar values.

11.4.2. Backoff Smoothing

Backoff smoothing is attractive because it is easy to implement for practical speech recognition systems. The Katz backoff model is the canonical example we discuss in this section. It is based on the Good-Turing smoothing principle.

11.4.2.1. Good-Turing Estimates and Katz Smoothing

The Good-Turing estimate is a smoothing technique to deal with infrequent n-grams. It is not used by itself for n-gram smoothing, because it does not include the combination of higher-order models with lower-order models necessary for good performance. However, it is used as a tool in several smoothing techniques. The basic idea is to partition n-grams into groups depending on their frequency (i.e. how many time the n-grams appear in the training data) such that the parameter can be smoothed based on n-gram frequency.

The Good-Turing estimate states that for any n-gram that occurs r times, we should pretend that it occurs r^* times as follows:

$$r^* = (r+1)\frac{n_{r+1}}{n_r} \tag{11.24}$$

where n_r is the number of n-grams that occur exactly r times in the training data. To convert this count to a probability, we just normalize: for an n-gram a with r counts, we take

$$P(a) = \frac{r^*}{N} \tag{11.25}$$

where $N = \sum_{r=0}^{\infty} n_r r^*$. Notice that $N = \sum_{r=0}^{\infty} n_r r^* = \sum_{r=0}^{\infty}(r+1)n_{r+1} = \sum_{r=0}^{\infty} n_r r$, i.e., N is equal to the original number of counts in the distribution [28].

Katz smoothing extends the intuitions of the Good-Turing estimate by adding the combination of higher-order models with lower-order models [38]. Take the bigram as our example, Katz smoothing suggested using the Good-Turing estimate for nonzero counts as follows:

$$C^*(w_{i-1}w_i) = \begin{cases} d_r r & \text{if } r>0 \\ \alpha(w_{i-1})P(w_i) & \text{if } r=0 \end{cases} \tag{11.26}$$

where d_r is approximately equal to r^*/r. That is, all bigrams with a nonzero count r are *discounted* according to a discount ratio d_r, which implies that the counts subtracted from the nonzero counts are distributed among the zero-count bigrams according to the next lower-order distribution, e.g., the unigram model. The value $\alpha(w_{i-1})$ is chosen to equalize the total number of counts in the distribution, i.e., $\sum_{w_i} C^*(w_{i-1}w_i) = \sum_{w_i} C(w_{i-1}w_i)$. The appropriate value for $\alpha(w_{i-1})$ is computed so that the smoothed bigram satisfies the probability constraint:

$$\alpha(w_{i-1}) = \frac{1-\sum_{w_i:C(w_{i-1}w_i)>0} P^*(w_i|w_{i-1})}{\sum_{w_i:C(w_{i-1}w_i)=0} P(w_i)} = \frac{1-\sum_{w_i:C(w_{i-1}w_i)>0} P^*(w_i|w_{i-1})}{1-\sum_{w_i:C(w_{i-1}w_i)>0} P(w_i)} \tag{11.27}$$

To calculate $P^*(w_i|w_{i-1})$ from the corrected count, we just normalize:

$$P^*(w_i \mid w_{i-1}) = \frac{C^*(w_{i-1}w_i)}{\sum_{w_k} C^*(w_{i-1}w_k)} \tag{11.28}$$

In Katz implementation, the d_r are calculated as follows: large counts are taken to be reliable, so they are not discounted. In particular, Katz takes $d_r = 1$ for all $r > k$ for some k, say k in the range of 5 to 8. The discount ratios for the lower counts $r \le k$ are derived from the Good-Turing estimate applied to the global bigram distribution; that is, n_r in Eq. (11.24) denotes the total number of bigrams that occur exactly r times in the training data. These d_r are chosen such that

- the resulting discounts are proportional to the discounts predicted by the Good-Turing estimate, and
- the total number of counts discounted in the global bigram distribution is equal to the total number of counts that should be assigned to bigrams with zero counts according to the Good-Turing estimate.

The first constraint corresponds to the following equation:

$$d_r = \mu \frac{r^*}{r} \tag{11.29}$$

for $r \in \{1,\ldots k\}$ with some constant μ. The Good-Turing estimate predicts that the total mass assigned to bigrams with zero counts is $n_0 \frac{n_1}{n_0} = n_1$, and the second constraint corresponds to the equation

$$\sum_{r=1}^{k} n_r (1-d_r) r = n_1 \tag{11.30}$$

Based on Eq. (11.30), the unique solution is given by:

$$d_r = \frac{\frac{r^*}{r} - \frac{(k+1)n_{k+1}}{n_1}}{1 - \frac{(k+1)n_{k+1}}{n_1}} \tag{11.31}$$

Katz smoothing for higher-order n-gram models is defined analogously. The Katz n-gram backoff model is defined in terms of the Katz (n-1)-gram model. To end the recursion, the Katz unigram model is taken to be the maximum likelihood unigram model. It is usually necessary to smooth n_r when using the Good-Turing estimate, e.g., for those n_r that are very low. However, in Katz smoothing this is not essential because the Good-Turing estimate is used only for small counts $r<=k$, and n_r is generally fairly high for these values of r. The procedure of Katz smoothing can be summarized as in Algorithm 11.2.

In fact, the Katz backoff model can be expressed in terms of the interpolated model defined in Eq. (11.23), in which the interpolation weight is obtained via Eq. (11.26) and (11.27).

ALGORITHM 11.2: *KATZ SMOOTHING*

$$P_{Katz}(w_i \mid w_{i-1}) = \begin{cases} C(w_{i-1}w_i)/C(w_{i-1}) & \text{if } r > k \\ d_r C(w_{i-1}w_i)/C(w_{i-1}) & \text{if } k \geq r > 0 \\ \alpha(w_{i-1})P(w_i) & \text{if } r = 0 \end{cases}$$

where $d_r = \frac{\frac{r^*}{r} - \frac{(k+1)n_{k+1}}{n_1}}{1 - \frac{(k+1)n_{k+1}}{n_1}}$ and $\alpha(w_{i-1}) = \frac{1 - \sum_{w_i : r>0} P_{Katz}(w_i \mid w_{i-1})}{1 - \sum_{w_i : r>0} P(w_i)}$

11.4.2.2. Alternative Backoff Models

In a similar manner to the Katz backoff model, there are other ways to discount the probability mass. For instance, *absolute discounting* involves subtracting a fixed discount $D <= 1$ from each nonzero count. If we express the absolute discounting in term of interpolated models, we have the following:

$$P_{abs}(w_i|w_{i-n+1}\cdots w_{i-1}) = \frac{\max\{C(w_{i-n+1}\cdots w_i)-D,0\}}{\sum_{w_i} C(w_{i-n+1}\cdots w_i)} + (1-\lambda_{w_{i-n+1}\cdots w_{i-1}})P_{abs}(w_i|w_{i-n+2}\cdots w_{i-1}) \quad (11.32)$$

To make this distribution sum to 1, we normalize it to determine $\lambda_{w_{i-n+1}\cdots w_{i-1}}$. Absolute discounting is explained with the Good-Turing estimate. Empirically the average Good-Turing discount $r-r^*$ associated with n-grams of larger counts (r over 3) is largely constant over r.

Consider building a bigram model on data where there exists a word that is very common, say *Francisco*, that occurs only after a single word, say *San*. Since *C(Francisco)* is high, the unigram probability *P(Francisco)* will be high, and an algorithm such as absolute discounting or Katz smoothing assigns a relatively high probability to occurrence of the word *Francisco* after novel bigram histories. However, intuitively this probability should not be high, since in the training data the word *Francisco* follows only a single history. That is, perhaps *Francisco* should receive a low unigram probability, because the only time the word occurs is when the last word is *San*, in which case the bigram probability models its probability well.

Extending this line of reasoning, perhaps the unigram probability used should not be proportional to the number of occurrences of a word, but instead to the number of different words that it follows. To give an intuitive argument, imagine traversing the training data sequentially and building a bigram model on the preceding data to predict the current word. Then, whenever the current bigram does not occur in the preceding data, the unigram probability becomes a large factor in the current bigram probability. If we assign a count to the corresponding unigram whenever such an event occurs, then the number of counts assigned to each unigram is simply the number of different words that it follows. In Kneser-Ney smoothing [40], the lower-order n-gram is not proportional to the number of occurrences of a word, but instead to the number of *different* words that it follows. We summarize the Kneser-Ney backoff model in Algorithm 11.3.

Kneser-Ney smoothing is an extension of other backoff models. Most of the previous models used the lower-order n-grams trained with ML estimation. Kneser-Ney smoothing instead considers a lower-order distribution as a significant factor in the combined model such that they are optimized together with other parameters. To derive the formula, more generally, we express it in terms of the interpolated model specified in Eq. (11.23) as:

$$P_{KN}(w_i \mid w_{i-n+1}...w_{i-1})$$
$$= \frac{\max\{C(w_{i-n+1}...w_i) - D, 0\}}{\sum_{w_i} C(w_{i-n+1}...w_i)} + (1 - \lambda_{w_{i-n+1}...w_{i-1}}) P_{KN}(w_i \mid w_{i-n+2}...w_{i-1}) \quad (11.33)$$

To make this distribution sum to 1, we have:

$$1 - \lambda_{w_{i-n+1}...w_{i-1}} = \frac{D}{\sum_{w_i} C(w_{i-n+1}...w_i)} \mathbb{C}(w_{i-n+1}...w_{i-1}\bullet) \quad (11.34)$$

where $\mathbb{C}(w_{i-n+1}...w_{i-1}\bullet)$ is the number of unique words that follow the history $w_{i-n+1}...w_{i-1}$. This equation enables us to interpolate the lower-order distribution with all words, not just with words that have zero counts in the higher-order distribution.

ALGORITHM 11.3: *KNESER-NEY BIGRAM SMOOTHING*

$$P_{KN}(w_i \mid w_{i-1}) = \begin{cases} \dfrac{\max\{C(w_{i-1}w_i) - D, 0\}}{C(w_{i-1})} & \text{if } C(w_{i-1}w_i) > 0 \\ \alpha(w_{i-1}) P_{KN}(w_i) & \text{otherwise} \end{cases}$$

where $P_{KN}(w_i) = \mathbb{C}(\bullet w_i) / \sum_{w_i} \mathbb{C}(\bullet w_i)$, $\mathbb{C}(\bullet w_i)$ is the number of unique words preceding w_i.

$\alpha(w_{i-1})$ is chosen to make the distribution sum to 1 so that we have:

$$\alpha(w_{i-1}) = \frac{1 - \sum_{w_i : C(w_{i-1}w_i) > 0} \dfrac{\max\{C(w_{i-1}w_i) - D, 0\}}{C(w_{i-1})}}{1 - \sum_{w_i : C(w_{i-1}w_i) > 0} P_{KN}(w_i)}$$

Now, take the bigram case as an example. We need to find a unigram distribution $P_{KN}(w_i)$ such that the marginal of the bigram smoothed distributions should match the marginal of the training data:

$$\frac{C(w_i)}{\sum_{w_i} C(w_i)} = \sum_{w_{i-1}} P_{KN}(w_{i-1}w_i) = \sum_{w_{i-1}} P_{KN}(w_i \mid w_{i-1}) P(w_{i-1}) \quad (11.35)$$

For $P(w_{i-1})$, we simply take the distribution found in the training data

$$P(w_{i-1}) = \frac{C(w_{i-1})}{\sum_{w_{i-1}} C(w_{i-1})} \quad (11.36)$$

We substitute Eq. (11.33) in Eq. (11.35). For the bigram case, we have:

$$
\begin{aligned}
& C(w_i) \\
& = \sum_{w_{i-1}} C(w_{i-1})[\frac{\max\{C(w_{i-1}w_i)-D,0\}}{\sum_{w_i} C(w_{i-1}w_i)} + \frac{D}{\sum_{w_i} C(w_{i-1}w_i)} \mathbb{C}(w_{i-1}\bullet) P_{KN}(w_i)] \\
& = \sum_{w_{i-1}:C(w_{i-1}w_i)>0} C(w_{i-1}) \frac{C(w_{i-1}w_i)-D}{C(w_{i-1})} + \sum_{w_{i-1}} C(w_{i-1}) \frac{D}{C(w_{i-1})} \mathbb{C}(w_{i-1}\bullet) P_{KN}(w_i) \\
& = C(w_i) - \mathbb{C}(\bullet w_{i-1})D + DP_{KN}(w_i) + DP_{KN}(w_i) \sum_{w_{i-1}} \mathbb{C}(w_{i-1}\bullet)
\end{aligned}
\tag{11.37}
$$

Solving the equation, we get

$$
P_{KN}(w_i) = \frac{\mathbb{C}(\bullet w_i)}{\sum_{w_i} \mathbb{C}(\bullet w_i)} \tag{11.38}
$$

which can be generalized to higher-order models:

$$
P_{KN}(w_i \mid w_{i-n+2}...w_{i-1}) = \frac{\mathbb{C}(\bullet w_{i-n+2}...w_i)}{\sum_{w_i} \mathbb{C}(\bullet w_{i-n+2}...w_i)} \tag{11.39}
$$

where $\mathbb{C}(\bullet w_{i-n+2}...w_i)$ is the number of different words that precede $w_{i-n+2}...w_i$.

In practice, instead of using a single discount D for all nonzero counts as in Kneser-Ney smoothing, we can have a number of different parameters (D_i) that depend on the range of counts:

$$
\begin{aligned}
& P_{KN}(w_i \mid w_{i-n+1}...w_{i-1}) \\
& = \frac{C(w_{i-n+1}...w_i) - D(C(w_{i-n+1}...w_i))}{\sum_{w_i} C(w_{i-n+1}...w_i)} + \\
& + \gamma(w_{i-n+1}...w_{i-1}) P_{KN}(w_i \mid w_{i-n+2}...w_{i-1})
\end{aligned}
\tag{11.40}
$$

This modification is motivated by evidence that the ideal average discount for n-grams with one or two counts is substantially different from the ideal average discount for n-grams with higher counts.

11.4.3. Class *N*-grams

As discussed in Chapter 2, we can define classes for words that exhibit similar semantic or grammatical behavior. This is another effective way to handle the data sparsity problem.

Class-based language models have been shown to be effective for rapid adaptation, training on small data sets, and reduced memory requirements for real-time speech applications.

For any given assignment of a word w_i to class c_i, there may be many-to-many mappings, e.g., a word w_i may belong to more than one class, and a class c_i may contain more than one word. For the sake of simplicity, assume that a word w_i can be uniquely mapped to only one class c_i. The n-gram model can be computed based on the previous n-1 classes:

$$P(w_i|c_{i-n+1}\cdots c_{i-1}) = P(w_i \mid c_i)P(c_i|c_{i-n+1}\cdots c_{i-1}) \tag{11.41}$$

where $P(w_i|c_i)$ denotes the probability of word w_i given class c_i in the current position, and $P(c_i|c_{i-n+1}\cdots c_{i-1})$ denotes the probability of class c_i given the class history. With such a model, we can learn the class mapping $w \rightarrow c$ from either a training text or task knowledge we have about the application. In general, we can express the class trigram as:

$$P(\mathbf{W}) = \sum_{c_1 \cdots c_n} \prod_i P(w_i \mid c_i)P(c_i \mid c_{i-2}, c_{i-1}) \tag{11.42}$$

If the classes are nonoverlapping, i.e. a word may belong to only one class, then Eq. (11.42) can be simplified as:

$$P(\mathbf{W}) = \prod_i P(w_i \mid c_i)P(c_i \mid c_{i-2}, c_{i-1}) \tag{11.43}$$

If we have the mapping function defined, we can easily compute the class n-gram. We can estimate the empirical frequency of each word $C(w_i)$, and of each class $C(c_i)$. We can also compute the empirical frequency that a word from one class will be followed immediately by a word from another $C(c_{i-1}c_i)$. As a typical example, the bigram probability of a word given the prior word (class) can be estimated as

$$P(w_i \mid w_{i-1}) = P(w_i \mid c_{i-1}) = P(w_i \mid c_i)P(c_i \mid c_{i-1}) = \frac{C(w_i)}{C(c_i)} \frac{C(c_{i-1}c_i)}{C(c_{i-1})} \tag{11.44}$$

For general-purpose large vocabulary dictation applications, class-based n-grams have not significantly improved recognition accuracy. They are mainly used as a backoff model to complement the lower-order n-grams for better smoothing. Nevertheless, for limited domain speech recognition, the class-based n-gram is very helpful as the class can efficiently encode semantic information for improved key word spotting and speech understanding accuracy.

11.4.3.1. Rule-Based Classes

There are a number of ways to cluster words together based on the syntactic-semantic information that exists for the language and the task. For example, part-of-speech can be gen-

erally used to produce a small number of classes although this may lead to significantly increased perplexity. Alternatively, if we have domain knowledge, it is often advantageous to cluster together words that have a similar semantic functional role. For example, if we need to build a conversational system for air travel information systems, we can group the name of different airlines such as *United Airlines, KLM,* and *Air China*, into a broad *airline class.* We can do the same thing for the names of different airports such as *JFK, Narita,* and *Heathrow*, the names of different cities like *Beijing, Pittsburgh,* and *Moscow,* and so on. Such an approach is particularly powerful, since the amount of training data is always limited. With generalized broad classes of semantically interpretable meaning, it is easy to add a new airline such as *Redmond Air* into the classes if there is indeed a start-up airline named *Redmond Air* that the system has to incorporate. The system is now able to assign a reasonable probability to a sentence like "*Show me all flights of Redmond Air from Seattle to Boston*" in a similar manner as "*Show me all flights of United Airlines from Seattle to Boston.*" We only need to estimate the probability of *Redmond Air,* given the airline class c_i. We can use the existing class n-gram model that contains the broad structure of the air travel information system as it is.

Without such a broad interpretable class, it would be extremely difficult to deal with new names the system needs to handle, although these new names can always be mapped to the special class of the unknown word or proper noun classes. For these new words, we can alternatively map them into a word that has a similar syntactic and semantic role. Thus, the new word inherits all the possible word trigram relationships that may be very similar to those of the existing word observed with the training data.

11.4.3.2. Data-driven Classes

For a general-purpose dictation application, it is impractical to derive functional classes in the same manner as a domain-specific conversational system that focuses on a narrow task. Instead, data-driven clustering algorithms have been used to generalize the concept of word similarities, which is in fact a search procedure to find a class label for each word with a predefined objective function. The set of words with the same class label is called a cluster. We can use the maximum likelihood criterion as the objective function for a given training corpus and a given number of classes, which is equivalent to minimizing the perplexity for the training corpus. Once again, the EM algorithm can be used here. Each word can be initialized to a random cluster (class label). At each iteration, every word is moved to the class that produces the model with minimum perplexity [9, 48]. The perplexity modifications can be calculated independently, so that each word is evaluated as if all other word classes were held fixed. The algorithm converges when no single word can be moved to another class in a way that reduces the perplexity of the clustered n-gram model.

One special kind of class n-gram models is based on the decision tree as discussed in Chapter 4. We can use it to create equivalent classes for words in the history, so that can we

have a compact long-distance n-gram language model [2]. The sequential decomposition, as expressed in Eq. (11.12), is approximated as:

$$P(\mathbf{W}) = \prod_{i=1}^{n} P(w_i | E(w_1, w_2, \ldots, w_{i-1})) = \prod_{i=1}^{n} P(w_i | E(\mathbf{h})) \tag{11.45}$$

where $E(\mathbf{h})$ denotes a many-to-one mapping function that groups word histories $\mathbf{h}$ into some equivalence classes. It is important to have a scheme that can provide adequate information about the history so it can serve as a basis for prediction. In addition, it must yield a set of classes that can be reliably estimated. The decision tree method uses entropy as a criterion in developing the equivalence classes that can effectively incorporate long-distance information. By asking a number of questions associated with each node, the decision tree can classify the history into a small number of equivalence classes. Each leaf of the tree, thus, has the probability $P(w_i | E(w_1 \ldots w_{i-1}))$ that is derived according to the number of times the word w_i is found in the leaf. The selection of questions in building the tree can be infinite. We can consider not only the syntactic structure, but also semantic meaning to derive permissible questions from which the entropy criterion would choose. A full-fledged question set that is based on detailed analysis of the history is beyond the limit of our current computing resources. As such, we often use the membership question to check each word in the history.

11.4.4. Performance of *N*-gram Smoothing

The performance of various smoothing algorithms depends on factors such as the training-set sizes. There is a strong correlation between the test-set perplexity and word error rate. Smoothing algorithms leading to lower perplexity generally result in a lower word error rate. Among all the methods discussed here, the Kneser-Ney method slightly outperforms other algorithms over a wide range of training-set sizes and corpora, and for both bigram and trigram models. Albeit the difference is not large, the good performance of the Kneser-Ney smoothing is due to the modified backoff distributions. The Katz algorithms and deleted interpolation smoothing generally yield the next best performance. All these three smoothing algorithms perform significantly better than the n-gram model without any smoothing. The deleted interpolation algorithm performs slightly better than the Katz method in sparse data situations, and the reverse is true when data are plentiful. Katz's algorithm is particularly good at smoothing larger counts; these counts are more prevalent in larger data sets.

Class n-grams offer different kind of smoothing. While clustered n-gram models often offer no significant test-set perplexity reduction in comparison to the word n-gram model, it is beneficial to smooth the word n-gram model via either backoff or interpolation methods.

For example, the decision-tree based long-distance class language model does not offer significantly improved speech recognition accuracy until it is interpolated with the word trigram. They are effective as a domain-specific language model if the class can accommodate domain-specific information.

Smoothing is a fundamental technique for statistical modeling, important not only for language modeling but for many other applications as well. Whenever data sparsity is an issue, smoothing can help performance, and data sparsity is almost always an issue in statistical modeling. In the extreme case, where there is so much training data that all parameters can be accurately trained without smoothing, you can almost always expand the model, such as by moving to a higher-order n-gram model, to achieve improved performance. With more parameters, data sparsity becomes an issue again, but a proper smoothing model is usually more accurate than the original model. Thus, no matter how much data you have, smoothing can almost always help performance, and for a relatively small effort.

11.5. ADAPTIVE LANGUAGE MODELS

Dynamic adjustment of the language model parameter, such as n-gram probabilities, vocabulary size, and the choice of words in the vocabulary, is important, since the topic of conversation is highly nonstationary [4, 33, 37, 41, 46]. For example, in a typical dictation application, a particular set of words in the vocabulary may suddenly burst forth and then become dormant later, based on the current conversation. Because the topic of the conversation may change from time to time, the language model should be dramatically different based on the topic of the conversation. We discuss several adaptive techniques that can improve the quality of the language model based on the real usage of the application.

11.5.1. Cache Language Models

To adjust word frequencies observed in the current conversation, we can use a dynamic *cache* language model [41]. The basic idea is to accumulate word n-grams dictated so far in the current document and use these to create a local dynamic n-gram model such as bigram $P_{cache}(w_i|w_{i-1})$. Because of limited data and nonstationary nature, we should use a lower-order language model that is no higher than a trigram model $P_{cache}(w_i|w_{i-2}w_{i-1})$, which can be interpolated with the dynamic bigram and unigram. Empirically, we need to normally give a high weight to the unigram cache model, because it is better trained with the limited data in the cache.

With the cache trigram, we interpolate it with the static n-gram model $P_s(w_i|w_{i-n+1}\cdots w_{i-1})$. The interpolation weight can be made to vary with the size of the cache.

$$\begin{aligned} &P_{cache}(w_i \mid w_{i-n+1}\cdots w_{i-1}) \\ &= \lambda_c P_s(w_i \mid w_{i-n+1}\cdots w_{i-1}) + (1-\lambda_c) P_{cache}(w_i \mid w_{i-2}w_{i-1}) \end{aligned} \tag{11.46}$$

The cache model is desirable in practice because of its impressive empirical performance improvement. In a dictation application, we often encounter new words that are not in the static vocabulary. The same words also tend to be repeated in the same article. The cache model can address this problem effectively by adjusting the parameters continually as recognition and correction proceed for incrementally improved performance. A noticeable benefit is that we can better predict words belonging to fixed phrases such as *Windows NT* and *Bill Gates*.

11.5.2. Topic-Adaptive Models

The topic can change over time. Such topic or style information plays a critical role in improving the quality of the static language model. For example, the prediction of whether the word following the phrase *the operating* is *system* or *table* can be improved substantially by knowing whether the topic of discussion is related to computing or medicine.

Domain or topic-clustered language models split the language model training data according to topic. The training data may be divided using the known category information or using automatic clustering. In addition, a given segment of the data may be assigned to multiple topics. A topic-dependent language model is then built from each cluster of the training data. Topic language models are combined using linear interpolation or other methods such as maximum entropy techniques discussed in Section 11.5.3.

We can avoid any pre-defined clustering or segmentation of the training data. The reason is that the best clustering may become apparent only when the current topic of discussion is revealed. For example, when the topic is hand-injury to baseball player, the pre-segmented clusters of topic *baseball & hand-injuries* may have to be combined. This leads to a union of the two clusters, whereas the ideal dataset is obtained by the intersection of these clusters. In general, various combinations of topics lead to a combinatorial explosion in the number of compound topics, and it appears to be a difficult task to anticipate all the needed combinations beforehand.

We base our determination of the most suitable language model data to build a model upon the particular history of a given document. For example, we can use it as a query against the entire training database of documents using *information retrieval* techniques [57]. The documents in the database can be ranked by relevance to the query. The most relevant documents are then selected as the adaptation set for the topic-dependent language model. The process can be repeated as the document is updated.

There are two major steps we need to consider here. The first involves using the available document history to retrieve similar documents from the database. The second consists of using the similar document set retrieved in the first step to adapt the general or topic-independent language model. Available document history depends upon the design and the requirements of the recognition system. If the recognition system is designed for live-mode application, where the recognition results must be presented to the user with a small delay, the available document history will be the history of the document user created so far. On the other hand, in a recognition system designed for batch operation, the amount of time

taken by the system to recognize speech is of little consequence to the user. In the batch mode, therefore, a multi-pass recognition system can be used, and the document history will be the recognizer transcript produced in the current pass.

The well-known information retrieval measure called *TFIDF* can be used to locate similar documents in the training database [57]. The term frequency (TF) tf_{ij} is defined as the frequency of the jth term in the document D_i, the unigram count of the term j in the document D_i. The inverse document frequency (IDF) idf_j is defined as the frequency of the jth term over the entire database of documents, which can be computed as:

$$idf_j = \frac{\text{Total number of documents}}{\text{Number of documents containing term } j} \tag{11.47}$$

The combined TF-IDF measure is defined as:

$$TFIDF_{ij} = tf_{ij} \log(idf_i) \tag{11.48}$$

The combination of TF and IDF can help to retrieve similar documents. It highlights words of particular interest to the query (via TF), while de-emphasizing common words that appear across different documents (via IDF). Each document including the query itself, can be represented by the TFIDF vector. Each element of the vector is the TFIDF value that corresponds to a word (or a term) in the vocabulary. Similarity between the two documents is then defined to be the cosine of the angle between the corresponding vectors. Therefore, we have:

$$Similarity(D_i, D_j) = \frac{\sum_k tfidf_{ik} * tfidf_{jk}}{\sqrt{\sum_k (tfidf_{ik})^2 * \sum_k (tfidf_{jk})^2}} \tag{11.49}$$

All the documents in the training database are ranked by the decreasing similarity between the document and the history of the current document dictated so far, or by a topic of particular interest to the user. The most similar documents are selected as the adaptation set for the topic-adaptive language model [46].

11.5.3. Maximum Entropy Models

The language model we have discussed so far combines different n-gram models via linear interpolation. A different way to combine sources is the maximum entropy approach. It constructs a single model that attempts to capture all the information provided by the various knowledge sources. Each such knowledge source is reformulated as a set of constraints that the desired distribution should satisfy. These constraints can be, for example, marginal distributions of the combined model. Their intersection, if not empty, should contain a set of

probability functions that are consistent with these separate knowledge sources. Once the desired knowledge sources have been incorporated, we make no other assumption about other constraints, which leads to choosing the flattest of the remaining possibilities, the one with the highest entropy. The maximum entropy principle can be stated as follows:

- Reformulate different information sources as constraints to be satisfied by the target estimate.
- Among all probability distributions that satisfy these constraints, choose the one that has the highest entropy.

Given a general event space $\{\mathbf{X}\}$, let $P(\mathbf{X})$ denote the combined probability function. Each constraint is associated with a characteristic function of a subset of the sample space, $f_i(\mathbf{X})$. The constraint can be written as:

$$\sum_{\mathbf{X}} P(\mathbf{X}) f_i(\mathbf{X}) = E_i \tag{11.50}$$

where E_i is the corresponding desired expectation for $f_i(\mathbf{X})$, typically representing the required marginal probability of $P(\mathbf{X})$. For example, to derive a word trigram model, we can reformulate Eq. (11.50) so that constraints are introduced for unigram, bigram, and trigram probabilities. These constraints are usually set only where marginal probabilities can be estimated from a corpus. For example, the unigram constraint can be expressed as

$$f_{w_1}(w) = \begin{cases} 1 & \text{if } w=w_1 \\ 0 & \text{otherwise} \end{cases} \tag{11.51}$$

The desired value E_{w_1} can be the empirical expectation in the training data, $\sum_{w \in \text{training data}} f_{w_1}(w)/N$, and the associated constraint is

$$\sum_{\mathbf{h}} P(\mathbf{h}) \sum_{w} P(w \mid \mathbf{h}) f_{w_1}(w) = E_{w_1} \tag{11.52}$$

where $\mathbf{h}$ is the word history preceding word w.

We can choose $P(\mathbf{X})$ to diverge minimally from some other known probability function $Q(\mathbf{X})$, that is, to minimize the divergence function:

$$\sum_{\mathbf{X}} P(\mathbf{X}) \log \frac{P(\mathbf{X})}{Q(\mathbf{X})} \tag{11.53}$$

When $Q(\mathbf{X})$ is chosen as the uniform distribution, the divergence is equal to the negative of entropy with a constant. Thus minimizing the divergence function leads to maximiz-

ing the entropy. Under a minor consistent assumption, a unique solution is guaranteed to exist in the form [20]:

$$P(\mathbf{X}) \propto \prod_i \mu_i^{f_i(\mathbf{X})} \tag{11.54}$$

where μ_i is an unknown constant to be found. To search the exponential family defined by Eq. (11.54) for the μ_i that make $P(\mathbf{X})$ satisfy all the constraints, an iterative algorithm called generalized iterative scaling exists [20]. It guarantees to converge to the solution with some arbitrary initial μ_i. Each iteration creates a new estimate $P(\mathbf{X})$, which is improved in the sense that it matches the constraints better than its previous iteration [20]. One of the most effective applications of the maximum entropy model is to integrate the cache constraint into the language model directly, instead of interpolating the cache n-gram with the static n-gram. The new constraint is that the marginal distribution of the adapted model is the same as the lower-order n-gram in the cache [56]. In practice, the maximum entropy method has not offered any significant improvement in comparison to the linear interpolation.

11.6. PRACTICAL ISSUES

In a speech recognition system, every string of words $\mathbf{W} = w_1 w_2 \ldots w_n$ taken from the prescribed vocabulary can be assigned a probability, which is interpreted as the a priori probability to guide the recognition process and is a contributing factor in the determination of the final transcription from a set of partial hypothesis. Without language modeling, the entire vocabulary must be considered at every decision point. It is impossible to eliminate many candidates from consideration, or alternatively to assign higher probabilities to some candidates than others to considerably reduce recognition costs and errors.

11.6.1. Vocabulary Selection

For most speech recognition systems, an inflected form is considered as a different word. This is because these inflected forms typically have different pronunciations, syntactic roles, and usage patterns. So the words *work*, *works*, *worked*, and *working* are counted as four different words in the vocabulary.

We prefer to have a smaller vocabulary size, since this eliminates potential confusable candidates in speech recognition, leading to improved recognition accuracy. However, the limited vocabulary size imposes a severe constraint on the users and makes the system less flexible. In practice, the percentage of the Out-Of-Vocabulary (OOV) word rate directly affects the perceived quality of the system. Thus, we need to balance two kinds of errors, the OOV rate and the word recognition error rate. We can have a larger vocabulary to minimize the OOV rate if the system resources permit. We can minimize the expected OOV rate of the

test data with a given vocabulary size. A corpus of text is used in conjunction with dictionaries to determine appropriate vocabularies.

The availability of various types and amounts of training data, from various time periods, affects the quality of the derived vocabulary. Given a collection of training data, we can create an ordered word list with the lowest possible OOV curve, such that, for any desired vocabulary size V, a minimum-OOV-rate vocabulary can be derived by taking the most frequent V words in that list. Viewed this way, the problem becomes one of estimating unigram probabilities of the test distribution, and then ordering the words by these estimates.

As illustrated in Figure 11.6, the perplexity generally increases with the vocabulary size, albeit it really does not make much sense to compare the perplexity of different vocabulary sizes. There are generally more competing words for a given context when the vocabulary size becomes big, which leads to increased recognition error rate. In practice, this is offset by the OOV rate, which decreases with the vocabulary size as illustrated in Figure 11.7. If we keep the vocabulary size fixed, we need more than 200,000 words in the vocabulary to have 99.5% English words coverage. For more inflectional languages such as German, larger vocabulary sizes are required to achieve coverage similar to that of English.[6]

In practice, it is far more important to use data from a specific topic or domain, if we know in what domain the speech recognizer is used. In general, it is also important to consider coverage of a specific time period. We should use training data from that period, or as close to it as possible. For example, if we know we will talk only about air travel, we benefit from using the air-travel related vocabulary and language model. This point is well illustrated by the fact that the perplexity of the domain-dependent bigram can be reduced by more than a factor of five over the general-purpose English trigram.

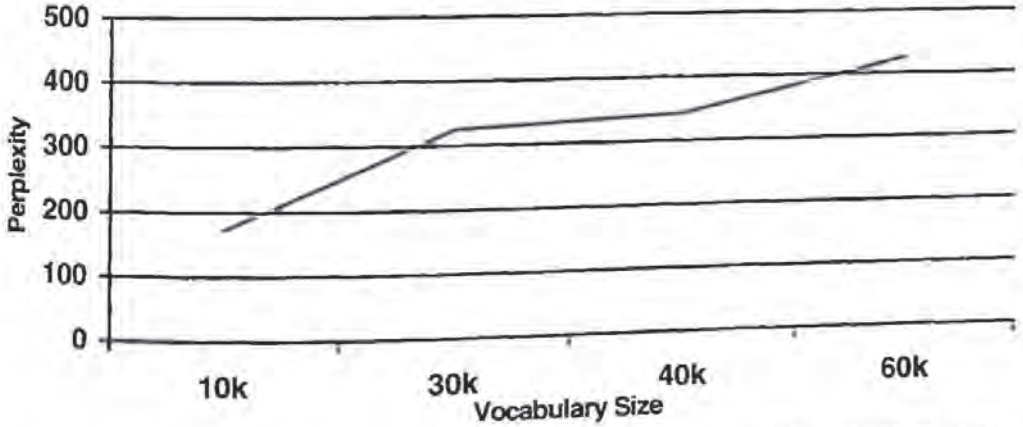


Figure 11.6 The perplexity of bigram with different vocabulary sizes. The training set consists of 500 million words derived from various sources, including newspapers and email. The test set comes from the whole Microsoft Encarta, an encyclopedia that has a wide coverage of different topics.

[6] The OOV rate of German is about twice as high as that of English with a 20k-word vocabulary [34].

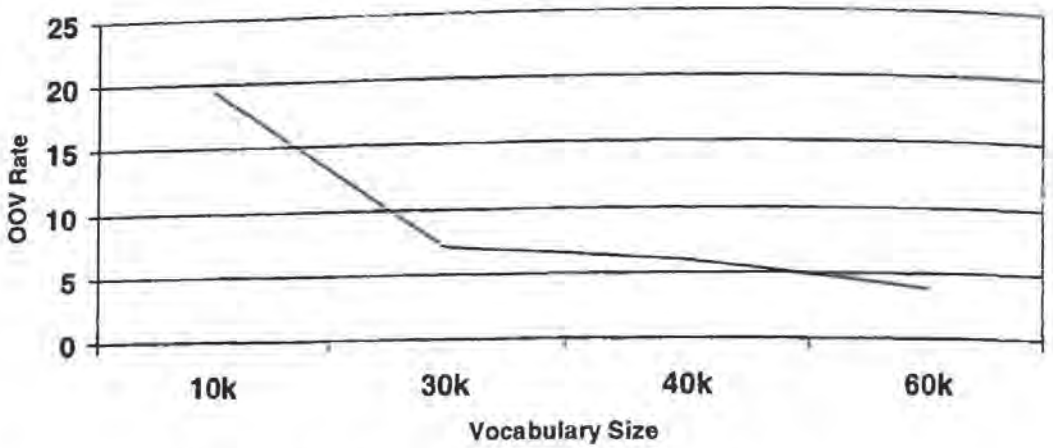


Figure 11.7 The OOV rate with different vocabulary size. The training set consists of 500 million words derived from various sources including newspaper and email. The test set came from the whole Microsoft Encarta encyclopedia.

For a user of a speech recognition system, a more personalized vocabulary can be much more effective than a general fixed vocabulary. The coverage can be dramatically improved as customized new words are added to a starting static vocabulary of 20,000. Typically, the coverage of such a system can be improved from 93% to more than 98% after 1000-4000 customized words are added to the vocabulary [18].

In North American general business English, the least frequent words among the most frequent 60,000 have a frequency of about 1:7,000,000. In optimizing a 60,000-word vocabulary we need to distinguish words with frequency of 1:7,000,000 from those that are slightly less frequent. To differentiate somewhat reliably between a 1:7,000,000 word and, say, a 1:8,000,000 word, we need to observe them enough times for the difference in their counts to be statistically reliable. For constructing a decent vocabulary, it is important that most such words are ranked correctly. We may need 100,000,000 words to estimate these parameters. This agrees with the empirical results, in which as more training data is used, the OOV curve improves rapidly up to 50,000,000 words and then more slowly beyond that point.

11.6.2. *N*-gram Pruning

When high order n-gram models are used, the model sizes typically become too large for practical applications. It is necessary to prune parameters from n-gram models such that the relative entropy between the original and the pruned model is minimized. You can choose n-grams so as to maximize performance (i.e., minimize perplexity) while minimizing the model size [39, 59, 64].

The criterion to prune n-grams can be based on some well-understood information-theoretic measure of language model quality. For example, the pruning method by Stolcke [64] removes some n-gram estimates while minimizing the performance loss. After pruning,

the retained explicit n-gram probabilities are unchanged, but backoff weights are recomputed. Stolcke pruning uses the criterion that minimizes the distance between the distribution embodied by the original model and that of the pruned model based on the *Kullback-Leibler distance* defined in Eq. (3.181). Since it is infeasible to maximize over all possible subsets of n-grams, Stolcke prunning assumes that the n-grams affect the relative entropy roughly independently, and compute the distance due to each individual n-gram. The n-grams are thus ranked by their effect on the model entropy, and those that increase relative entropy the least are pruned accordingly. The main approximation is that we do not consider possible interactions between selected n-grams, and prune based solely on relative entropy due to removing a single n-gram. This avoids searching the exponential space of n-gram subsets.

To compute the relative entropy, $KL(p \parallel p')$, between the original and pruned n-gram models p and p', there is no need to sum over the vocabulary. By plugging in the terms for the backoff estimates, the sum can be factored as shown in Eq. (11.55) for a more efficient computation.

$$\begin{aligned} KL(p \parallel p') = & -P(h)\{P(w \mid h)[\log P(w \mid h') + \log \alpha'(h) - \log P(w \mid h)] \\ & + [\log \alpha'(h) - \log \alpha(h)](1 - \sum_{w_i \in \neg Backoff(w,h)} P(w_i \mid h))\} \end{aligned} \tag{11.55}$$

where the sum in $\sum_{w_i \in \neg Backoff(w,h)} P(w_i \mid h)$ is over all non-backoff estimates. To compute the revised backoff weights $\alpha'(h)$, you can simply drop the term for the pruned n-gram from the summation (backoff weight computation is illustrated in Algorithm 11.1).

In practice, pruning is highly effective. Stolcke reported that the trigram model can be compressed by more than 25% without degrading recognition performance. Comparing the pruned 4-gram model to the unpruned trigram model, it is better to use pruned 4-grams than to use a much larger number of trigrams.

11.6.3. CFG vs. N-gram Models

This chapter has discussed two major language models. While CFGs remain one of the most important formalisms for interpreting natural language, word n-gram models are surprisingly powerful for domain-independent applications. These two formalisms can be unified for both speech recognition and spoken language understanding. To improve portability of the domain-independent n-gram, it is possible to incorporate domain-specific CFGs into the domain-independent n-gram that can improve generalizability of the CFG and specificity of the n-gram.

The CFG is not only powerful enough to describe most of the structure in spoken language, but also restrictive enough to have efficient parsers. $P(\mathbf{W})$ is regarded as 1 or 0 depending upon whether the word sequence is accepted or rejected by the grammar. The

problem is that the grammar is *almost always incomplete*. A CFG-based system is good only when you know what sentences to speak, which diminishes the system's value and usability of the system. The advantage of CFG's structured analysis is, thus, nullified by the poor coverage in most real applications. On the other hand, the *n*-gram model is trained with a large amount of data, the *n*-word dependency can often accommodate both syntactic and semantic structure seamlessly. The prerequisite of this approach is that we have enough training data. The problem for *n*-gram models is that we need a lot of data and the model may not be specific enough.

It is possible to take advantage of both rule-based CFGs and data-driven *n*-grams. Let's consider the following training sentences:

```
Meeting at three with Zhou Li.
Meeting at four PM with Derek.
```

If we use a word trigram, we estimate *P(Zhou|three with)* and *P(Derek|PM with)*, etc. There is no way we can capture needed long-span semantic information in the training data. A unified model has a set of CFGs that can capture the semantic structure of the domain. For the example listed here, we have a CFG for {name} and {time}, respectively. We can use the CFG to parse the training data to spot all the potential semantic structures in the training data. The training sentences now look like:

```
Meeting {at three:TIME} with {Zhou Li:NAME}
Meeting {at four PM:TIME} with {Derek: NAME}
```

With analyzed training data, we can estimate our *n*-gram probabilities as usual. We have probabilities, such as P({name}|{time} with), instead of *P(Zhou|three with)*, which is more meaningful and accurate. Inside each CFG we also derive P("*Zhou Li*"|{name}) and P("*four PM*"|{time}) from the existing *n*-gram (*n*-gram probability inheritance) so that they are normalized. If we add a new name to the existing {name} CFG, we use the existing *n*-gram probabilities to renormalize our CFGs for the new name. The new approach can be regarded as a standard *n*-gram in which the vocabulary consists of words and structured classes, as discussed in Section 11.4.3. The structured class can be very simple, such as {date}, {time}, and {name}, or can be very complicated, such as a CFG that contains deep structured information. The probability of a word or class depends on the previous words or CFG classes.

It is possible to inherit probability from a word *n*-gram LM. Let's take word trigram as our example here. An input utterance $\mathbf{W} = w_1 w_2 ... w_n$ can be segmented into a sequence $\mathbf{T} = t_1 t_2 ... t_m$, where each t_i is either a word in $\mathbf{W}$ or a CFG non-terminal that covers a sequence of words $\overline{u}_{t_i}$ in $\mathbf{W}$. The likelihood of $\mathbf{W}$ under the segmentation $\mathbf{T}$ is, therefore,

$$P(\mathbf{W},\mathbf{T}) = \prod_i P(t_i \mid t_{i-1}, t_{i-2}) \prod_i P(\overline{u}_{t_i} \mid t_i) \tag{11.56}$$

$P(\bar{u}_{t_i} | t_i)$, the likelihood of generating a word sequence $\bar{u}_{t_i} = [u_{t_i,1} u_{t_i,2} ... u_{t_i,k}]$ from the CFG non-terminal t_i, can be inherited from the domain-independent word trigram. We can essentially use the CFG constraint to condition the domain-independent trigram into a domain-specific trigram. Such a unified language model can dramatically improve cross-domain performance using domain-specific CFGs [66].

In summary, the CFG is widely used to specify the permissible word sequences in natural language processing when training corpora are unavailable. It is suitable for dealing with structured command and control applications in which the vocabulary is small and the semantics of the task is well defined. The CFG either accepts the input sentence or rejects it. There is a serious coverage problem associated with CFGs. In other words, the accuracy for the CFG can be extremely high when the test data are covered by the grammar. Unfortunately, unless the task is narrow and well-defined, most users speak sentences that may not be accepted by the CFG, leading to word recognition errors.

Statistical language models such as trigrams assign an estimated probability to any word that can follow a given word history without parsing the structure of the history. Such an approach contains some limited syntactic and semantic information, but these probabilities are typically trained from a large corpus. Speech recognition errors are much more likely to occur within trigrams and (especially) bigrams that have not been observed in the training data. In these cases, the language model typically relies on lower-order statistics. Thus, increased n-gram coverage translates directly into improved recognition accuracy, but usually at the cost of increased memory requirements.

It is interesting to compute the true entropy of the language so that we understand what a solid lower bound is for the language model. For English, Shannon [60] used human subjects to guess letters by looking at how many guesses it takes people to derive the correct one based on the history. We can thus estimate the probability of the letters and hence the entropy of the sequence. Shannon computed the per-letter entropy of English with an entropy of 1.3 bits for 26 letters plus space. This may be an underestimate, since it is based on a single text. Since the average length of English written words (including space) is about 5.5 letters, the Shannon estimate of 1.3 bits per letter corresponds to a per-word perplexity of 142 for general English.

Table 11.2 summarizes the performance of several different n-gram models on a 60,000-word continuous speech dictation application. The experiments used about 260 million words from a newspaper such as the *Wall Street Journal*. The speech recognizer is based on Whisper described in Chapter 9. As you can see from the table, when the amount of training data is sufficient, both Katz and Kneser-Ney smoothing offer comparable recognition performance, although Kneser-Ney smoothing offers a modest improvement when the amount of training data is limited.

In comparison to Shannon's estimate of general English word perplexity, the trigram language for the *Wall Street Journal* is lower (91.4 vs. 142). This is because the text is mostly business oriented with a fairly homogeneous style and word usage pattern. For example, if we use the trigram language for data from a new domain that is related to personal information management, the test-set word perplexity can increase to 378 [66].

Table 11.2 *N*-gram perplexity and its corresponding speaker-independent speech recognition word error rate.

Models	Perplexity	Word Error Rate
Unigram Katz	1196.45	14.85%
Unigram Kneser-Ney	1199.59	14.86%
Bigram Katz	176.31	11.38%
Bigram Kneser-Ney	176.11	11.34%
Trigram Katz	95.19	9.69%
Trigram Kneser-Ney	91.47	9.60%

11.7. HISTORICAL PERSPECTIVE AND FURTHER READING

There is a large and active area of research in both speech and linguistics. These two distinctive communities worked on the problem with very different paths, leading to the stochastic language models and the formal language theory. The linguistics community has developed tools for tasks like parsing sentences, assigning semantic relations to the parts of a sentence, and so on. Most of these parser algorithms have the same characteristics, that is, they tabulate each sub-derivation and reuse it in building any derivation that shares that sub-derivation with appropriate grammars [22, 65, 67]. They have polynomial complexity with respect to sentence length because of *dynamic programming* principles to search for optimal derivations with respect to appropriate evaluation functions on derivations. There are three well-known dynamic programming parsers with a worst-case behavior of $O(n^3)$, where n is the number of words in the sentence: the Cocke-Younger-Kasami (CYK) algorithm (a bottom-up parser, proposed by J. Cocke, D. Younger, and T. Kasami) [32, 67], the Graham-Harrison-Ruzzo algorithm (bottom-up) [30], and the Earley algorithm (top-down) [21].

On the other hand, the speech community has developed tools to predict the next word on the basis of what has been said, in order to improve speech recognition accuracy [35]. Neither approach has been completely successful. The formal grammar and the related parsing algorithms are too brittle for comfort and require a lot of human retooling to port from one domain to another. The lack of structure and deep understanding has taken its toll on statistical technology's ability to choose the right words to guide speech recognition.

In addition to those discussed in this chapter, many alternative formal techniques are available. Augmented context-free grammars are used for natural language to capture grammatical natural languages such as agreement and subcategorization. Examples include generalized phrase structure grammars and head-driven phrase structure grammars [26, 53]. You can further generalize the augmented context-free grammar to the extent that the requirement of *context free* becomes unnecessary. The entire grammar, known as the *unification grammar*, can be specified as a set of constraints between feature structures [62]. Most of these grammars have only limited success when applied to spoken language systems. In fact, no practical domain-independent parser of unrestricted text has been developed for spoken language systems, partly because disambiguation requires the specification of detailed semantic information. Analysis of the Susanne Corpus with a crude parser suggests

that over 80% of sentences are structurally ambiguous. More recently, large *treebanks* of parsed texts have given impetus to statistical approaches to parsing. Probabilities can be estimated from treebanks or plain text [6, 8, 24, 61] to efficiently rank analyses produced by modified chart parsing algorithms. These systems have yielded results of around 75% accuracy in assigning analyses to (unseen) test sentences from the same source as the unambiguous training material. Attempts have also been made to use statistical induction to *learn* the correct grammar for a given corpus of data [7, 43, 51, 58]. Nevertheless, these techniques are limited to simple grammars with category sets of a dozen or so non-terminals, or to training on manually parsed data. Furthermore, even when parameters of the grammar and control mechanism can be learned automatically from training corpora, the required corpora do not exist or are too small for proper training. In practice, we can devise grammars that specify directly how relationships relevant to the task may be expressed. For instance, one may use a phrase-structure grammar in which nonterminals stand for task concepts and relationships and rules specify possible expressions of those concepts and relationships. Such *semantic grammars* have been widely used for spoken language applications as discussed in Chapter 17.

It is worthwhile to point out that many natural language parsing algorithms are *NP-complete*, a term for a class of problems that are suspected to be particularly difficult to process. For example, maintaining lexical and agreement features over a potentially infinite-length sentence causes the unification-based formalisms to be NP-complete [3].

Since the predictive power of a general-purpose grammar is insufficient for reasonable performance, n-gram language models continue to be widely used. A complete proof of Good-Turing smoothing was presented by Church *et al.* [17]. Chen and Goodman [13] provide a detailed study on different n-gram smoothing algorithms. Jelinek's Eurospeech tutorial paper [35] provides an interesting historical perspective on the community's efforts to improve trigrams. Mosia and Giachin's paper [48] has detailed experimental results on class-based language models. Class-based model may be based on parts of speech or morphology [10, 16, 23, 47, 63]. More detailed discussion of the maximum entropy language model can be found in [5, 36, 42, 44, 52, 55, 56].

One interesting research area is to combine both n-grams and the structure that is present in language. A concerted research effort to explore structure-based language model may be the key for significant progress to occur in language modeling. This can be done as annotated data becomes available. Nasr *et al.* [50] have considered a new unified language model composed of several local models and a general model linking the local models together. The local model used in their system is based on the stochastic FSA, which is estimated from the training corpora. Other efforts to incorporate structured information are described in [12, 25, 27, 49, 66].

You can find tools to build n-gram language models at the CMU open source Web site[7] and SRI's language modeling toolkit Web site.[8] Both contain language modeling toolkits and documentation.

[7] http://www.speech.cs.cmu.edu/sphinx/
[8] http://www.speech.sri.com/projects/srilm/download.html

REFERENCES

[1] Aho, A.V. and J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, 1972, Englewood Cliffs, NJ, Prentice-Hall.

[2] Bahl, L.R., *et al.*, "A Tree-Based Statistical Language Model for Natural Language Speech Recognition," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 1989, **37**(7), pp. 1001-1008.

[3] Barton, G., R. Berwick, and E. Ristad, *Computational Complexity and Natural Language*, 1987, Cambridge, MA, MIT Press.

[4] Bellegarda, J., "A Latent Semantic Analysis Framework for Large-Span Language Modeling," *Eurospeech*, 1997, Rhodes, Greece, pp. 1451-1454.

[5] Berger, A., S. DellaPietra, and V. DellaPietra, "A Maximum Entropy Approach to Natural Language Processing," *Computational Linguistics*, 1996, **22**(1), pp. 39-71.

[6] Black, E., *et al.*, "Towards History-based Grammars: Using Richer Models for Probabilistic Parsing," *Proc. of the Annual Meeting of the Association for Computational Linguistics*, 1993, Columbus, Ohio, USA, pp. 31-37.

[7] Briscoe, E.J., ed. *Prospects for Practical Parsing: Robust Statistical Techniques*, in Corpus-based Research into Language: A Feschrift for Jan Aarts, ed. P.d. Haan and N. Oostdijk, 1994, Amsterdam. 67-95, Rodopi.

[8] Briscoe, E.J. and J. Carroll, "Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-based Grammars," *Computational Linguistics*, 1993, **19**, pp. 25-59.

[9] Brown, P.F., *et al.*, "Class-Based *N*-gram Models of Natural Language," *Computational Linguistics*, 1992(4), pp. 467-479.

[10] Cerf-Danon, H. and M. El-Bèze, "Three Different Probabilistic Language Models: Comparison and Combination," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1991, Toronto, Canada, pp. 297-300.

[11] Charniak, E., "Statistical Parsing with a Context-Free Grammar and Word Statistics," *AAAI-97*, 1997, Menlo Park, pp. 598-603.

[12] Chelba, C., A. Corazza, and F. Jelinek, "A Context Free Headword Language Model" in *Proc. of IEEE Automatic Speech Recognition Workshop"* 1995, Snowbird, Utah, pp. 89-90.

[13] Chen, S. and J. Goodman, "An Empirical Study of Smoothing Techniques for Language Modeling," *Proc. of Annual Meeting of the ACL*, 1996, Santa Cruz, CA.

[14] Chomsky, N., *Syntactic Structures*, 1957, The Hague: Mouton.

[15] Chomsky, N., *Aspects of the Theory of Syntax*, 1965, Cambridge, MIT Press.

[16] Church, K., "A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text," *Proc. of 2nd Conf. on Applied Natural Language Processing*, 1988, Austin, Texas, pp. 136-143.

[17] Church, K.W. and W.A. Gale, "A Comparison of the Enhanced Good-Turing and Deleted Estimation Methods for Estimating Probabilities of English Bigrams," *Computer Speech and Language*, 1991, pp. 19-54.

[18] Cole, R., *et al.*, *Survey of the State of the Art in Human Language Technology*, eds. http://cslu.cse.ogi.edu/HLTsurvey/HLTsurvey.html, 1996, Cambridge University Press.

[19] Collins, M., "A New Statistical Parser Based on Bigram Lexical Dependencies," *ACL-96*, 1996, pp. 184-191.

[20] Darroch, J.N. and D. Ratcliff, "Generalized Iterative Scaling for Log-Linear Models," *The Annals of Mathematical Statistics*, 1972, **43**(5), pp. 1470-1480.

[21] Earley, J., *An Efficient Context-Free Parsing Algorithm*, PhD Thesis, 1968, Carnegie Mellon University, Pittsburgh.

[22] Earley, J., "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, 1970, 6(8), pp. 451-455.

[23] El-Bèze, M. and A.-M. Derouault, "A Morphological Model for Large Vocabulary Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1990, Albuquerque, NM, pp. 577-580.

[24] Fujisaki, T., *et al.*, "A probabilistic parsing method for sentence disambiguation," *Proc. of the Int. Workshop on Parsing Technologies*, 1989, Pittsburgh.

[25] Galescu, L., E.K. Ringger, and A.F. Allen, "Rapid Language Model Development for New Task Domains," *Proc. of the ELRA First Int. Conf. on Language Resources and Evaluation (LREC)*, 1998, Granada, Spain.

[26] Gazdar, G., *et al.*, *Generalized Phrase Structure Grammars*, 1985, Cambridge, MA, Harvard University Press.

[27] Gillett, J. and W. Ward, "A Language Model Combining Trigrams and Stochastic Context-Free Grammars," *Int. Conf. on Spoken Language Processing*, 1998, Sydney, Australia.

[28] Good, I.J., "The Population Frequencies of Species and the Estimation of Population Parameters," *Biometrika*, 1953, pp. 237-264.

[29] Goodman, J., *Parsing Inside-Out*, PhD Thesis in Computer Science, 1998, Harvard University, Cambridge.

[30] Graham, S.L., M.A. Harrison, and W. L.Ruzzo, "An Improved Context-Free Recognizer," *ACM Trans. on Programming Languages and Systems*, 1980, 2(3), pp. 415-462.

[31] Hindle, D. and M. Rooth, "Structural Ambiguity and Lexical Relations," *DARPA Speech and Natural Language Workshop*, 1990, Hidden Valley, PA, Morgan Kaufmann.

[32] Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 1979, Reading, MA, Addision Wesley.

[33] Iyer, R., M. Ostendorf, and J.R. Rohlicek, "Language Modeling with Sentence-Level Mixtures," *Proc. of the ARPA Human Language Technology Workshop*, 1994, Plainsboro, NJ, pp. 82-86.

[34] Jardino, M., "Multilingual Stochastic *N*-gram Class Language Models," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1996, Atlanta, GA, pp. 161-163.

[35] Jelinek, F., "Up From Trigrams! The Struggle for Improved Language Models" in *Proc. of the European Conf. on Speech Communication and Technology*, 1991, Genoa, Italy, pp. 1037-1040.

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

[37] Jelinek, F., *et al.*, "A dynamic language model for speech recognition" in *Proc. of the DARPA Speech and Natural Language Workshop*, 1991, Asilomar, CA.

[38] Katz, S.M., "Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer," *IEEE Trans. Acoustics, Speech and Signal Processing*, 1987(3), pp. 400-401.

[39] Kneser, R., "Statistical Language Modeling using a Variable Context" in *Proc. of the Int. Conf. on Spoken Language Processing*, 1996, Philadelphia, PA, p. 494.

[40] Kneser, R. and H. Ney, "Improved Backing-off for N-gram Language Modeling" in *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing* 1995, Detroit, MI, pp. 181-184.

[41] Kuhn, R. and R.D. Mori, "A Cache-Based Natural Language Model for Speech Recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1990(6), pp. 570-582.

[42] Lafferty, J.D. and B. Suhm, "Cluster Expansions and Iterative Scaling for Maximum Entropy Language Models" in *Maximum Entropy and Bayesian Methods*, K. Hanson and R. Silver, eds., 1995, Kluwer Academic Publishers.

[43] Lari, K. and S.J. Young, "Applications of Stochastic Context-free Grammars Using the Inside-Outside Algorithm," *Computer Speech and Language*, 1991, 5(3), pp. 237-257.

[44] Lau, R., R. Rosenfeld, and S. Roukos, "Trigger-Based Language Models: A Maximum Entropy Approach," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN, pp. 108-113.

[45] Magerman, D.M. and M.P. Marcus, "Pearl: A Probabilistic Chart Parser," *Proc. of the Fourth DARPA Speech and Natural Language Workshop*, 1991, Pacific Grove, California.

[46] Mahajan, M., D. Beeferman, and X.D. Huang, "Improved Topic-Dependent Language Modeling Using Information Retrieval Techniques," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1999, Phoenix, AZ, pp. 541-544.

[47] Maltese, G. and F. Mancini, "An Automatic Technique to Include Grammatical and Morphological Information in a Trigram-based Statistical Language Model," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1992, San Francisco, CA, pp. 157-160.

[48] Moisa, L. and E. Giachin, "Automatic Clustering of Words for Probabilistic Language Models" in *Proc. of the European Conf. on Speech Communication and Technology* 1995, Madrid, Spain, pp. 1249-1252.

[49] Moore, R., *et al.*, "Combining Linguistic and Statistical Knowledge Sources in Natural-Language Processing for ATIS," *Proc. of the ARPA Spoken Language Sys-*

tems Technology Workshop, 1995, Austin, Texas, Morgan Kaufmann, Los Altos, CA.

[50] Nasr, A., *et al.*, "A Language Model Combining *N*-grams and Stochastic Finitie State Automata," *Proc. of the Eurospeech*, 1999, Budapest, Hungary, pp. 2175-2178.

[51] Pereira, F.C.N. and Y. Schabes, "Inside-Outside Reestimation from Partially Bracketed Corpora," *Proc. of the 30th Annual Meeting of the Association for Computational Linguistics*, 1992, pp. 128-135.

[52] Pietra, S.A.D., *et al.*, "Adaptive Language Model Estimation using Minimum Discrimination Estimation," *Proc. of the IEEE Int. Conf. on Acoustics. Speech and Signal Processing*, 1992, San Francisco, CA, pp. 633-636.

[53] Pollard, C. and I.A. Sag, *Head-Driven Phrase Structure Grammar*, 1994, Chicago, University of Chicago Press.

[54] Pullum, G. and G. Gazdar, "Natural Languages and Context-Free Languages," *Linguistics and Philosophy*, 1982, **4**, pp. 471-504.

[55] Ratnaparkhi, A., S. Roukos, and R.T. Ward, "A Maximum Entropy Model for Parsing," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan, pp. 803-806.

[56] Rosenfeld, R., *Adaptive Statistical Language Modeling: A Maximum Entropy Approach*, Ph.D. Thesis in School of Computer Science, 1994, Carnegie Mellon University, Pittsburgh, PA.

[57] Salton, G. and M.J. McGill, *Introduction to Modern Information Retrieval*, 1983, New York, McGraw-Hill.

[58] Schabes, Y., M. Roth, and R. Osborne, "Parsing the *Wall Street Journal* with the Inside-Outside Algorithm," *Proc. of the Sixth Conf. of the European Chapter of the Association for Computational Linguistics*, 1993, pp. 341-347.

[59] Seymore, K. and R. Rosenfeld, "Scalable Backoff Language Models," *Proc. of the Int. Conf. on Spoken Language Processing*, 1996, Philadelphia, PA, pp. 232.

[60] Shannon, C.E., "Prediction and Entropy of Printed English," *Bell System Technical Journal*, 1951, pp. 50-62.

[61] Sharman, R., F. Jelinek, and R.L. Mercer, "Generating a Grammar for Statistical Training," *Proc. of the Third DARPA Speech and Natural Language Workshop*, 1990, Hidden Valley, Pennsylvania, pp. 267-274.

[62] Shieber, S.M., *An Introduction to Unification-Based Approaches to Grammars*, 1986, Cambridge, UK, CSLI Publication, Leland Stanford Junior University.

[63] Steinbiss, V., *et al.*, "A 10,000-word Continuous Speech Recognition System," *Proc. of the IEEE Int. Conf. on Acoustics. Speech and Signal Processing*, 1990, Albuquerque, NM, pp. 57-60.

[64] Stolcke, A., "Entropy-based Pruning of Backoff Language Models," *DARPA Broadcast News Transcription and Understanding Workshop*, 1998, Lansdowne, VA.

[65] Tomita, M., "An Efficient Augmented-Context-Free Parsing Algorithm," *Computational Linguistics*, 1987, **13**(1-2), pp. 31-46.

[66] Wang, Y., M. Mahajan, and X. Huang, "A Unified Context-Free Grammar and *N*-Gram Model for Spoken Language Processing," *Int. Conf. on Acoustics, Speech and Signal Processing*, 2000, Istanbul, Turkey, pp. 1639-1642.

[67] Younger, D.H., "Recognition and Parsing of Context-Free Languages in Time n^3," *Information and Control*, 1967, **10**, pp. 189-208.

CHAPTER 12

Basic Search Algorithms

Continuous speech recognition (CSR) is both a pattern recognition and search problem. As described in previous chapters, the acoustic and language models are built upon a statistical pattern recognition framework. In speech recognition, making a search decision is also referred to as decoding. In fact, decoding got its name from information theory (see Chapter 3) where the idea is to *decode* a signal that has presumably been encoded by the source process and has been transmitted through the communication channel, as depicted in Chapter 1, Figure 1.1. In this chapter, we first review the general decoder architecture that is based on such a source-channel model.

The decoding process of a speech recognizer is to find a sequence of words whose corresponding acoustic and language models best match the input signal. Therefore, the process of such a decoding process with trained acoustic and language models is often referred to as just a *search* process. Graph search algorithms have been explored extensively in the fields of artificial intelligence, operation research, and game theory. In this chapter first we present several basic search algorithms, which serve as the basic foundation for CSR.

The complexity of a search algorithm is highly correlated with the search space, which is determined by the constraints imposed by the language models. We discuss the impact of different language models, including finite-state grammars, context-free grammars, and n-grams.

Speech recognition search is usually done with the Viterbi or A* stack decoders. The reasons for choosing the Viterbi decoder involve arguments that point to speech as a left-to-right process and to the efficiencies afforded by a time-synchronous process. The reasons for choosing a stack decoder involve its ability to more effectively exploit the A* criteria, which holds out the hope of performing an optimal search as well as the ability to handle huge search spaces. Both algorithms have been successfully applied to various speech recognition systems. The relative merits of both search algorithms were quite controversial in the 1980s. Lately, with the help of efficient pruning techniques, Viterbi beam search has been the preferred method for almost all speech recognition tasks. Stack decoding, on the other hand, remains an important strategy to uncover the n-best and lattice structures.

12.1. BASIC SEARCH ALGORITHMS

Search is a subject of interest in artificial intelligence and has been well studied for expert systems, game playing, and information retrieval. We discuss several general graph search methods that are fundamental to spoken language systems. Although the basic concept of graph search algorithms is independent of any specific task, the efficiency often depends on how we exploit domain-specific knowledge.

The idea of search implies moving around, examining things, and making decisions about whether the sought object has yet been found. In general, search problems can be represented using the *state-space search* paradigm. It is defined by a triplet (S, O, G), where S is a set of initial states, O a set of operators (or rules) applied on a state to generate a transition with its corresponding cost to another state, and G a set of goal states. A solution in the state-space search paradigm consists in finding a path from an initial state to a goal state. The state-space representation is commonly identified with a directed graph in which each node corresponds to a state and each arc to an application of an operator (or a rule), which transitions from one state to another. Thus, the state-space search is equivalent to searching through the graph with some objective function.

Before we present any graph search algorithms, we need to remind the readers of the importance of the dynamic programming algorithm described in Chapter 8. Dynamic programming should be applied whenever possible and as early as possible because (1) unlike any heuristics, it will not sacrifice optimality; (2) it can transform an exponential search into a polynomial search.

12.1.1. General Graph Searching Procedures

Although dynamic programming is a powerful polynomial search algorithm, many interesting problems cannot be handled by it. A classical example is the traveling salesman's problem. We need to find a shortest-distance tour, starting at one of many cities, visiting each city exactly once, and returning to the starting city. This is one of the most famous problems in the *NP*-hard class [1, 32]. Another classical example is the *N*-queens problem (typically 8-queens), where the goal is to place N queens on an $N \times N$ chessboard in such a way that no queen can capture any other queen, i.e., there is no more than one queen in any given row, column, or diagonal. Many of these puzzles have the same characteristics. As we know, the best algorithms currently known for solving the *NP*-hard problem are exponential in the problem size. Most graph search algorithms try to solve those problems using heuristics to avoid or moderate such a combinatorial explosion.

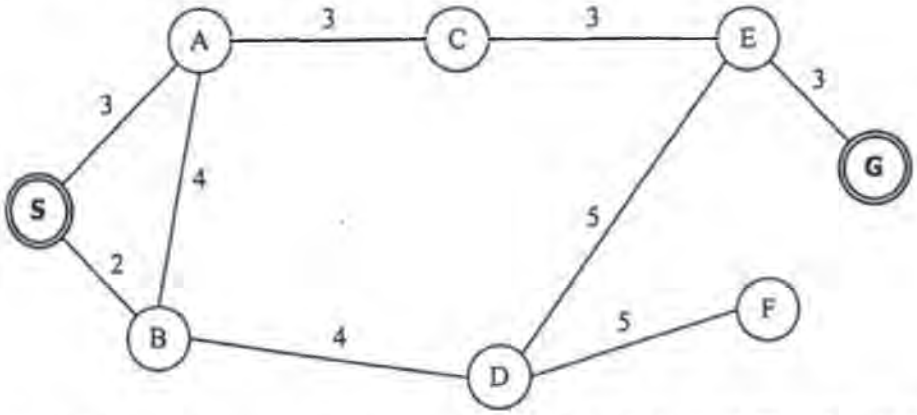


Figure 12.1 A highway distance map for cities S, A, B, C, D, E, F, and G. The salesman needs to find a path to travel from city S to city G [42].

Let's start our discussion of graph search procedure with a simple city-traveling problem [42]. Figure 12.1 shows a highway distance map for all the cities. A salesman named John needs to travel from the starting city S to the end city G. One obvious way to find a path is to derive a graph that allows orderly exploration of all possible paths. Figure 12.2 shows the graph that traces out all possible paths in the city-distance map shown in Figure 12.1. Although the city-city connection is bi-directional, we should note that the search graph in this case must not contain cyclic paths, because they would not lead to any progress in this scenario.

If we define the search space as the potential number of nodes (states) in the graph search procedure, the search space for finding the optimal state sequence in the Viterbi algorithm (described in Chapter 8) is $N \times T$, where N is the number of states for the HMM and T is the length of the observation. Similary, the search space for John's traveling problem will be 27.

Another important measure for a search graph is the *branching factor*, defined as the average number of successors for each node. Since the number of nodes of a search graph

(or tree) grows exponentially with base equal to this branching factor, we certainly need to watch out for search graphs (or trees) with a large branching factor. Sometimes they can be too big to handle (even infinite, as in game playing). We often trade the optimal solution for improved performance and feasibility. That is, the goal for such search problems is to find one satisfactory solution instead of the optimal one. In fact, most AI (artifical intelligence) search problems belong to this category.

The search tree in Figure 12.2 may be implemented either explicitly or implicitly. In an explicit implementation, the nodes and arcs with their corresponding distances (or costs) are explicitly specified by a table. However, an explicit implementation is clearly impractical for large search graphs and impossible for those with infinite nodes. In practice, most parts of the graph may never be explored before a solution is found. Therefore, a sensible strategy is to dynamically generate the search graph. The part that becomes explicit is often referred to as an *active* search space. Throughout the discussion here, it is important to keep in mind this distinction between the implicit search graph that is specified by the start node S and the explicit partial search graphs that are actually constructed by the search algorithm.

To expand the tree, the term *successor operator* (or *move generator*, as it is often called in game search) is defined as an operator that is applied to a node to generate all of the successors of that node and to compute the distance associated with each arc. The successor operator obviously depends on the topology (or rules) of the problem space. Expanding the starting node S, and successors of S, ad infinitum, gradually makes the implicitly

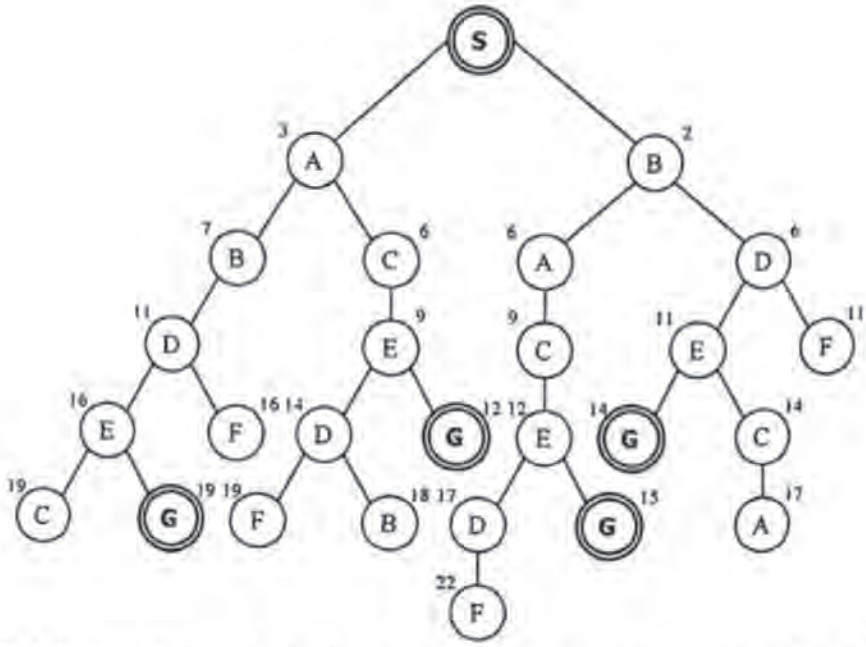


Figure 12.2 The search tree (graph) for the salesman problem illustrated in Figure 12.1. The number next to each node is the accumulated distance from start city to end city [42].

defined graph explicit. This recursive procedure is straightforward, and the search graph (tree) can be constructed without the extra bookkeeping. However, this process would only generate a search tree where the same node might be generated as a part of several possible paths.

For example, node E is being generated in four different paths. If we are interested in finding an optimal path to travel from S to G, it is more efficient to merge those different paths that lead to the same node E. We can pick the shortest path up to C, since everything following E is the same for the rest of the paths. This is consistent with the dynamic programming principle—when looking for the best path from S to G, all partial paths from S to any node E, other than the best path from S to E, should be discarded. The dynamic programming merge also eliminates cyclic paths implicitly, since a cyclic path cannot be the shortest path. Performing this extra bookkeeping (merging different paths leading into the same node) generates a search graph rather than a search tree.

Although a graph search has the potential advantage over a tree search of being more efficient, it does require extra bookkeeping. Whether this effort is justified depends on the individual problem one has to address.

Most search strategies search in a forward direction, i.e., build the search graph (or tree) by starting with the initial configuration (the starting state S) from the root. In the general AI literature, this is referred to as *forward reasoning* [43], because it performs rule-base reasoning by matching the left side of rules first. However, for some specific problem domains, it might be more efficient to use *backward reasoning* [43], where the search graph is built from the bottom up (the goal state G). Possible scenarios include:

- *There are more initial states than goal states.* Obviously it is easy to start with a small set of states and search for paths leading to one of the bigger sets of states. For example, suppose the initial state S is the hometown for John in the city-traveling problem in Figure 12.1 and the goal state G is an unfamiliar city for him. In the absence of a map, there are certainly more locations (neighboring cities) that John can identify as being close[1] to his home city S than those he can identify as being close to an unfamiliar location. In a sense, all of those locations being identified as close to John's home city S are equivalent to the initial state S. This means John might want to consider reasoning backward from the unfamiliar goal city G for the trip planning.
- *The branching factor for backward reasoning is smaller than that for forward reasoning.* In this case it makes sense to search in the direction with lower branching factor.

It is in principle possible to search from both ends simultaneously, until two partial paths meet somewhere in the middle. This strategy is called *bi-directional search* [43]. Bi-directional search seems particularly appealing if the number of nodes at each step grows

[1] *Being close* means that, once John reaches one of those neighboring cities, he can easily remember the best path to return home. It is similar to the killer book for chess play. Once the player reaches a particular board configuration, he can follow the killer book for moves that can guarantee a victory.

exponentially with the depth that needs to be explored. However, sometimes bi-directional search can be devastating. The two searches may cross each other, as illustrated in Figure 12.3.

The process of explicitly generating part of an implicitly defined graph forms the essence of our general graph search procedure. The procedure is summarized in Algorithm 12.1. It maintains two lists: *OPEN*, which stores the nodes waiting for expansion, and *CLOSE*, which stores the already expanded nodes. Steps 6a and 6b are basically the bookkeeping process to merge different paths going into the same node by picking the one that has the minimum distance. Step 6a handles the case where v is in the *OPEN* list and thus is not expanded. The merging process is straightforward, with a single comparison and change of traceback pointer if necessary. However, when v is in the *CLOSE* list and thus is already expanded in Step 6b, the merging requires additional forward propagation of the new score if the current path is found to be better than the best subpath already in the *CLOSE* list. This forward propagation could be very expensive. Fortunately, most of the search strategy can avoid such a procedure if we know that the already expanded node must belong in the best path leading to it. We discuss this in Section 12.5.

As described earlier, it may not be worthwhile to perform bookkeeping for a graph search, so Steps 6a and 6b are optional. If both steps are omitted, the graph search algorithm described above becomes a tree search algorithm. To illustrate different search strategies, tree search is used as the basic graph search algorithm in the sections that follows. However, you should note that all the search methods described here could be easily extended to graph search with the extra bookkeeping (merging) process as illustrated in Steps 6a and 6b of Algorithm 12.1.

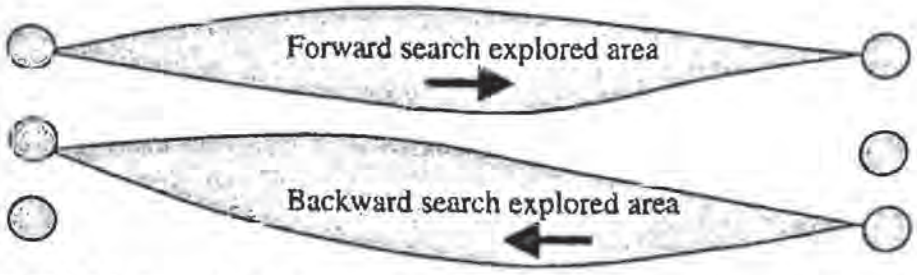


Figure 12.3 A bad case for bi-directional search, where the forward search and the backward search crossed each other [42].

ALGORITHM 12.1: *THE GRAPH-SEARCH ALGORITHM*

Step 1: Initialization: Put S in the $OPEN$ list and create an initially empty $CLOSE$ list.
Step 2: If the $OPEN$ list is empty, exit and declare failure.
Step 3: Pop up the first node N in the $OPEN$ list, remove it from the $OPEN$ list and put it into the $CLOSE$ list.
Step 4: If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.
Step 5: Expand node N by applying the successor operator to generate the successor set $SS(N)$ of node N. Be sure to eliminate the ancestors of N from $SS(N)$.
Step 6: $\forall v \in SS(N)$ do

6a. (optional) If $v \in OPEN$ and the accumulated distance of the new path is smaller than that for the one in the $OPEN$ list, do

(i) change the traceback (parent) pointer of v to N and adjust the accumulated distance for v.

(ii) go to Step 7.

6b. (optional) If $v \in CLOSE$ and the accumulated distance of the new path is smaller than the partial path ending at v in the $CLOSE$ list, do

(i) change the traceback (parent) pointer of v to N and adjust the accumulated distance for all paths that contain v.

(ii) go to Step 7.

6c. Create a pointer pointing to N and push it into the $OPEN$ list.
Step 7: Reorder the $OPEN$ list according to search strategy or some heuristic measurement.
Step 8: Go to Step 2.

12.1.2. Blind Graph Search Algorithms

If the aim of the search problem is to find an acceptable path instead of the best path, blind search is often used. *Blind search* treats every node in the $OPEN$ list the same and blindly decides the order to be expanded without using any domain knowledge. Since blind search treats every node equally, it is often referred to as *uniform search* or *exhaustive search*, because it exhaustively tries out all possible paths. In AI, people are typically not interested in blind search. However, it does provide a lot of insight into many sophisticated heuristic search algorithms. You should note that blind search does not expand nodes randomly. Instead, it follows some systematic way to explore the search graph. Two popular types of blind search are depth-first search and breadth-first search.

12.1.2.1. Depth-First Search

When we are in a maze, the most natural way to find a way out is to mark the branch we take whenever we reach a branching point. The marks allow us to go back to a choice point with an unexplored alternative, withdraw the most recently made choice and undo all consequences of the withdrawn choice whenever a dead-end is reached. Once the alternative choice is selected and marked, we go forward based on the same procedure. This intuitive search strategy is called *backtracking*. The famous N-queens puzzle [32] can be handily solved by the backtracking strategy.

Depth-first search picks an arbitrary alternative at every node visited. The search sticks with this partial path and works forward from the partial path. Other alternatives at the same level are ignored completely (for the time being) in the hope of finding a solution based on the current choice. This strategy is equivalent to ordering the nodes in the *OPEN* list by their depth in the search graph (tree). The deepest nodes are expanded first and nodes of equal depth are ordered arbitrarily.

Although depth-first search hopes the current choice leads to a solution, sometimes the current choice could lead to a dead-end (a node which is neither a goal node nor can be expanded further). In fact, it is desirable to have many short dead-ends. Otherwise, the algorithm may search for a very long time before it reaches a dead-end, or it might not ever reach a solution if the search space is infinite. When the search reaches a dead-end, it goes back to the last decision point and proceeds with another alternative.

Figure 12.4 shows all the nodes being expanded under the depth-first search algorithm for the city-traveling problem illustrated in Figure 12.1. The only differences between the graph search and the depth-first search algorithms are:

1. The graph search algorithm generates all successors at a time (although all except one are ignored first), while depth-first search generates only one successor at a time.
2. The graph search, when successfully finding a path, saves only one path from the starting node to the goal node, while depth-first search in general saves the entire record of the search graph.

Depth-first search could be dangerous because it might search an impossible path that is actually an infinite dead-end. To prevent exploring of paths that are too long, a depth bound can be placed to constrain the nodes to be expanded, and any node reaching that depth limit is treated as a terminal node (as if it had no successor).

The general graph search algorithm can be modified into a depth-first search algorithm as illustrated in Algorithm 12.2.

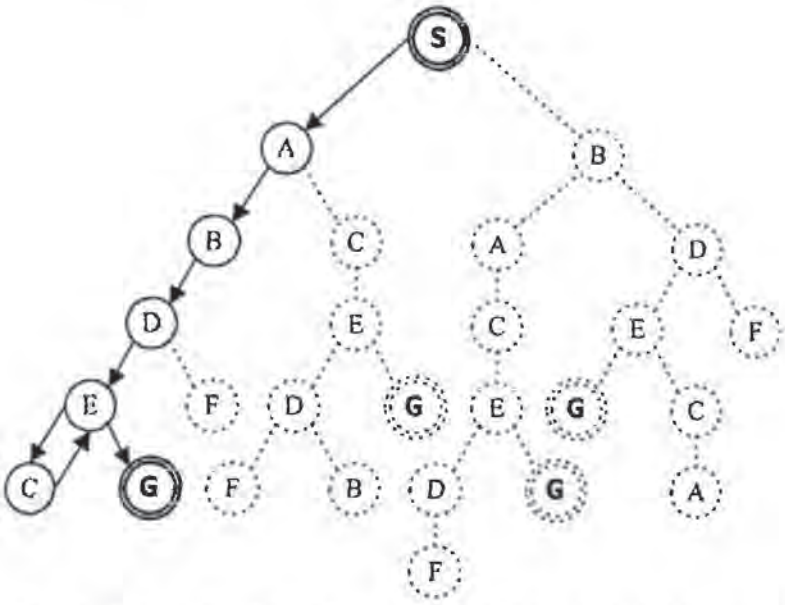


Figure 12.4 The node-expanding procedure of the depth-first search for the path search problem in Figure 12.1. When it fails to find the goal city in node *C*, it backtracks to the parent and continues the search until it finds the goal city. The gray nodes are those that are explored. The dotted nodes are not visited during the search [42].

ALGORITHM 12.2: *THE DEPTH-FIRST SEARCH ALGORITHM*

Step 1: Initialization: Put S in the *OPEN* list and create an initially empty the CLOSE list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.

4a. If the depth of node N is equal to the depth bound, go to Step 2.

Step 5: Expand node N by applying the successor operator to generate the successor set SS(N) of node N. Be sure to eliminate the ancestors of N from SS(N).
Step 6: $\forall v \in SS(N)$ do

6c. Create a pointer pointing to N and push it into the *OPEN* list.

Step 7: Reorder the the *OPEN* list in descending order of the depth of the nodes.
Step 8: Go to Step 2.

12.1.2.2. Breadth-First Search

One natural alternative to the depth-first search strategy is breadth-first search. *Breadth-first search* examines all the nodes on one level before considering any of the nodes on the next level (depth). As shown in Figure 12.5, node *B* would be examined just after node *A*. The search moves on level-by-level, finally discovering G on the fourth level.

Breadth-first search is guaranteed to find a solution if one exists, assuming that a finite number of successors (branches) always follow any node. The proof is straightforward. If there is a solution, its path length must be finite. Let's assume the length of the solution is *M*. Breadth-first search explores all paths of the same length increasingly. Since the number of paths of fixed length *N* is always finite, it eventually explores all paths of length *M*. By that time it should find the solution.

It is also easy to show that a breadth-first search can work on a search tree (graph) with infinite depth on which an unconstrained depth-first search will fail. Although a breadth-first might not find a shortest-distance path for the city-travel problem, it is guaranteed to find the one with fewest cities visited (minimum-length path). In some cases, it is a very desirable solution. On the other hand, a breadth-first search may be highly inefficient when all solutions leading to the goal node are at approximately the same depth. The breadth-first search algorithm is summarized in Algorithm 12.3.

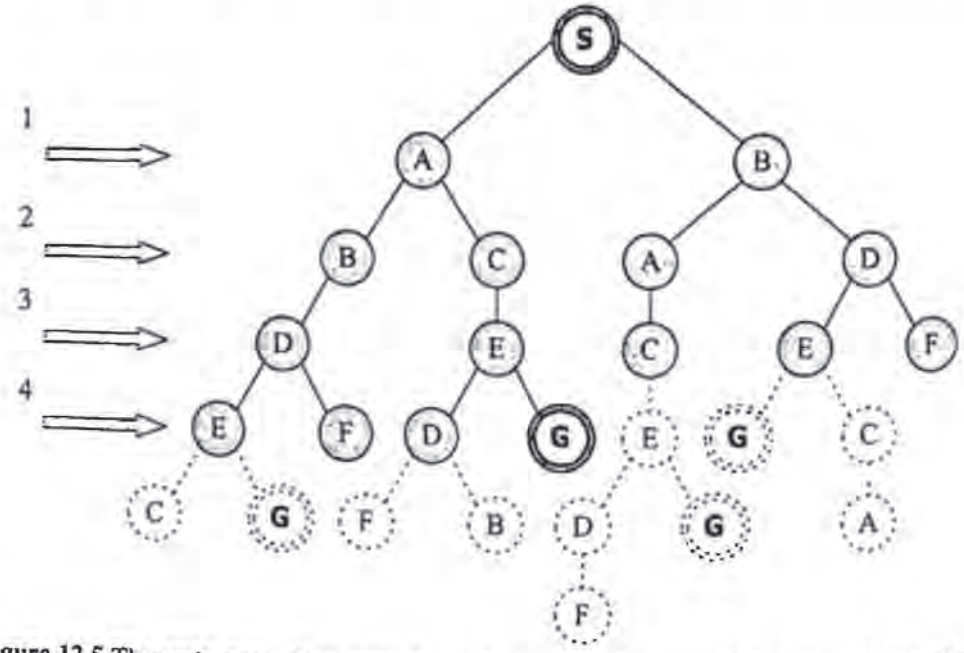


Figure 12.5 The node-expanding procedure of a breadth-first search for the path search problem in Figure 12.1. It searches through each level until the goal is identified. The gray nodes are those that are explored. The dotted nodes are not visited during the search [42].

ALGORITHM 12.3: *THE BREADTH-FIRST SEARCH ALGORITHM*

Step 1: Initialization: Put S in the *OPEN* list and create an initially empty the *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.
Step 5: Expand node N by applying the successor operator to generate the successor set $SS(N)$ of node N. Be sure to eliminate the ancestors of N, from $SS(N)$.
Step 6: $\forall v \in SS(N)$ do

6c. Create a pointer pointing to N and push it into the *OPEN* list.

Step 7: Reorder the *OPEN* list in increasing order of the depth of the nodes.
Step 8. Go to Step 2.

12.1.3. Heuristic Graph Search

Blind search methods, like depth-first search and breadth-first search, have no sense (or guidance) of where the goal node lies ahead. Consequently, they often spend a lot of time searching in hopeless directions. If there is guidance, the search can move in the direction that is more likely to lead to the goal. For example, you may want to find a driving route to the World Trade Center in New York. Without a map at hand, you can still use a straight-line distance estimated by eye as a hint to see if you are closer to the goal (World Trade Center). This *hill-climbing* style of guidance can help you to find the destination much more efficiently.

Blind search finds only one arbitrary solution instead of the optimal solution. To find the optimal solution with depth-first or breadth-first search, you must not stop searching when the first solution is discovered. Instead, the search needs to continue until it reaches all the solutions, so you can compare them to pick the best. This strategy for finding the optimal solution is called *British Museum search* or *brute-force search*. Obviously, it is unfeasible when the search space is large. Again, to conduct selective search and yet still be able to find the optimal solution, some guidance on the search graph is necessary.

The guidance obviously comes from domain-specific knowledge. Such knowledge is usually referred to as *heuristic* information, and search methods taking advantage of it are called *heuristic search* methods. There is usually a wide variety of different heuristics for the problem domain. Some heuristics can reduce search effort without sacrificing optimality, while other can greatly reduce search effort but provide only sub-optimal solutions. In most practical problems, the choice of different heuristics is usually a tradeoff between the quality of the solution and the cost of finding the solution.

Heuristic information works like an evaluation function $h(N)$ that maps each node N to a real number, and which serves to indicate the relative goodness (or cost) of continuing the search path from that node. Since in our city-travel problem, straight-line distance is a natural way of measuring the goodness of a path, we can use the heuristic function $h(N)$ for the distance evaluation as:

$h(N)$=Heuristic estimate of the remaining distance from node N to goal G (12.1)

Since $g(N)$, the distance of the partial path to the current node N, is generally known, we have:

$g(N)$=The distance of the partial path already traveled from root S to node N (12.2)

We can define a new heuristic function, $f(N)$, which estimates the total distance for the path (not yet finished) going through node N.

$$f(N) = g(N) + h(N) \tag{12.3}$$

A heuristic search method basically uses the heuristic function $f(N)$ to re-order the *OPEN* list in the Step 7 of Algorithm 12.1. The node with the best heuristic value is explored first (expanded first). Some heuristic search strategies also prune some unpromising partial paths forever to save search space. This is why heuristic search is often referred to as heuristic pruning.

The choice of the heuristic function is critical to the search results. If we use one that overestimates the distance of some nodes, the search results may be suboptimal. Therefore, heuristic functions that do not overestimate the distance are often used in search methods aiming to find the optimal solution.

To close this section, we describe two of the most popular heuristic search methods: best-first (or A* Search) [32, 43] and beam search [43]. They are widely used in many components of spoken language systems.

12.1.3.1. Best-First (A* Search)

Once we have a reasonable heuristic function to evaluate the goodness of each node in the *OPEN* list, we can explore the best node (the node with smallest $f(N)$ value) first, since it offers the best hope of leading to the best path. This natural search strategy is called *best-first search*. To implement best-first search based on the Algorithm 12.1, we need to first evaluate $f(N)$ for each successor before putting the successors in the *OPEN* list in Step 6. We also need to sort the elements in the *OPEN* list based on $f(N)$ in Step 7, so that the best node is in the front-most position waiting to be expanded in Step 3. The modified procedure for performing best-first search is illustrated in Algorithm 12.4. To avoid duplicating nodes in the *OPEN* list, we include Steps 6a and 6b to take advantage of the dynamic programming principle. They perform the needed bookkeeping process to merge different paths leading into the same node.

ALGORITHM 12.4: *THE BEST-FIRST SEARCH ALGORITHM*

Step 1: Initialization: Put S in the *OPEN* list and create an initially empty the *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3. Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.
Step 5: Expand node N by applying the successor operator to generate the successor set $SS(N)$ of node N. Be sure to eliminate the ancestors of N, from $SS(N)$.
Step 6: $\forall v \in SS(N)$ do

6a. (optional) If $v \in OPEN$ and the accumulated distance of the new path is smaller than that for the one in the the *OPEN* list, do

(i) Change the traceback (parent) pointer of v to N and adjust the accumulated distance for v.

(ii) Evaluate heuristic function $f(v)$ for v and go to Step 7.

6b. (optional) If $v \in CLOSE$ and the accumulated distance of the new path is small than the partial path ending at v in the the *CLOSE* list,

(i) Change the traceback (parent) pointer of v to N and adjust the accumulated distance and heuristic function f for all the paths containing v.

(ii) go to Step 7.

6c. Create a pointer pointing to N and push it into the *OPEN* list.
Step 7: Reorder the the *OPEN* list in the increasing order of the heuristic function $f(N)$.
Step 8: Go to Step 2.

A search algorithm is said to be *admissible* if it can guarantee to find an optimal solution, if one exists. Now we show that if the heuristic function $h(N)$ of estimating the remaining distance from N to goal node G is an underestimate[2] of the true distance from N to goal node G, the best-first search illustrated in Algorithm 12.4 is admissible. In fact, when $h(N)$ satisfies the above criterion, the best-first algorithm is called A* (pronounced as /eh/-star) Search.

The proof can be carried out informally as follows. When the frontmost node in the *OPEN* list is the goal node G in Step 4, it immediately implies that

$$\forall v \in OPEN \quad f(v) \geq f(G) = g(G) + h(G) = g(G) \tag{12.4}$$

[2] For admissibility, we actually require only that the heuristic function not overestimate the distance from N to G. Since it is very rare to have an exact estimate, we use underestimate throughout this chapter without loss of generality. Sometimes we refer to an underestimate function as a lower-bound estimate of the true value.

Equation (12.4) says that the distance estimate of any incomplete path is no shorter than the first found complete path. Since the distance estimate for any incomplete path is underestimated, the first found complete path in Step 4 must be the optimal path. A similar argument can also be used to prove that the Step 6b is actually not necessary for admissible heuristic functions; that is, there cannot be another path with a shorter distance from the starting node to a node that has been expanded. This is a very important feature since Step 6b is, in general, very expensive and it requires significant updates of many already expanded paths.

The A* search method is actually a family of search algorithms. When $h(N) = 0$ for all N, the search degenerates into an uninformed search[3] [40]. In fact, this type of uninformed search is the famous *branch-and-bound search* algorithm that is often used in many *operations research* problems. Branch-and-bound search always expands the shortest path leading into an open node until there is a path reaching the goal that is of a length no longer than all incomplete paths terminating at open nodes. When $g(N)$ is defined as the depth of the node N, the use of heuristic function $f(N)$ makes the search method identical to breadth-first search. In Section 12.1.2.2, we mention that breadth-first search is guaranteed to find a minimum length path. This can certainly be derived from the admissibility of the A* search method.

When the heuristic function is close to the true remaining distance, the search can usually find the optimal solution without too much effort. In fact, when the true remaining distances for all nodes are known, the search can be done in a totally greedy fashion without any search at all, i.e., the only path explored is the solution. Any non-zero heuristic function is then called an informed heuristic function, and the search using such a function is called informed search. A heuristic function h_1 is said to be more informed than a heuristic function h_2 if the estimate h_1 is everywhere larger than h_2 and yet still admissible (underestimate). Finding an informed admissible heuristic function (guaranteed to underestimate for all nodes) is, in general, a difficult task. The heuristic often requires extensive analysis of the domain-specific knowledge and knowledge representation.

Let's look at a simple example—the 8-puzzle problem. The 8-puzzle consists of eight numbered, movable tiles set in a 3×3 frame. One cell of this frame is always empty, so it is possible to move an adjacent numbered tile into the empty cell. A solution for the 8-puzzle is to find a sequence of moves to change the initial configuration into a given goal configuration as shown in Figure 12.6. One choice for an informed admissible heuristic function h_1 is the number of misplaced tiles associated with the current configuration. Since each misplaced tile needs to move at least once to be in the right position, this heuristic function is clearly a lower bound of the true movements remaining. Based on this heuristic function, the value for the initial configuration will be 7 in Figure 12.7. If we examine this problem further, a more informed heuristic function h_2 can be defined as the sum of all row and column distances of all misplaced tiles and their goal positions. For example, the row and column distance between the tile 8 in the initial configuration and the goal position is 2 + 1= 3,

[3] In some literature an uninformed search is referred to as uniform-cost search.

Figure 12.6 Initial and goal configurations for the 8-puzzle problem.

which indicates that one must move tile 8 at least 3 times in order for it to be in the right position. Based on the heuristic function h_2, the value for the initial configuration will be 16 in Figure 12.6. h_2 is again admissible.

In our city-travel problem, one natural choice for the underestimating heuristic function of the remaining distance between node N and goal G is the straight-line distance since the true distance must be no shorter than the straight-line distance.

Figure 12.7 shows an augmented city-distance map with straight-line distance to goal node attached to each node. Accordingly, the heuristic search tree can be easily constructed for improved efficiency. Figure 12.8 shows the search progress of applying the A* search algorithm for the city-traveling problem by using the straight-line distance heuristic function to estimate the remaining distances.

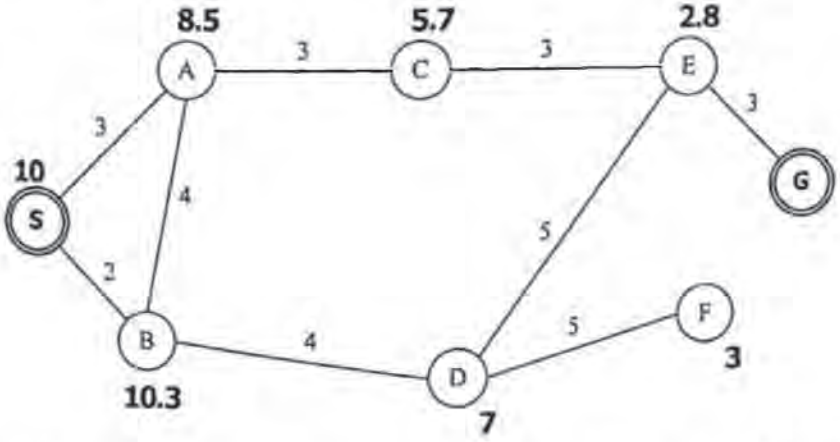


Figure 12.7 The city-travel problem augmented with heuristic information. The numbers beside each node indicate the straight-line distance to the goal node G [42].

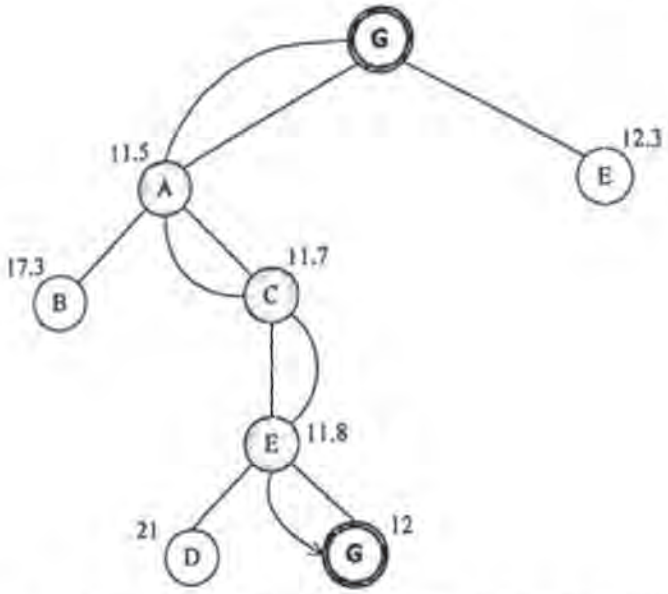


Figure 12.8 The search progress of applying A* search for the city-travel problem. The search determines that path S-A-C-E-G is the optimal one. The number beside the node is f values on which the sorting of the *OPEN* list is based [42].

12.1.3.2. Beam Search

Sometimes, it is impossible to find any effective heuristic estimate, as required in A* search, particularly when there is very little (or no) information about the remaining paths. For example, in real-time speech recognition, there is little information about what the speaker will utter for the remaining speech. Therefore, an efficient uninformed search strategy is very important to tackle this type of problem.

Breadth-first style search is an important strategy for heuristic search. A breadth-first search virtually explores all the paths with the same depth before exploring deeper paths. In practice, paths of the same depth are often easier to compare. It requires fewer heuristics to rank the goodness of each path. Even with uninformed heuristic function ($h(N)=0$), the direct comparison of g (distance so far) of the paths with the same length should be a reasonable choice.

Beam search is a widely used search technique for speech recognition systems [26, 31, 37]. It is a breadth-first style search and progresses along with the depth. Unlike traditional breadth-first search, however, beam search only expands nodes that are likely to succeed at each level. Only these nodes are kept in the beam, and the rest are ignored (pruned) for improved efficiency.

In general, a beam search only keeps up to w best paths at each stage (level), and the rest of the paths are discarded. The number w is often referred to as beam width. The number of nodes explored remains manageable in beam search even if the whole search space is gigantic. If a beam width w is used in a beam search with an average branching factor b, only $w \times b$ nodes need to be explored at any depth, instead of the exponential number

needed for breadth-first search. Suppose that a beam width of 2 is used for the city-travel problem. Figure 12.9 illustrates how beam search progresses to find the path. We can also see that the beam search saved a large number of unneeded nodes, as shown by the dotted nodes.

The beam search algorithm can be easily modified from the breadth-first search algorithm and is illustrated in Algorithm 12.5. For simplicity, we do not include the merging step here. In Algorithm 12.5, Step 4 obviously requires sorting, which is time-consuming if the number $w \times b$ is huge. In practice, the beam is usually implemented as a flexible list where nodes are expanded if their heuristic functions $f(N)$ are within some threshold (a.k.a., beam threshold) of the best node (the smallest value) at the same level. Thus, we only need to identify the best node and then prune away nodes that are outside of the threshold. Although this makes the beam size change dynamically, it significantly reduces the effort for sorting of the *Beam-Candidate* list. In fact, by adjusting the beam threshold, the beam size can be controlled indirectly and yet kept manageable.

Unlike A′ search, beam search is an approximate heuristic search method that is not admissible. However, it has a number of unique merits. Because of its simplicity in both its search strategy and its requirement of domain-specific heuristic information, it has become one of the most popular methods for complicated speech recognition problems. It is particularly attractive when integration of different knowledge sources is required in a time-synchronous fashion. It has the advantages of providing a consistent way of exploring nodes level by level and of offering minimally needed communication between different paths. It is also very suitable for parallel implementation because of its breadth-first search nature.

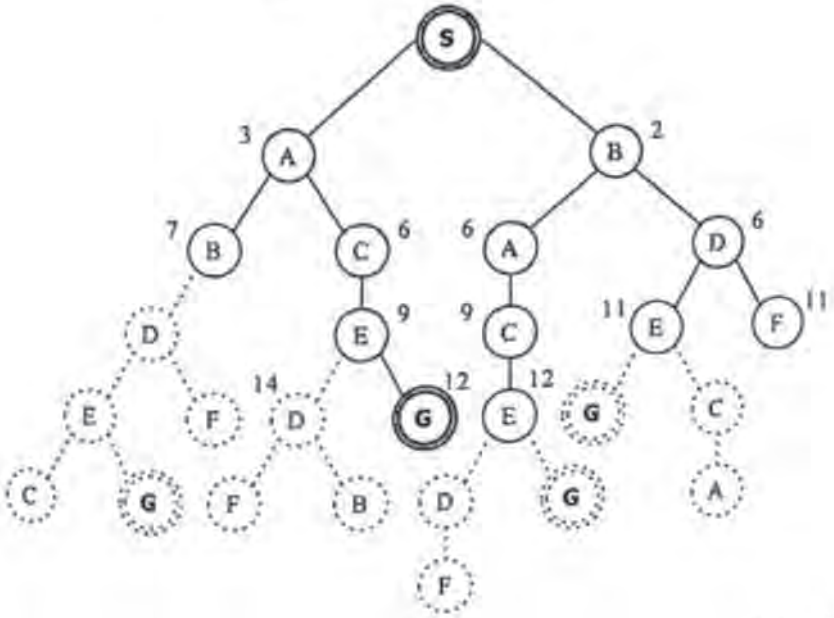


Figure 12.9 Beam search for the city-travel problem. The nodes with gray color are the ones kept in the beam. The transparent nodes were explored but pruned because of higher cost. The dotted nodes indicate all the savings because of pruning [42].

ALGORITHM 12.5: *THE BEAM SEARCH ALGORITHM*

Step 1: Initialization: Put S in the *OPEN* list and create an initially empty *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: $\forall N \in OPEN$ do
3a. Pop up node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
3b. If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.
3c. Expand node N by applying a successor operator to generate the successor set $SS(N)$ of node N. Be sure to eliminate the successors, which are ancestors of N, from $SS(N)$.
3d. $\forall v \in SS(N)$ Create a pointer pointing to N and push it into *Beam-Candidate* list.
Step 4: Sort the *Beam-Candidate* list according to the heuristic function $f(N)$ so that the best w nodes can be pushed into the the *OPEN* list. Prune the rest of nodes in the *Beam-Candidate* list.
Step 5: Go to Step 2.

12.2. SEARCH ALGORITHMS FOR SPEECH RECOGNITION

As described in Chapter 9, the decoder is basically a search process to uncover the word sequence $\hat{\mathbf{W}} = w_1 w_2 ... w_m$ that has the maximum posterior probability $P(\mathbf{W}|\mathbf{X})$ for the given acoustic observation $\mathbf{X} = X_1 X_2 ... X_n$. That is,

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{W}} P(\mathbf{W}|\mathbf{X}) = \arg\max_{\mathbf{W}} \frac{P(\mathbf{W})P(\mathbf{X}|\mathbf{W})}{P(\mathbf{X})} = \arg\max_{\mathbf{W}} P(\mathbf{W})P(\mathbf{X}|\mathbf{W}) \qquad (12.5)$$

One obvious way is to search all possible word sequences and select the one with the best posterior probability score.

The unit of acoustic model $P(\mathbf{X}|\mathbf{W})$ is not necessary a word model. For large-vocabulary speech recognition systems, subword models, which include phonemes, demisyllables, and syllables are often used. When subword models are used, the word model $P(\mathbf{X}|\mathbf{W})$ is then obtained by concatenating the subword models according to the pronunciation transcription of the words in a lexicon or dictionary.

When word models are available, speech recognition becomes a search problem. The goal for speech recognition is thus to find a sequence of word models that best describes the input waveform against the word models. As neither the number of words nor the boundary of each word or phoneme in the input waveform is known, appropriate search strategies to deal with these variable-length nonstationary patterns are extremely important.

When HMMs are used for speech recognition systems, the states in the HMM can be expanded to form the state-search space in the search. In this chapter, we use HMMs as our speech models. Although the HMM framework is used to describe the search algorithms, all

techniques mentioned in this and the following chapter can be used for systems based on other modeling techniques, including template matching and neural networks. In fact, many search techniques had been invented before HMMs were applied to speech recognition. Moreover, the HMMs state transition network is actually general enough to represent the general search framework for all modeling approaches.

12.2.1. Decoder Basics

The lessons learned from dynamic programming or the Viterbi algorithm introduced in Chapter 8 tell us that the exponential blind search can be avoided if we can store some intermediate optimal paths (results). Those intermediate paths are used for other paths without being recomputed each time. Moreover, the beam search described in the previous section shows us that efficient search is possible if appropriate pruning is employed to discard highly unlikely paths. In fact, all the search techniques use two strategies: sharing and pruning. *Sharing* means that intermediate results can be kept, so that they can be used by other paths without redundant re-computation. *Pruning* means that unpromising paths can be discarded reliably without wasting time in exploring them further.

Search strategies based on dynamic programming or the Viterbi algorithm with the help of clever pruning, have been applied successfully to a wide range of speech recognition tasks [31], ranging from small-vocabulary tasks, like digit recognition, to unconstraint large-vocabulary (more than 60,000 words) speech recognition. All the efficient search algorithms we discuss in this chapter and the next are considered as variants of dynamic programming or the Viterbi search algorithm.

In Section 12.1, cost (distance) is used as the measure of goodness for graph search algorithms. With Bayes' formulation, searching the minimum-cost path (word sequence) is equivalent to finding the path with maximum probability. For the sake of consistency, we use the inverse of Bayes' posterior probability as our objective function. Furthermore, logarithms are used on the inverse posterior probability to avoid multiplications. That is, the following new criterion is used to find the optimal word sequence $\hat{\mathbf{W}}$:

$$C(\mathbf{W}|\mathbf{X}) = \log\left[\frac{1}{P(\mathbf{W})P(\mathbf{X}|\mathbf{W})}\right] = -\log\left[P(\mathbf{W})P(\mathbf{X}|\mathbf{W})\right] \tag{12.6}$$

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} C(\mathbf{W}|\mathbf{X}) \tag{12.7}$$

For simplicity, we also define the following cost measures to mirror the likelihood for acoustic models and language models:

$$C(\mathbf{X}|\mathbf{W}) = -\log\left[P(\mathbf{X}|\mathbf{W})\right] \tag{12.8}$$

$$C(\mathbf{W}) = -\log\left[P(\mathbf{W})\right] \tag{12.9}$$

12.2.2. Combining Acoustic and Language Models

Although Bayes' equation [Eq. (12.5)] suggests that the acoustic model probability (conditional probability) and language model probability (prior probability) can be combined through simple multiplication, in practice some weighting is desirable. For example, when HMMs are used for acoustic models, the acoustic probability is usually underestimated, owing to the fallacy of the Markov and independence assumptions. Combining the language model probability with an underestimated acoustic model probability according to Eq. (12.5) would give the language model too little weight. Moreover, the two quantities have vastly different dynamic ranges particularly when continuous HMMs are used. One way to balance the two probability quantities is to add a *language model weight* LW to raise the language model probability $P(\mathbf{W})$ to that power $P(\mathbf{W})^{LW}$ [4, 25]. The language model weight LW is typically determined empirically to optimize the recognition performance on a development set. Since the acoustic model probabilities are underestimated, the language model weight LW is typically >1.

Language model probability has another function as a penalty for inserting a new word (or existing words). In particular, when a uniform language model (every word has an equal probability for any condition) is used, the language model probability here can be viewed as purely the penalty of inserting a new word. If this penalty is large, the decoder will prefer fewer longer words in general, and if this penalty is small, the decoder will prefer a greater number of shorter words instead. Since varying the language model weight to match the underestimated acoustic model probability will have some side effect of adjusting the penalty of inserting a new word, we sometimes use another independent *insertion penalty* to adjust the issue of longer or short words. Thus the language model contribution becomes:

$$P(\mathbf{W})^{LW} IP^{N(\mathbf{W})} \tag{12.10}$$

where IP is the insertion penalty (generally $0 < IP \leq 1.0$) and $N(\mathbf{W})$ is the number of words in sentence $\mathbf{W}$. According to Eq. (12.10), insertion penalty is generally a constant that is added to the negative-logarithm domain when extending the search to another new word. In Chapter 9, we described how to compute errors in a speech recognition system and introduced three types of error: substitutions, deletions and insertions. Insertion penalty is so named because it usually affects only insertions. Similar to language model weight, the insertion penalty is determined empirically to optimize the recognition performance on a development set.

12.2.3. Isolated Word Recognition

With isolated word recognition, word boundaries are known. If word HMMs are available, the acoustic model probability $P(\mathbf{X}|\mathbf{W})$ can be computed using the forward algorithm introduced in Chapter 8. The search becomes a simple pattern recognition problem, and the word

$\hat{W}$ with highest forward probability is then chosen as the recognized word. When subword models are used, word HMMs can be easily constructed by concatenating corresponding phoneme HMMs or other types of subword HMMs according to the procedure described in Chapter 9.

12.2.4. Continuous Speech Recognition

Search in continuous speech recognition is rather complicated, even for a small vocabulary, since the search algorithm has to consider the possibility of each word starting at any arbitrary time frame. Some of the earliest speech recognition systems took a two-stage approach towards continuous speech recognition, first hypothesizing the possible word boundaries and then using pattern matching techniques for recognizing the segmented patterns. However, due to significant cross-word co-articulation, there is no reliable segmentation algorithm for detecting word boundaries other than doing recognition itself.

Let's illustrate how you can extend the isolated-word search technique to continuous speech recognition by a simple example, as shown in Figure 12.10. This system contains only two words, w_1 and w_2. We assume the language model used here is an uniform unigram ($P(w_1) = P(w_2) = 1/2$).

It is important to represent the language structures in the same HMM framework. In Figure 12.10, we add one starting state S and one collector state C. The starting state has a null transition to the initial state of each word HMM with corresponding language model probability (1/2 in this case). The final state of each word HMM has a null transition to the collector state. The collector state then has a null transition back to the starting state in order to allow recursion. Similar to the case of embedding the phoneme (subword) HMMs into the word HMM for isolated speech recognition, we can embed the word HMMs for w_1 and w_2 into a new HMM corresponding to structure in Figure 12.10. Thus, the continuous speech search problem can be solved by the standard HMM formulations.

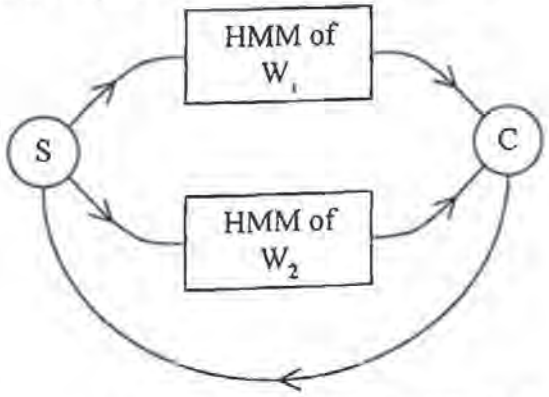


Figure 12.10 A simple example of continuous speech recognition task with two words w_1 and w_2. A uniform unigram language model is assumed for these words. State S is the starting state while state C is a collector state to save fully expanded links between every word pair.

The composite HMMs shown in Figure 12.10 can be viewed as a stochastic finite state network with transition probabilities and output distributions. The search algorithm is essentially producing a match between the acoustic observation $\mathbf{X}$ and a path[4] in the stochastic finite state network. Unlike isolated word recognition, continuous speech recognition needs to find the optimal word sequence $\hat{\mathbf{W}}$. The Viterbi algorithm is clearly a natural choice for this task since the optimal state sequence $\hat{\mathbf{S}}$ corresponds to the optimal word sequence $\hat{\mathbf{W}}$. Figure 12.11 shows the HMM Viterbi trellis computation for the two-word continuous speech recognition example in Figure 12.10. There is a cell for each state in the stochastic finite state network and each time frame t in the trellis. Each cell $C_{s,t}$ in the trellis can be connected to a cell corresponding to time t or $t+1$ and to states in the stochastic finite state network that can be reached from s. To make a word transition, there is a null transition to connect the final state of each word HMM to the initial state of the next word HMM that can be followed. The trellis computation is done *time-synchronously* from left to right, i.e., each cell for time t is completely computed before proceeding to time $t+1$.

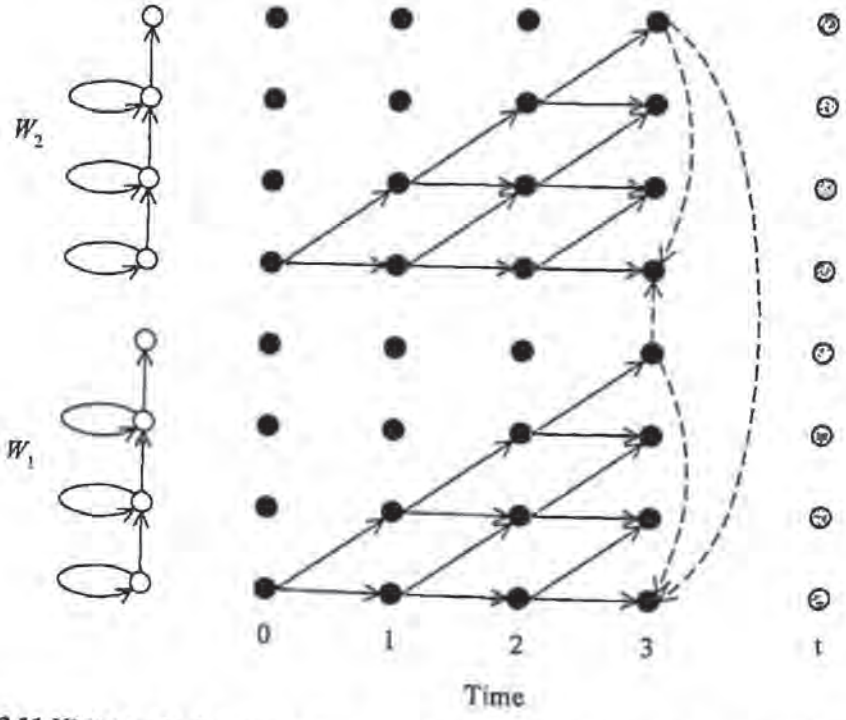


Figure 12.11 HMM trellis for continuous speech recognition example in Figure 12.10. When the final state of the word HMM is reached, a null arc (indicated by a dashed line) is linked from it to the initial state of the following word.

[4] A path here means a sequence of states and transitions.

12.3. LANGUAGE MODEL STATES

The state-space is a good indicator of search complexity. Since the HMM representation for each word in the lexicon is fixed, the state-space is determined by the language models. According to Chapter 11, every language model (grammar) is associated with a state machine (automata). Such a state machine is expanded to form the state-space for the recognizer. The states in such a state machine are referred to as *language models states*. For simplicity, we will use the concepts of state-space and language model states interchangeably. The expansion of language model states to HMM states will be done implicitly. The language model states for isolated word recognition are trivial. They are just the union of the HMM states of each word. In this section we look at the language model states for various grammars for continuous speech recognition.

12.3.1. Search Space with FSM and CFG

As described in Chapter 8, the complexity for the Viterbi algorithm is $O(N^2T)$, where N is the total number of states in the composite HMM and T is the length of input observation. A full time-synchronous Viterbi search is quite efficient for moderate tasks (vocabulary $\leq$ 500). We have already demonstrated in Figure 12.11 how to search for a two-word continuous speech recognition task with a uniform language model. The uniform language model, which allows all words in the vocabulary to follow every word with the same probability, is suitable for connected-digit task. In fact, most small vocabulary tasks in speech recognition applications usually use a finite state grammar (FSG).

Figure 12.12 shows a simple example of an FSM. Similar to the process described in Sections 12.2.3 and 12.2.4, each of the word arcs in an FSG can be expanded as a network of phoneme (subword) HMMs. The word HMMs are connected with null transitions with the grammar state. A large finite state HMM network that encodes all the legal sentences can be constructed based on the expansion procedure. The decoding process is achieved by performing a time-synchronous Viterbi search on this composite finite state HMM.

In practice, FSGs are sufficient for simple tasks. However, when an FSG is made to satisfy the constraints of sharing of different sub-grammars for compactness and support for dynamic modifications, the resulting non-deterministic FSG is very similar to context-free grammar (CFG) in terms of implementation. The CFG grammar consists of a set of productions or rules, which 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, names, and commands. The terminals are words in the vocabulary. A grammar also has a non-terminal designated as its start state.

Although efficient parsing algorithms, like chart parsing (described in Chapter 11), are available for CFG, they are not suitable for speech recognition, which requires left-to-right processing. A context-free grammar can be formulated with a recursive transition network (RTN). RTNs are more powerful and complicated than the finite state machines described in

Chapter 11 because they allow arc labels to refer to other networks as well as words. We use Figure 12.13 to illustrate how to embed HMMs into a recursive transition network.

Figure 12.13 is an RTN representation of the following CFG:

```
S→ NP VP
NP→ sam | sam davis
VP → VERB tom
VERB → likes | hates
```

There are three types of arcs in an RTN, as shown in Figure 12.13: CAT(x), PUSH (x), and POP(x). The CAT(x) arc indicates that x is a terminal node (which is equivalent to a word arc). Therefore, all the CAT(x) arcs can be expanded by the HMM network for x. The word HMM can again be a composite HMM built from phoneme (or subword) HMMs. Similar to the finite state grammar case in Figure 12.12, each grammar state acts as a state with incoming and outgoing null transitions to connect word HMMs in the CFG.

During decoding, the search 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 CFG module to extend the path further by one word. There may be several alternative successor words for the given path. The decoder considers all the successor word possibilities. This may cause the path to be extended to generate several more paths to be considered, each with its own grammar state.

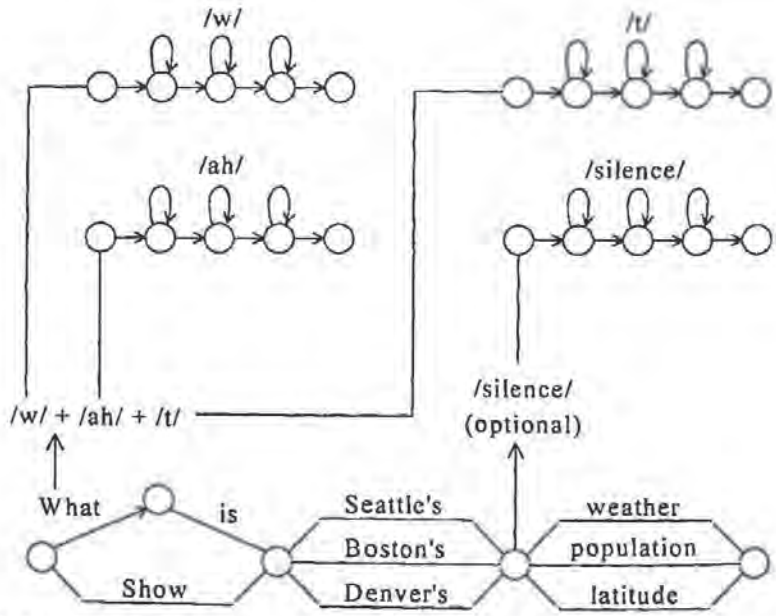


Figure 12.12 An illustration of how to compile a speech recognition task with finite state grammar into a composite HMM.

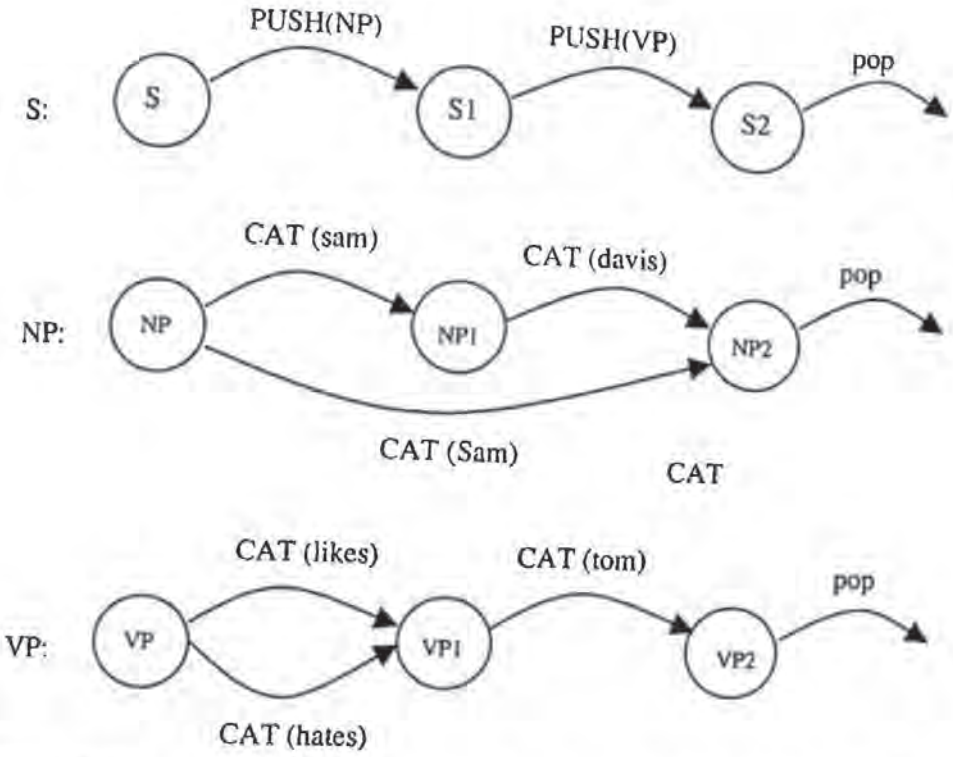


Figure 12.13 A simple RTN example with three types of arcs: CAT(x), PUSH (x), POP.

Readers should 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. For example, there are two word arcs CAT (Sam) in Figure 12.13. Their HMM states should be considered as distinct states in the trellis because they are in completely different grammar states. Two different states in the trellis also means that different paths going into these two states cannot be merged. Since these two partial paths will lead to different successive paths, the search decision needs to be postponed until the end of search. Therefore, when embedding HMMs into word arcs in the grammar network, the HMM state will be assigned a new state identity, although the HMM parameters (transition probabilities and output distributions) can still be shared across different grammar arcs.

Each path consists of a stack of production rules. Each element of the stack also contains the position within the production rule of the symbol that is currently being explored. The search graph (trellis) started from the initial state of CFG (state S). When the path needs to be extended, we look at the next arc (symbol in CFG) in the production. When the search enters a CAT(x) arc (terminal), the path gets extended with the terminal, and the HMM trellis computation is performed on the CAT(x) arc to match the model x against the acoustic data. When the final state of the HMM for x is reached, the search moves on via the null

transition to the destination of the CAT(x) arc. When the search enters a PUSH(x) arc, it indicates a nonterminal symbol x is encountered. In effect, the search is about to enter a subnetwork of x; the destination of the PUSH(x) arc is stored in a last-in first-out (LIFO) stack. When the search reaches a POP arc that signals the end of the current network, the control should jump back to the calling network. In other words, the search returns to the state extracted from the top of the LIFO stack. Finally, when we reach the end of the production rule at the very bottom of the stack, we have reached an accepting state in which we have seen a complete grammatical sentence. For our decoding purpose, that is the state we want to pick as the best score at the end of time frame T to get the search result.

The problem of connected word recognition by finite state or context-free grammars is that the number of states increases enormously when it is applied to more complex grammars. Moreover it remains a challenge to generate such FSGs or CFGs from a large corpus, either manually or automatically. As mentioned in Chapter 11, it is questionable whether FSG or CFG is adequate to describe natural languages or unconstrained spontaneous languages. Instead, n-gram language models are often used for natural languages or unconstrained spontaneous languages. In the next section we investigate how to integrate various n-grams into continuous speech recognition.

12.3.2. Search Space with the Unigram

The simplest n-gram is the unigram that is memory-less and depends only on the current word.

$$P(\mathbf{W}) = \prod_{i=1}^{n} P(w_i) \tag{12.11}$$

Figure 12.14 shows such a unigram grammar network. The final state of each word HMM is connected to the collector state by a null transition, with probability 1.0. The collector state is then connected to the starting state by another null transition, with transition probability equal to 1.0. For word expansion, the starting state is connected to the initial state of each word HMM by a null transition, with transition probability equal to the corresponding unigram probability. Using the collector state and starting state for word expansion allows efficient expansion because it first merges all the word-ending paths[5] (only the best one survives) before expansion. It can cut the total cross-word expansion from N^2 to N.

[5] In graph search, a partial path still under consideration is also referred to as a theory, although we will use paths instead of theories in this book.

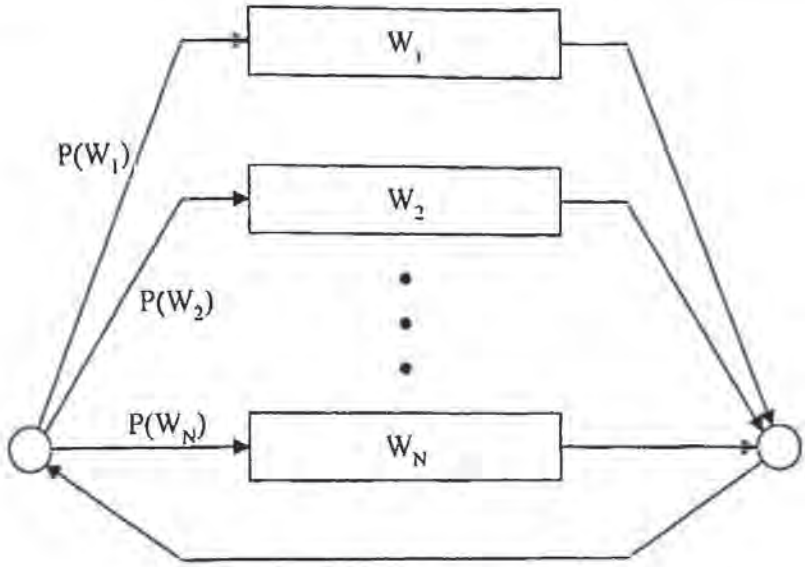


Figure 12.14 A unigram grammar network where the unigram probability is attached as the transition probability from starting state *S* to the first state of each word HMM.

12.3.3. Search Space with Bigrams

When the bigram is used, the probability of a word depends only on the immediately preceding word. Thus, the language model score is:

$$P(\mathbf{W}) = P(w_1 \mid <\text{s}>)\prod_{i=2}^{n} P(w_i \mid w_{i-1}) \tag{12.12}$$

where <s> represents the symbol of starting of a sentence.

Figure 12.15 shows a grammar network using a bigram language model. Because of the bigram constraint, the merge-and-expand framework for unigram search no longer applies here. Instead, the bigram search needs to perform expand-and-merge. Thus, bigram expansion is more expensive than unigram expansion. For a vocabulary size N, the bigram would need N^2 word-to-word transitions in comparison to N for the unigram. Each word transition has a transition probability equal to the corresponding bigram probability. Fortunately, the total number of states for bigram search is still proportional to the vocabulary size N.

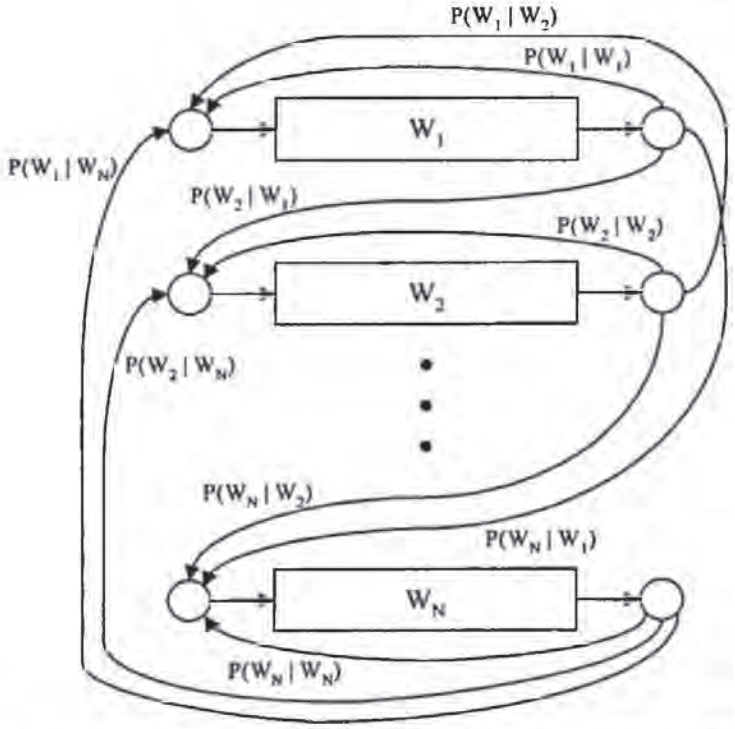


Figure 12.15 A bigram grammar network where the bigram probability $P(w_j \mid w_i)$ is attached as the transition probability from word w_i to w_j [19].

Because the search space for bigram is kept manageable, bigram search can be implemented very efficiently. Bigram search is a good compromise between efficient search and effective language models. Therefore, bigram search is arguably the most widely used search technique for unconstrained large-vocabulary continuous speech recognition. Particularly for the multiple-pass search techniques described in Chapter 13, a bigram search is often used in the first pass search.

12.3.3.1. Backoff Paths

When the vocabulary size N is large, the total bigram expansion N^2 can become computationally prohibitive. As described in Chapter 11, only a limited number of bigrams are observable in any practical corpora for a large vocabulary size. Suppose the probabilities for unseen bigrams are obtained through Katz's backoff mechanism. That is, for unseen bigram $P(w_j \mid w_i)$,

$$P(w_j \mid w_i) = \alpha(w_i)P(w_j) \tag{12.13}$$

where $\alpha(w_i)$ is the backoff weight for word w_i.

Using the backoff mechanism for unseen bigrams, the bigram expansion can be significantly reduced [12]. Figure 12.16 shows the new word expansion scheme. Instead of full bigram expansion, only observed bigrams are connected by direct word transitions with correspondent bigram probabilities. For backoff bigrams, the last state of word w_i is first connected to a central backoff node with transition probability equal to backoff weight $\alpha(w_i)$. The backoff node is then connected to the beginning of each word w_j with transition probability equal to its corresponding unigram probability $P(w_j)$. Readers should note that there are now two paths from w_i to w_j for an observed bigram $P(w_j \mid w_i)$. One is the direct link representing the observable bigram $P(w_j \mid w_i)$, and the other is the two-link backoff path representing $\alpha(w_i)P(w_j)$. For a word pair whose bigram exists, the two-link backoff path is likely to be ignored since the backoff unigram probability is almost always smaller than the observed bigram $P(w_j \mid w_i)$. Suppose there are only N_b different observable bigrams, this scheme requires $N_b + 2N$ instead of N^2 word transitions. Since under normal circumstance $N_b \ll N^2$, this backoff scheme significantly reduces the cost of word expansion.

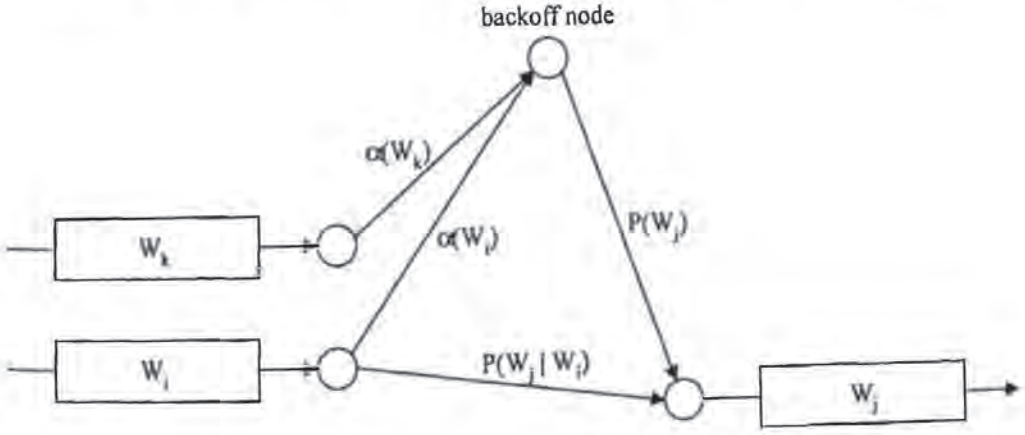


Figure 12.16 Reducing bigram expansion in a search by using the backoff node. In addition to normal bigram expansion arcs for all observed bigrams, the last state of word w_i is first connected to a central backoff node with transition probability equal to backoff weight $\alpha(w_i)$. The backoff node is then connected to the beginning of each word w_j with its corresponding unigram probability $P(w_j)$ [12].

12.3.4. Search Space with Trigrams

For a trigram language model, the language model score is:

$$P(\mathbf{W}) = P(w_1 \mid <s>)P(w_2 \mid <s>, w_1)\prod_{i=3}^{n} P(w_i \mid w_{i-2}, w_{i-1}) \tag{12.14}$$

The search space is considerably more complex, as shown in Figure 12.17. Since the equivalence grammar class is the previous two words w_i and w_j, the total number of grammar states is N^2. From each of these grammar states, there is a transition to the next word [19].

Obviously, it is very expensive to implement large-vocabulary trigram search given the complexity of the search space. It becomes necessary to dynamically generate the trigram search graph (trellis) via a graph search algorithm. The other alternative is to perform a multiple-pass search strategy, in which the first-pass search uses less detailed language models, like bigrams, to generate an n-best list or word lattice, and then a second-pass detailed search can use trigrams on a much smaller search space. Multiple-pass search strategy is discussed in Chapter 13.

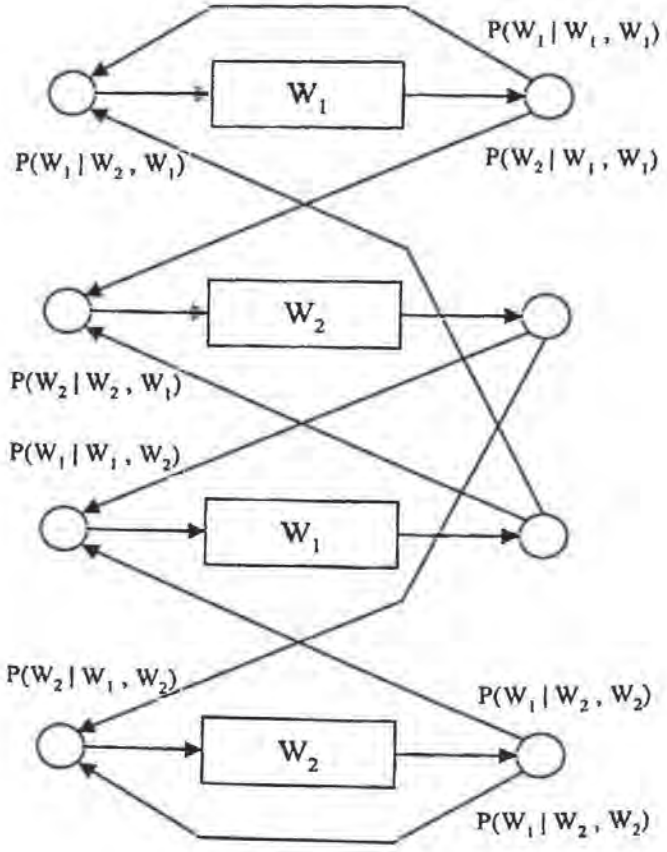


Figure 12.17 A trigram grammar network where the trigram probability $P(w_k \mid w_i, w_j)$ is attached to transition from grammar state word w_i, w_j to the next word w_k. Illustrated here is a two-word vocabulary, so there are four grammar states in the trigram network [19].

12.3.5. How to Handle Silences Between Words

In continuous speech recognition, there are unavoidable pauses (silences) between words or sentences. The pause is often referred to as silence or a non-speech event in continuous speech recognition. Acoustically, the pause is modeled by a silence model[6] that models background acoustic phenomena. The silence model is usually modeled with a topology flexible enough to accommodate a wide range of lengths, since the duration of a pause is arbitrary.

It can be argued that silences are actually linguistically distinguishable events, which contribute to prosodic and meaning representation. For example, people are likely to pause more often in phrasal boundaries. However, these patterns are so far not well understood for unconstrained natural speech (particularly for spontaneous speech). Therefore, the design of almost all automatic speech recognition systems today allows silences occurring just about anywhere between two lexical tokens or between sentences. It is relatively safe to assume that people pause a little bit between sentences to catch breath, so the silences between sentences are assumed mandatory while silences between words are optional. In most systems, silence is often modeled as a special lexicon entry with special language model probability. This special language model probability is also referred to as silence insertion penalty that is set to adjust the likelihood of inserting such an optional silence between words.

It is relatively straightforward to handle the optional silence between words. We need only to replace all the grammar states connecting words with a small network like the one shown in Figure 12.18. This arrangement is similar to that of the optional silence in training continuous speech, described in Chapter 9. The small network contains two parallel paths. One is the original null transition acting as the direct transition from one word to another, while the other path will need to go through a silence model with the silence insertion penalty attached in the transition probability before going to the next word.

One thing to clarify in the implementation of Figure 12.18 is that this silence expansion needs to be done for every grammar state connecting words. In the unigram grammar network of Figure 12.14, since there is only one collector node to connect words, the silence expansion is required only for this collector node. On the other hand, in the bigram grammar network of Figure 12.15, there is a collector node for every word before expanding to the next word. In this case, the silence expansion is required for every collector node. For a vocabulary size $|V|$, this means there are $|V|$ numbers of silence networks in the grammar search network. This requirement lies in the fact that in bigram search we cannot merge paths before expanding into the next word. Optional silence can then be regarded as part of the search effort for the previous word, so the word expansion needs to be done after finishing the optional silence. Therefore, we treat each word as having two possible pronunciations, one with the silence at the end and one without. This viewpoint integrates silence in the word pronunciation network like the example shown in Figure 12.19.

[6] Some researchers extend the context-dependent modeling to silence models. In that case, there are several silence models based on surrounding contexts.

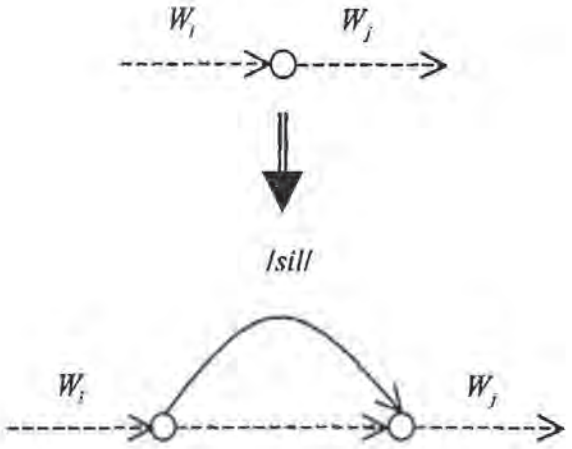


Figure 12.18 Incorporating optional silence (a non-speech event) in the grammar search network where the grammar state connecting different words is laced by two parallel paths. One is the original null transition directly from one word to the other, while the other first goes through the silence word to accommodate the optional silence.

For efficiency reasons, a single silence is sometimes used for large-vocabulary continuous speech recognition using higher order n-gram language model. Theoretically, this could be a source of pruning errors.[7] However, the error could turn out to be so small as to be negligible because there are, in general, very few pauses between word for continuous speech. On the other hand, the overhead of using multiple silences should be very minimal because it is less likely to visit those silence models at the end of words due to pruning.

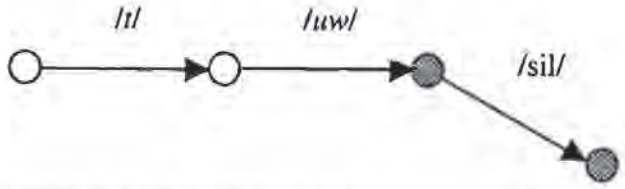


Figure 12.19 An example of treating silence as of the pronunciation network of word TWO. The shaded nodes represent possible word-ending nodes: one without silence and the other one with silence.

12.4. TIME-SYNCHRONOUS VITERBI BEAM SEARCH

When HMMs are used for acoustic models, the acoustic model score (likelihood) used in search is by definition the forward probability. That is, all possible state sequences must be considered. Thus,

[7] Speech recognition errors due to sub-optimal search or heuristic pruning are referred to as *pruning errors*, which will be described in detail in Chapter 13.

$$P(\mathbf{X} \mid \mathbf{W}) = \sum_{\text{all possible } s_0^T} P(\mathbf{X}, s_0^T \mid \mathbf{W}) \tag{12.15}$$

where the summation is to be taken over all possible state sequences S with the word sequence W under consideration. However, under the trellis framework (as in Figure 12.11), more bookkeeping must be performed since we cannot add scores with different word sequence history. Since the goal of decoding is to uncover the best word sequence, we could approximate the summation with the maximum to find the best state sequence instead. The Bayes' decision rule, Eq. (12.5), becomes

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{W}} P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W}) \cong \arg\max_{\mathbf{W}} \left\{ P(\mathbf{W}) \max_{s_0^T} P(\mathbf{X}, s_0^T \mid \mathbf{W}) \right\} \tag{12.16}$$

Equation (12.16) is often referred to as the *Viterbi approximation*. It can be literally translated to "the *most likely word sequence* is approximated by the *most likely state sequence*." Viterbi search is then sub-optimal. Although the search results by using forward probability and Viterbi probability could, in principle, be different, in practice this is rarely the case. We use this approximation for the rest of this chapter.

The Viterbi search has already been discussed as a solution to one of the three fundamental HMM problems in Chapter 8. It can be executed very efficiently via the same trellis framework. To briefly reiterate, the Viterbi search is a time-synchronous search algorithm that completely processes time t before going on to time $t+1$. For time t, each state is updated by the best score (instead of the sum of all incoming paths) from all states in at time $t-1$. This is why it is often called *time-synchronous Viterbi search*. When one update occurs, it also records the backtracking pointer to remember the most probable incoming state. At the end of search, the most probable state sequence can be recovered by tracing back these backtracking pointers. The Viterbi algorithm provides an optimal solution for handling nonlinear time warping between hidden Markov models and acoustic observation, word boundary detection and word identification in continuous speech recognition. This unified Viterbi search algorithm serves as the basis for all search algorithms as described in the rest of the chapter.

It is necessary to clarify the backtracking pointer for time-synchronous Viterbi search for continuous word recognition. We are generally not interested in the optimal state sequence for speech recognition.[8] Instead, we are only interested in the optimal word sequence indicated by Eq. (12.16). Therefore, we use the backtrack pointer just to remember the word history for the current path, so the optimal word sequence can be recovered at the end of search. To be more specific, when we reach the final state of a word, we create a history node containing the word identity and current time index and append this history node to the existing backtrack pointer. This backtrack pointer is then passed onto the successor node if it

[8] While we are not interested in optimal state sequences for ASR, they are very useful in deriving phonetic segmentation, which could provide important information for developing ASR systems.

is the optimal path leading to the successor node for both intra-word and inter-word transition. The side benefit of keeping this backtrack pointer is that we no longer need to keep the entire trellis during the search. Instead, we only need space to keep two successive time slices (columns) in the trellis computation (the previous time slice and the current time slice) because all the backtracking information is now kept in the backtrack pointer. This simplification is a significant benefit in the implementation of a time-synchronous Viterbi search.

Time-synchronous Viterbi search can be considered as a *breadth-first search* with dynamic programming. Instead of performing a tree search algorithm, the dynamic programming principle helps create a search graph where multiple paths leading to the same search state are merged by keeping the best path (with minimum cost). The Viterbi trellis is a representation of the search graph. Therefore, all the efficient techniques for graph search algorithms can be applied to time-synchronous Viterbi search. Although so far we have described the trellis in an explicit fashion—the whole search space needs to be explored before the optimal path can be found—it is not necessary to do so. When the search space contains an enormous number of states, it becomes impractical to pre-compile the composite HMM entirely and store it in the memory. It is preferable to dynamically build and allocate portions of the search space sufficient to search the promising paths. By using the graph search algorithm described in Section 12.1.1, only part of the entire Viterbi trellis is generated explicitly. By constructing the search space dynamically, the computation cost of the search is proportional only to the number of active hypotheses, independent of the overall size of the potential search space. Therefore, dynamically generated trellises are key to heuristic Viterbi search for efficient large-vocabulary continuous speech recognition, as described in Chapter 13.

12.4.1. The Use of Beam

Based on Chapter 8, the search space for Viterbi search is $O(NT)$ and the complexity is $O(N^2T)$, where N is the total number of HMM states and T is the length of the utterance. For large-vocabulary tasks these numbers are astronomically large even with the help of dynamic programming. In order to avoid examining the overwhelming number of possible cells in the HMM trellis, a heuristic search is clearly needed. Different heuristics generate or explore portions of the trellis in different ways.

A simple way to prune the search space for breadth-first search is the beam search described in Section 12.1.3.2. Instead of retaining all candidates (cells) at every time frame, a threshold T is used to keep only a subset of promising candidates. The state at time t with the lowest cost $D_{\min}$ is first identified. Then each state at time t with cost $> D_{\min} + T$ is discarded from further consideration before moving on to the next time frame $t+1$. The use of the beam alleviates the need to process all the cells. In practice, it can lead to substantial savings in computation with little or no loss of accuracy.

Although beam search is a simple idea, the combination of time-synchronous Viterbi and beam search algorithms produces the most powerful search strategy for large-vocabulary speech recognition. Comparing paths with equal length under a time-

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:

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

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

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:

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

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

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

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)

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

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

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].

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.

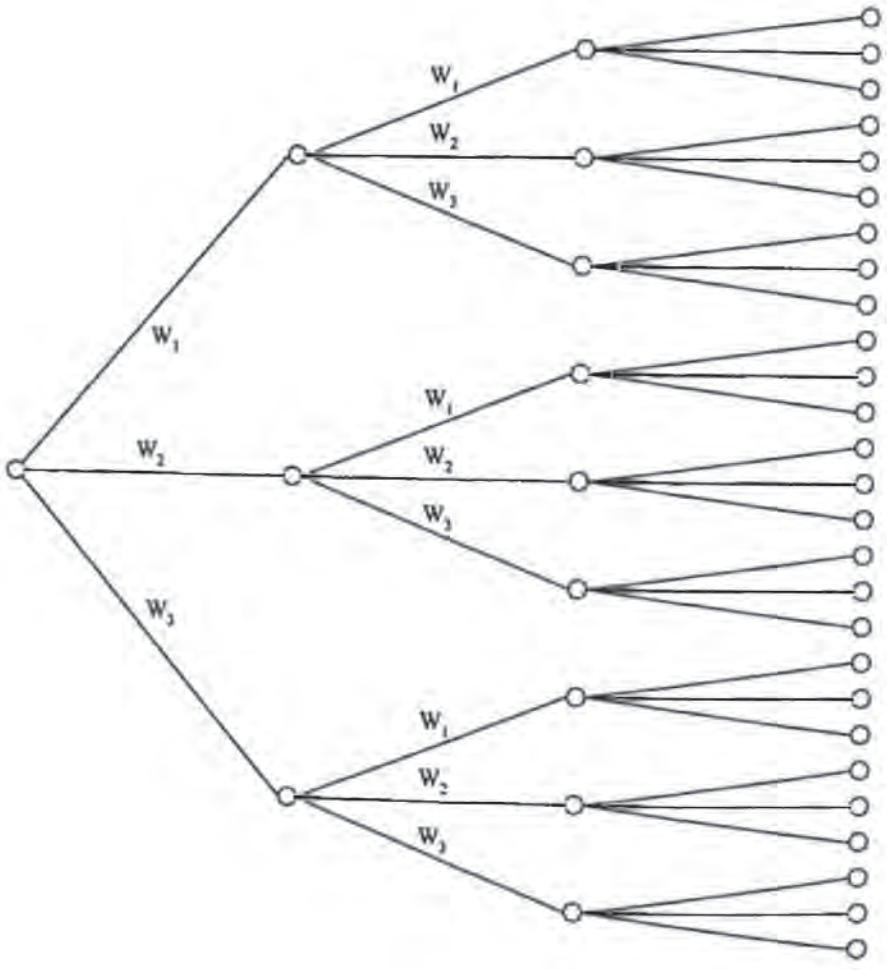


Figure 12.20 A stack decoding search tree for a vocabulary size of three [19].

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.

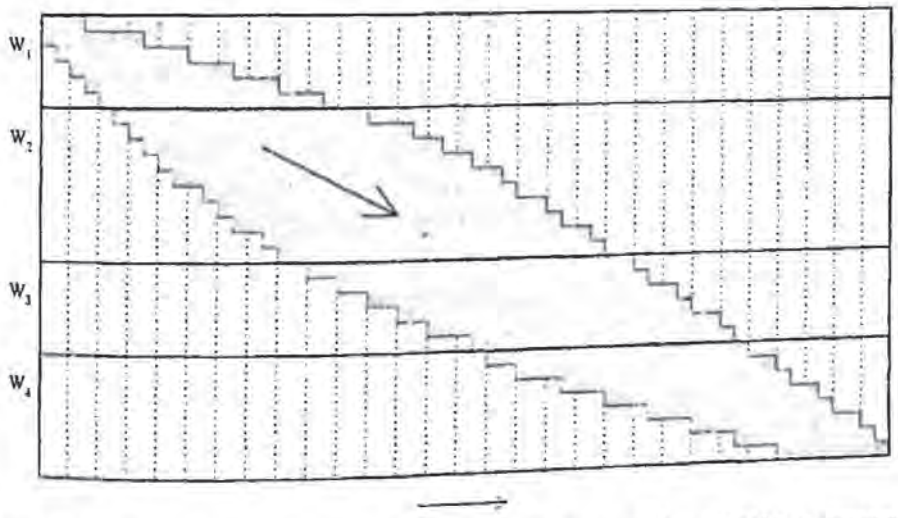


Figure 12.21 The forward trellis space for stack decoding. Each grid point corresponds to a trellis cell in the forward computation. The shaded area represents the values contributing to the computation of the forward score for the optimal word sequence $w_1, w_2, w_3, \ldots$ [24].

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.

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

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

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.

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

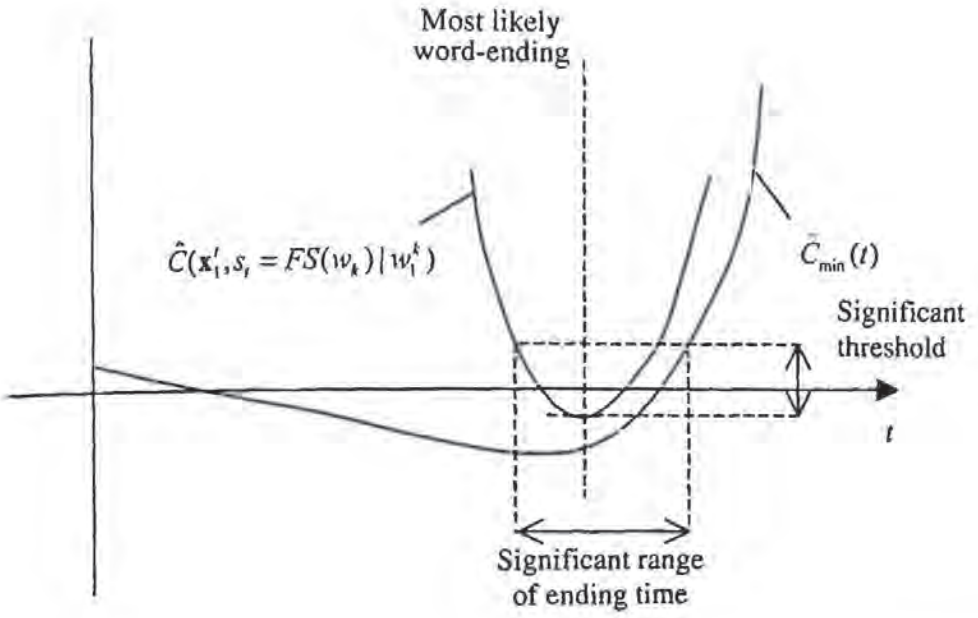


Figure 12.22 $\hat{C}_{\min}(t)$ and $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$ as functions of time t. The valley region represents possible ending times for the correct phone/word.

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.

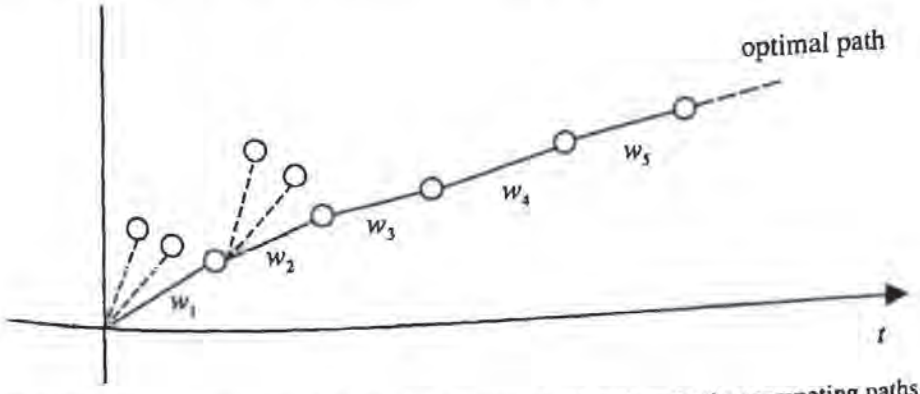


Figure 12.23 Unnormalized cost $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ for optimal path and other competing paths as a function of time.

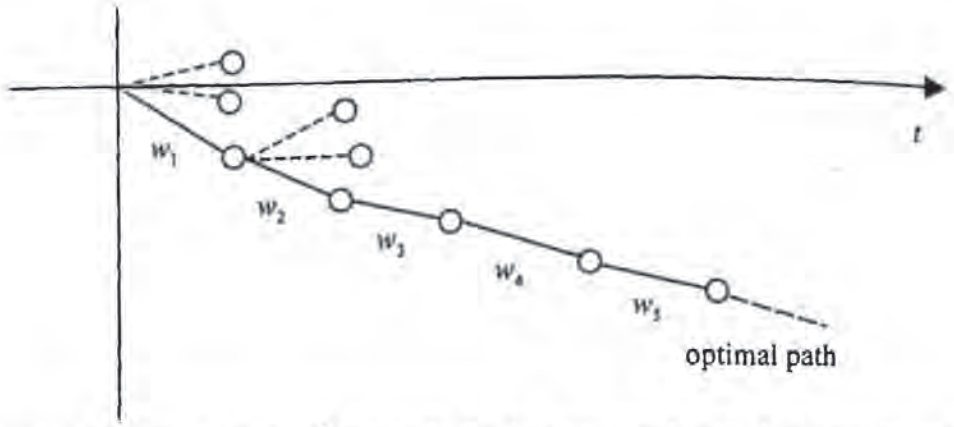


Figure 12.24 Normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^t)$ for the optimal path and other competing paths as a function of time.

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]:

$$\gamma_t = \gamma(\mathbf{x}_t) \qquad \text{first-order estimate}$$
$$\gamma_t = \gamma(\mathbf{x}_t, \mathbf{x}_{t-1}) \qquad \text{second-order estimate}$$
$$\gamma_t = \gamma(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-N+1}) \qquad n\text{-order estimate}$$

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_t 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_t.

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

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

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

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).

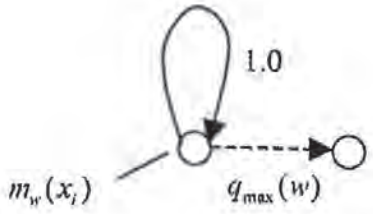


Figure 12.25 The equivalent one-state HMM corresponding to fast match computation defined in Eq. (12.40) [15].

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

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-

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

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.

[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.

[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.

[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.

CHAPTER 13

Large-Vocabulary Search Algorithms

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

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

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

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

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

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

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

13.1. Efficient Manipulation of 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.

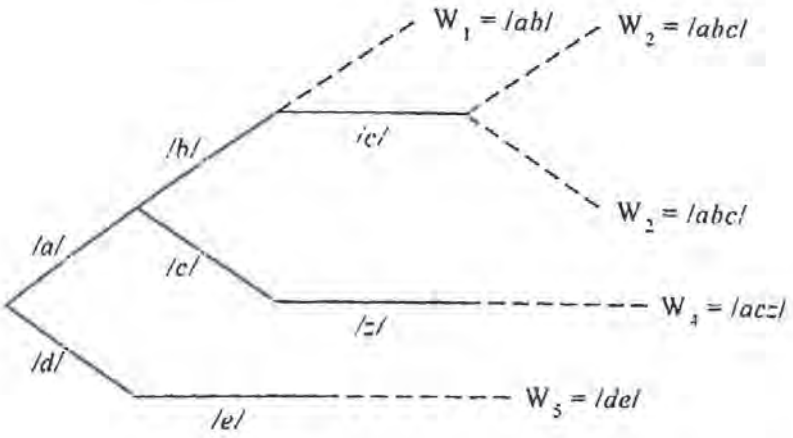


Figure 13.1 An example of a lexical tree, where each branch corresponds to a shared phoneme and the leaf corresponds to a word.

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

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

case. In general, when more detailed (better) acoustic and/or language models are used, the effect of a potentially increased search space is often compensated by a more focused beam search from the use of more accurate models. In other words, although the static search space might increase significantly by using more accurate models, the dynamic search space can be under control (sometimes even smaller), thanks to improved evaluation functions.

To deal with tree copies [19, 23, 37], you can create redundant subtrees. When copies of lexical trees are used to disambiguate active linguistic contexts, many of the active state hypotheses correspond to the same redundant unigram state, due to the postponed application of language models. To apply the language model sooner, and to eliminate redundant unigram state computations, a successor tree, T_i, can be created for each linguistic context i. T_i encodes the nonzero n-grams of the linguistic context i as an isomorphic subgraph of the unigram tree, T_0. Figure 13.2 shows the organization of such successor trees and unigram tree for bigram search. For each word w a successor tree, T_w is created with the set of successor words that have nonzero bigram probabilities. Suppose u is a successor of w; the bigram probability $P(u \mid w)$ is attached to the transition connecting the leaf corresponding to u in the successor tree T_w, with the root of the successor tree T_u. The unigram tree is a full-size lexical tree and is shared by all words as the back-off lexical tree. Each leaf of the unigram tree corresponds to one of $|V|$ words in the vocabulary and is linked to the root of its bigram successor tree (T_u) by an arc with the corresponding unigram probability $P(u)$. The backoff weight, $\alpha(u)$, of predecessor u is attached to the arc which links the root of successor tree T_u to the root of the unigram tree.

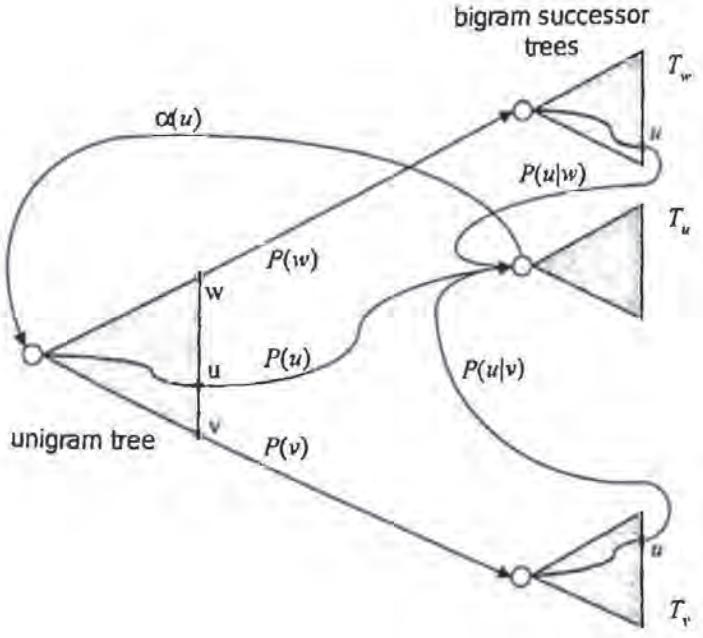


Figure 13.2 Successor trees and unigram trees for bigram search [13].

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

13.1.3. Factored Language Probabilities

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

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

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

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

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

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

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

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

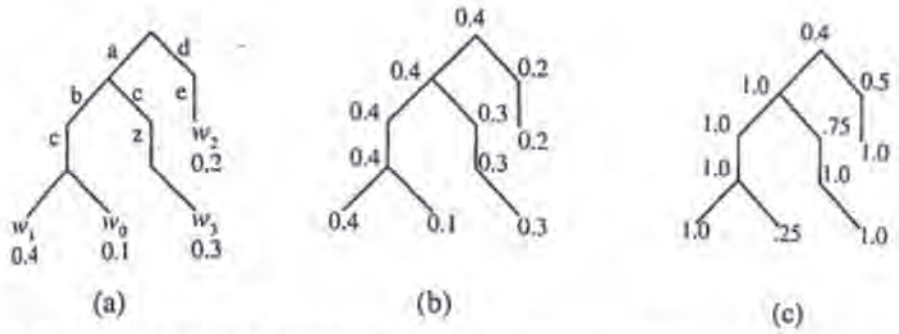


Figure 13.3 (a) Unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree $F^*(n)$ [4].

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.

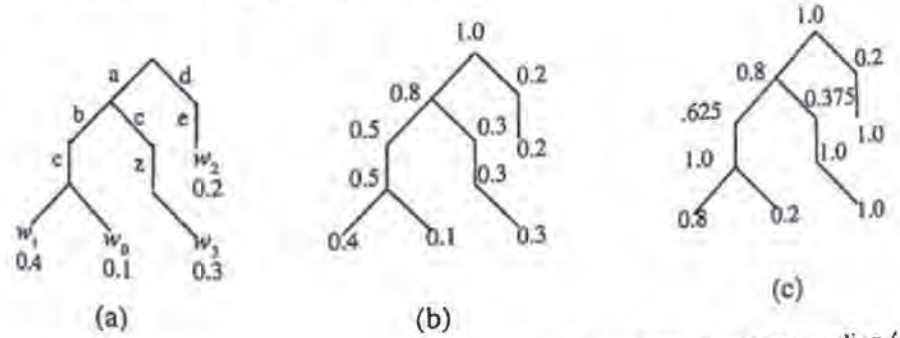


Figure 13.4 Using sum instead of upper bound when factoring tree, the corresponding (a) unfactored lexical tree; (b) distributed probabilities with computed $P^*(n)$; (c) factored tree with computed $F^*(n)$ [4].

One interesting observation is that the language model score can be regarded as a heuristic function to estimate the linguistic expectation of the current word to be searched. In a linear representation of the pronunciation lexicon, application of the linguistic expectation was straightforward, since each state is associated with a unique word. Therefore, given the context defined by the hypothesis under consideration, the expectation for the first phone of word w_i is just $P(w_i \mid w_1^{i-1})$. After the first phone, the expectation for the rest of the phones becomes 1.0, since there is only one possible phone sequence when searching the word w_i. However, for the tree lexicon, it is necessary to compute $E(p_j \mid p_1^{j-1}, w_1^{i-1})$, the expectation of phone p_j given the phonetic prefix p_1^{j-1} and the linguistic context w_1^{i-1}. Let $\phi(j, w_k)$ denote the phonetic prefix of length j for w_k. Based on Eqs. (13.1) and (13.2), we can compute the expectation as:

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

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

13.1.3.1. Efficient Memory Organization of Factored Lexical Trees

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

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.

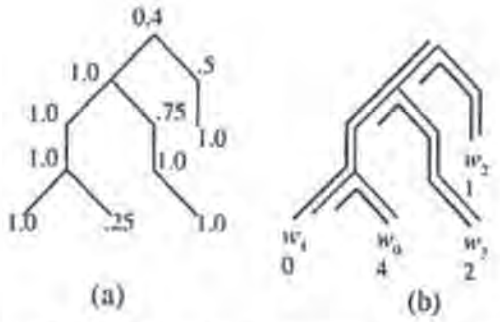


Figure 13.5 (a) Factored tree; (b) tree with common prefix-length annotation.

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.

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].

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.

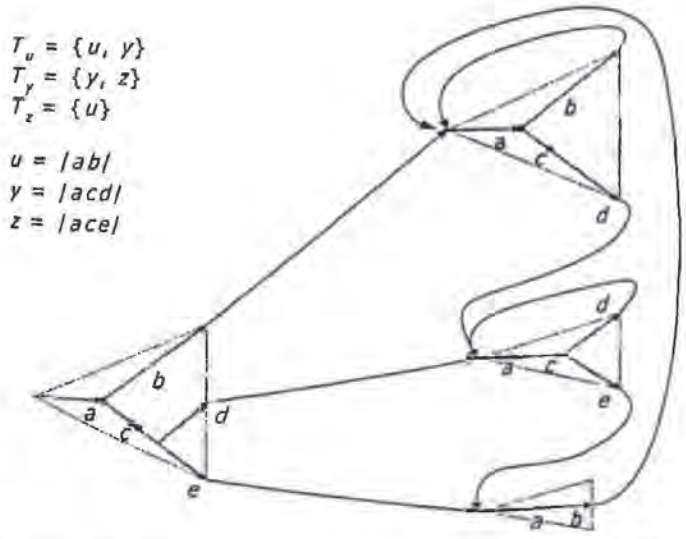


Figure 13.6 An example of a lexical tree network without shared-tail optimization [12]. The vocabulary includes three words, u, y, and z. T_u, T_y, and T_z are the successor trees for u, y, and z respectively [13].

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

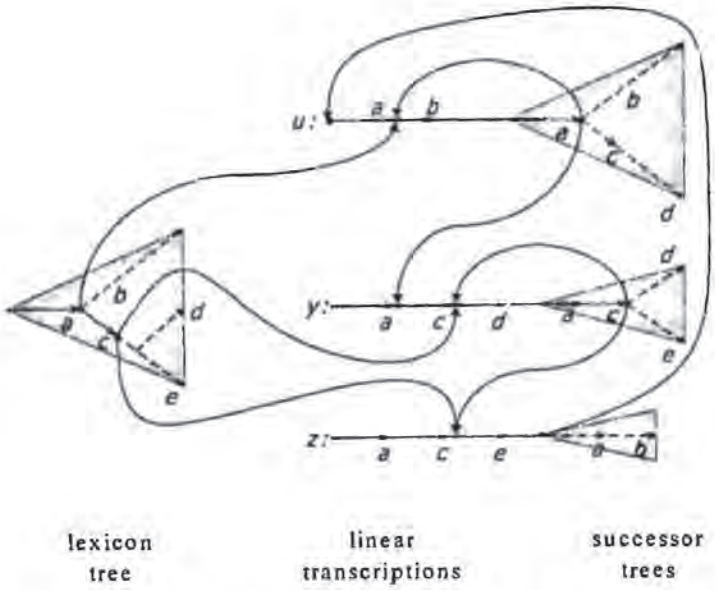


Figure 13.7 The lexical tree network in Figure 13.6 after shared-tail optimization [12].

13.1.5. Exploiting Subtree Polymorphism

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

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

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

In place of computing strict subtree dominance, a *polymorphic* linguistic context assignment to reduce redundancy is employed by estimating subtree dominance based on local information and ignoring the subgraph isomorphism problem. Polymorphic context assignment involves keeping a single copy of the lexical tree and allowing each state to assume the linguistic context of the most promising history. The advantage of this approach is that it employs maximum sharing of data structures and information, so each node in the tree is

evaluated, at most, once. However, the use of local knowledge to determine the dominant context could introduce significant errors because of premature pruning. Whisper [4] reports a 65.7% increase in error rate when only the dominant context is kept, based on local knowledge.

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. $d = Cost(s_n, c) \div (-\log P(s_m \mid s_n, c))$
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
   $Delete(s_m, w)$
   $Add(s_m, c)$; $StateInfo(s_m, c) = StateInfo(s_n, c)$
   $Cost(s_m, c) = d$

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.

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

degree of this phonetic HMM, i.e., the fate of all the states within this phonetic HMM. Although this does not actually reduce the beam beyond normal pruning, it has the effect of processing fewer candidates in the beam. In programming, this means less checking and bookkeeping, so some computation savings can be expected.

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

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

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

13.2.2. Different Layers of Beams

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

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

Pruning can introduce search errors if a state is pruned that would have been on the globally best path. The principle applied here is that the more constraints you have available, the more aggressively you decide whether this path will participate in the globally best path. In this case, at the state level, you have the least constraints. At the phonetic level there are more, and there are 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

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.

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*.

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..

- *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.

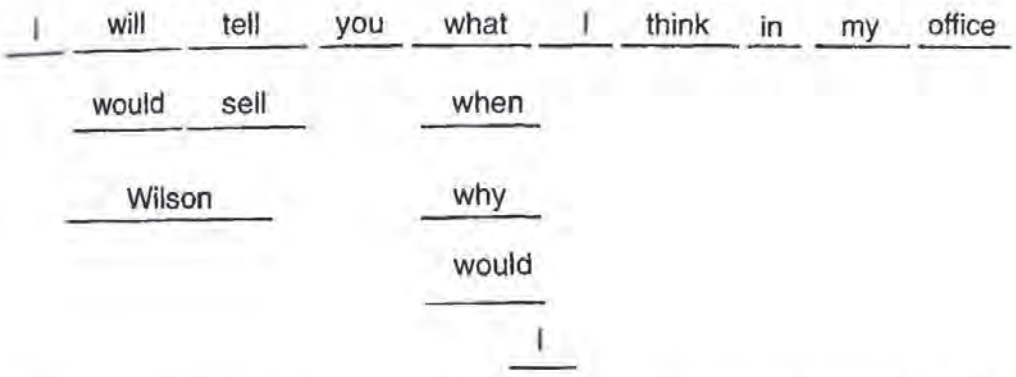


Figure 13.8 A word lattice example. Each word has an explicit time interval associated with it.

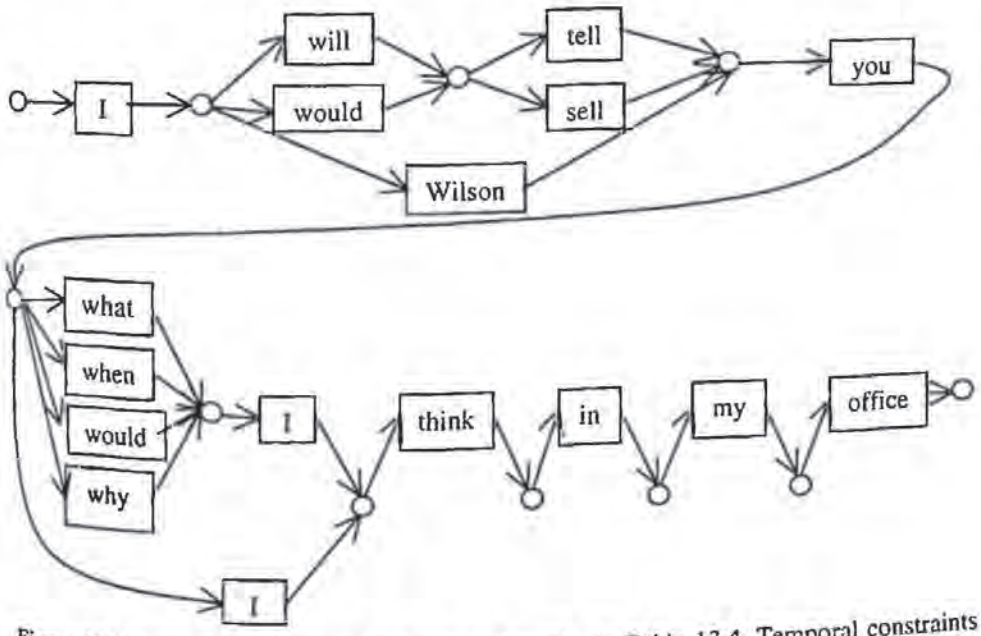


Figure 13.9 A word graph example for the n-best list in Table 13.4. Temporal constraints are implicit in the topology.

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.

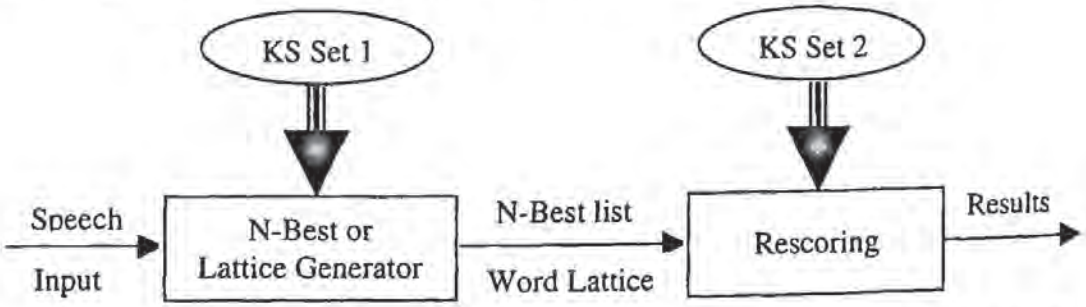


Figure 13.10 N-best/lattice search framework. The most discriminant and inexpensive knowledge sources (KSs 1) are used first to generate the n-best/lattice. The remaining knowledge sources (KSs 2, usually expensive to apply) are used in the rescoring phase to pick up the optimal solution [40].

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

with distinct histories. The history is defined as the whole word sequence up to the current time t and word w. This exact n-best algorithm is also called *sentence-dependent* n-best algorithm. When two or more paths come to the same state at the same time, paths having the same history are merged and their probabilities are summed together; otherwise, only the n-best paths are retained for each state. As commonly used in speech recognition, a typical HMM state has 2 or 3 predecessor states within the word HMM. Thus, for each time frame and each state, the n-best search algorithm needs to compare and merge 2 or 3 sets of n paths into n new paths. At the end of the search, the n paths in the final state of the trellis are simply re-ordered to obtain the n-best word sequences.

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

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

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

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

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

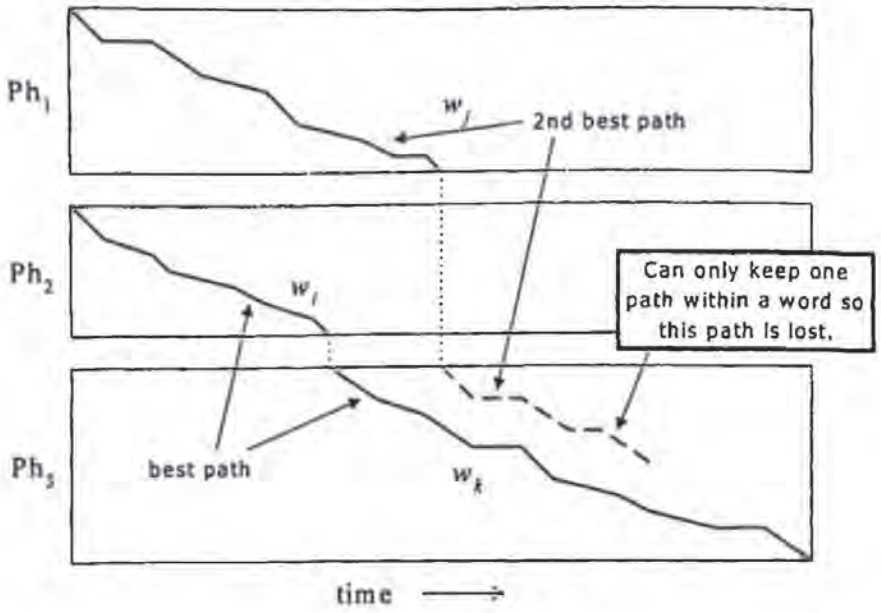


Figure 13.11 Deficiency in traceback-based n-best algorithm. The best subpath, w_i - w_k, will prune away subpath w_j - w_k while searching the word w_k; the second-best subpath cannot be recovered [40].

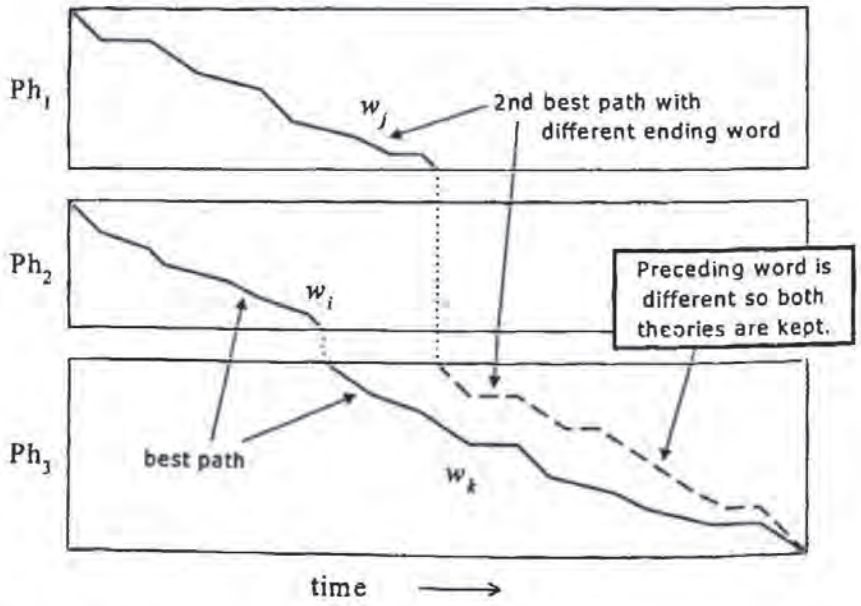


Figure 13.12 Word-dependent n-best algorithm. Both subpaths w_i - w_k and w_j - w_k are kept under the word-dependent assumption [40].

The *word-dependent* n-best algorithm [38] can alleviate the deficiency of the trace-back-based n-best algorithm, in which only one starting time is kept for each word, so the starting time is independent of the preceding words. On the other hand, in the sentence-dependent n-best algorithm, the starting time for a word 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.

$h(w_j;\tau(t;w_i,w_j),t) = -\log P(\mathbf{x}'_\tau \mid w_j)$—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.

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.

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:

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

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 $(N+1)^{th}$ 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.

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

understanding technologies in one-pass search. On the other hand, n-best lists or word lattices provide a trivial interface between speech recognition and natural language understanding modules. Such an interface also provides a convenient mechanism for integrating different KSs in a modular way. This is important because the KSs could come from different modalities (like video or pen) that make one-pass integration almost infeasible. This high degree of modality allows different component subsystems to be optimized and implemented independently.

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

13.4. SEARCH-ALGORITHM EVALUATION

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

- Viterbi score instead of forward score described in Chapter 12.
- Beam pruning or stack pruning described in Section 13.2.2 and Chapter 12.

- Subtree dominance pruning described in Section 13.1.5.
- Fast match pruning described in Section 13.2.3.
- Rich-get-richer pruning described in Section 13.2.3.2.
- Multipass search strategies described in Section 13.3.5.

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

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

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

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

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

During the development of a speech recognizer, it is a good idea to always include the correct path in the search space. By including such a path, and some bookkeeping, one can use the correct path to help in determining all the pruning thresholds. If the correct path is pruned away during search by some threshold, some adjustment can be made to relax such a

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

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.

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.

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.

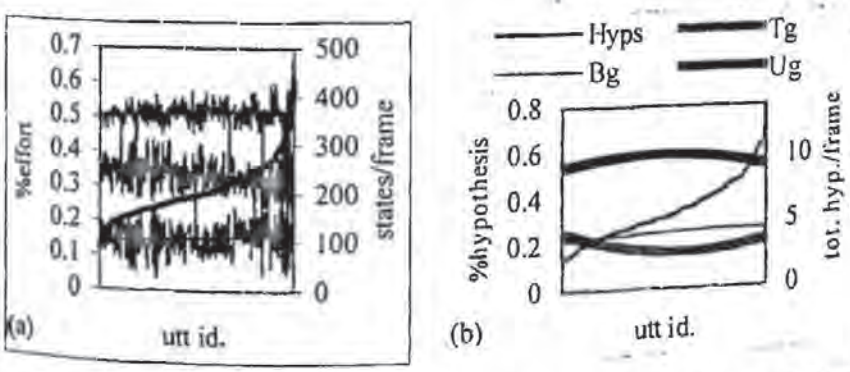


Figure 13.13 (a) Search effort for different linguistic contexts measured by number of active states in each of the three different linguistic contexts. The top series is for the bigram, then the unigram and trigram. The remaining series is the effort per utterance and is plotted on the secondary y-axis.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

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

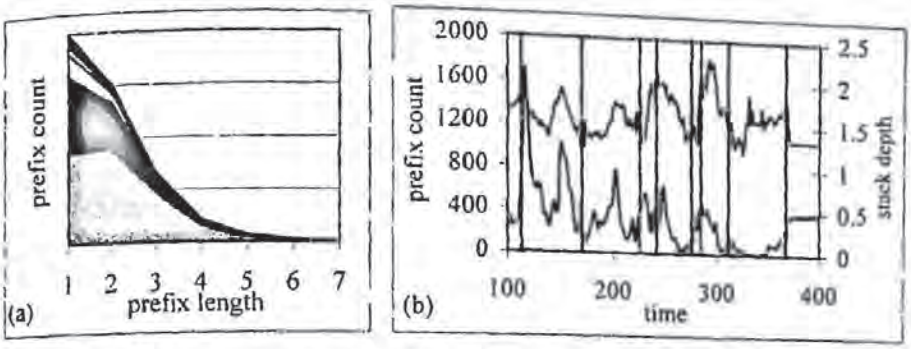


Figure 13.14 (a) A cumulative graph of the prefix count for each stack depth, starting with depth 1, showing the distribution according to prefix length. (b) The prefix count and the average stack depth with respect to one utterance. The vertical bars show the word boundaries [3].

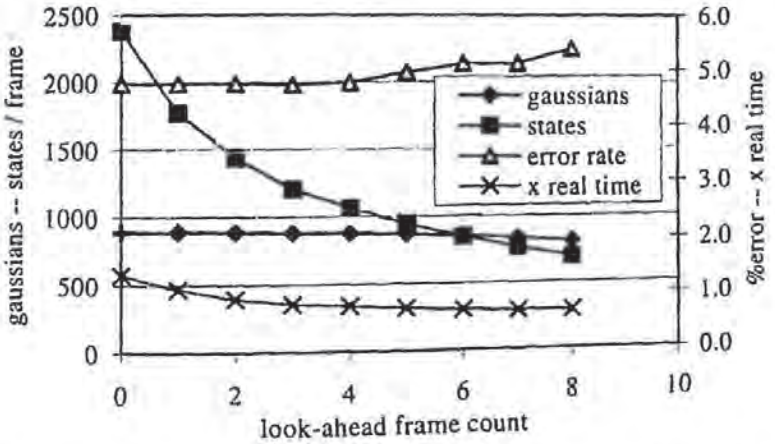


Figure 13.15 Search effort, percent error rate, and real-time factor as a function of the acoustic look-ahead. Note that search effort is the number of Gaussians evaluated per frame and the number of states evaluated per frame [3].

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-

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.

[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.

[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.

[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.

PART IV

TEXT-TO-SPEECH SYSTEMS

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.

- *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].

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.

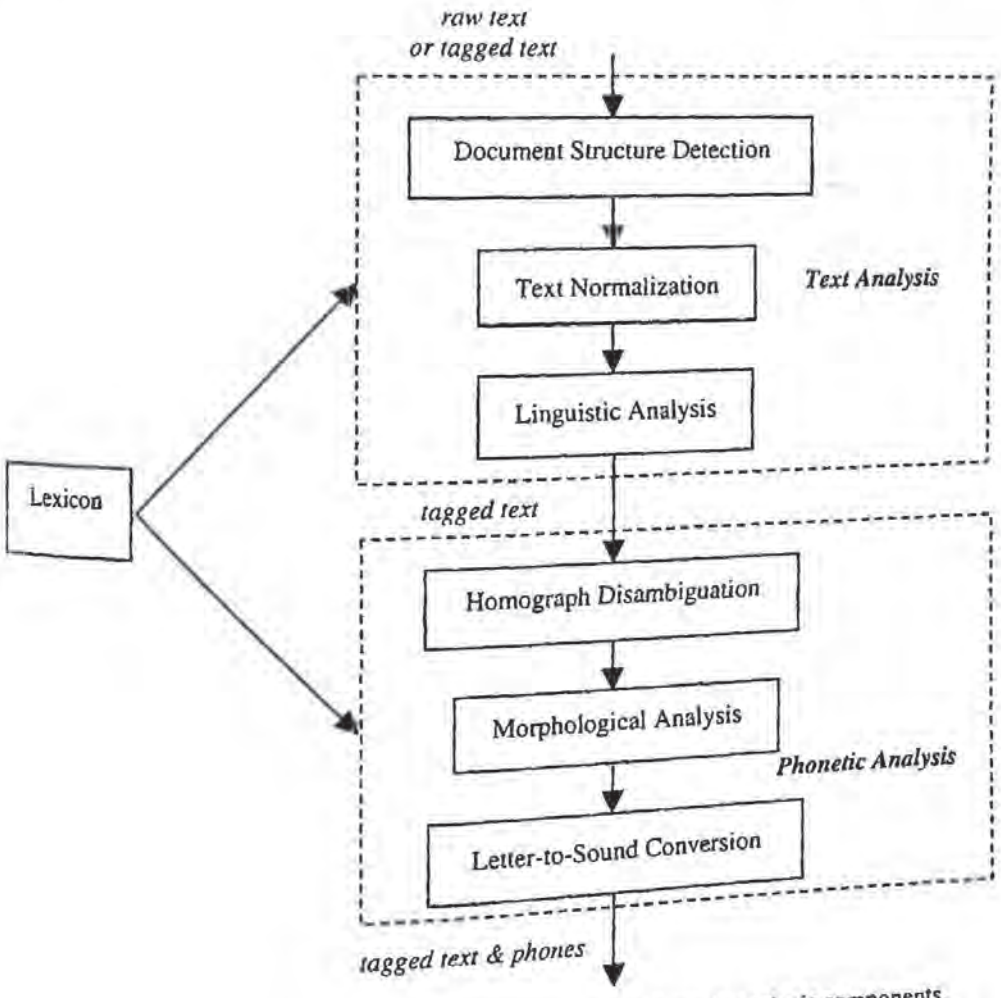


Figure 14.1 Modularized functional blocks for text and phonetic analysis components.

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-

netic language because English words often have many distinct origins. It is generally believed that the following three services are necessary to produce accurate pronunciations.

- *Homograph disambiguation*—It is important to disambiguate words with different senses to determine proper phonetic pronunciations, such as object (*/ah b jh eh k t/*) as a verb or as a noun (*/aa b jh eh k t/*).
- *Morphological analysis*—Analyzing the component morphemes provides important cues to attain the pronunciations for inflectional and derivational words.
- *Letter-to-sound conversion*—The last stage of the phonetic analysis generally includes general letter-to-sound rules (or modules) and a dictionary lookup to produce accurate pronunciations for any arbitrary word.

All the processes in text and phonetic analysis phases above need not to be deterministic, although most TTS systems today have deterministic processes. What we mean by *not deterministic* is that each of the above processes can generate multiple hypotheses with the hope that the later process can disambiguate those hypotheses by using more knowledge. For example, often it might not be trivial to decide whether the punctuation "." is a sentence ending mark or abbreviation mark during document structure detection. The document structure detection process can pass both hypotheses to the later processes, and the decision can then be delayed until there is enough information to make an informed decision in later modules, such as the text normalization or linguistic analysis phases. When generating multiple hypotheses, the process can also assign probabilistic information if it comprehends the underlying probabilistic structure. This flexible pipeline architecture avoids the mistakes made by early processes based on insufficient knowledge.

Much of the work done by the text/phonetic analysis phase of a TTS system mirrors the processing attempted by *natural language process* (NLP) systems for other purposes, such as automatic proofreading, machine translation, database document indexing, and so on. Increasingly sophisticated NL analysis is needed to make certain TTS processing decisions in the examples illustrated in Table 14.1. Ultimately all decisions are context driven and probabilistic in nature, since, for example, dogs might be cooked and eaten in some cultures.

Table 14.1 Examples of several ambiguous text normalization cases.

Examples	Alternatives	Techniques
Dr. Smith	*doctor* or *drive*?	abbreviation analysis, case analysis
Will you go?	yes-no or wh-question?	syntactic analysis
I ate a hot dog.	accent on *dog*?	semantic, verb/direct object likelihood
I saw a hot dog.	accent on *dog*?	discourse, pragmatic analysis

Most TTS systems today employ specialized natural language processing modules for front-end analysis. In the future, it is likely that less emphasis will be placed on construction of TTS-specific text/phonetic analysis components such as those described in [27], while more resources will likely go to general-purpose NLP systems with cross-functional potential [23]. In other words, all the modules above only perform simple processing and pass all possible hypotheses to the later modules. At the end of the text/phonetic phase, a unified NLP module then performs extensive syntactic/semantic analysis for the best decisions. The necessity for such an architectural approach is already visible in markets where language issues have forced early attention to common lexical and tokenization resources, such as Japan. Japanese system services and applications can usually expect to rely on common cross-functional linguistic resources, and many benefits are reaped, including elimination of bulk, reduction of redundancy and development time, and enforcement of systemwide consistent behavior. For example, under Japanese architectures, TTS, recognition, sorting, word processing, database, and other systems are expected to share a common language and dictionary service.

14.1.2. Data Flows

It is arguable that text input alone does not give the system enough information to express and render the intention of the text producer. Thus, more and more TTS systems focus on providing an infrastructure of standard set of markups (tags), so that the text producer can better express their semantic intention with these markups in addition to plain text. These kinds of markups have different levels of granularity, ranging from simple speed settings specified in *words per minute* up to elaborate schemes for semantic representation of concepts that may bypass the ordinary text analysis module altogether.[2] The markup can be done by internal proprietary conventions or by some standard markup, such as XML (Extensible Markup Language [35]). Some of these markup capabilities will be discussed in Sections 14.3 and 14.4.

For example, an application may know a lot about the structure and content of the text to be spoken, and it can apply this knowledge to the text, using common markup conventions, to greatly improve spoken output quality. On the other hand, some applications may have certain broad requirements such as rate, pitch, callback types, etc. For engines providing such supports, the text and/or phonetic analysis phase can be skipped, in whole or in part. Whether the application or the system has provided the text analysis markup, the structural conventions should be identical and must be sufficient to guide the phonetic analysis. The phonetic analysis module should be presented only with markup tags indicating structure or functions of textual chunks, and words in standard orthography. The similar phonetic markups could also be presented to the phonetic analysis module, the module could be skipped.

[2] This latter type of system is sometimes called *concept-to-speech* or *message-to-speech*, which is described in Chapter 17. It generally generates better speech rendering when domain-specific knowledge is provided to the system.

Internal architectures, data structures, and interfaces may vary widely from system to system. However, most modern TTS systems initially construct a simple description of an utterance or paragraph based on observable attributes, typically text words and punctuation, perhaps augmented by control annotations. This minimal initial *skeleton* is then augmented with many layers of structure hypothesized by the TTS system's internal analysis modules. Beginning with a surface stream of words, punctuation, and other symbols, typical layers of detected structure that may be added include:

- Phonemes
- Syllables
- Morphemes
- Words derived from nonwords (such as dates like "9/10/99")
- Syntactic constituents
- Relative importance of words and phrases
- Prosodic phrasing
- Accentuation
- Duration controls
- Pitch controls

We can now consider how the information needed to support synthesis of a sentence is developed in processing an example sentence such as: "A skilled electrician reported."

In Figure 14.2, the information that must be inferred from text is diagrammed. The flow proceeds as follows:

- **W(ords) → Σ, C(ontrols)**: the syllabic structure (Σ) and the basic phonemic form of a word are derived from lexical lookup and/or the application of rules. The Σ tier shows the syllable divisions (written in text form for convenience). The C tier, at this stage, shows the basic phonemic symbols for each word's syllables.
- **W(ords) → S(yntax/semantics)**: The word stream from text is used to infer a syntactic and possibly semantic structure (S tier) for an input sentence. Syntactic and semantic structure above the word would include syntactic constituents such as Noun Phrase (NP), Verb Phrase (VP), etc. and any semantic features that can be recovered from the current sentence or analysis of other contexts that may be available (such as an entire paragraph or document). The lower-level phrases such as NP and VP may be grouped into broader constituents such as Sentence (S), depending on the parsing architecture.
- **S(yntax/semantics) → P(rosody)**: The P(rosodic) tier is also called the symbolic prosodic module. If a word is semantically *important* in a sentence, that importance can be reflected in speech with a little extra phonetic prominence, called an accent. Some synthesizers begin building a prosodic structure by

placing metrical foot boundaries to the left of every accented syllable. The resulting metrical foot structure is shown as F1, F2, etc. in Figure 14.2 (some *feet* lack an accented head and are 'degenerate'). Over the metrical foot structure, higher-order prosodic constituents, with their own characteristic relative pitch ranges, boundary pitch movements, etc. can be constructed, shown in the figure as intonational phrases IP1, IP2. The details of prosodic analysis, including the meaning of those symbols, are described in Chapter 15.

The final phonetic form of the words to be spoken will reflect not only the original phonetics, but decisions made in the S and P tiers as well. For example, the P(rosody) tier adds detailed pitch and duration controls to the C(ontrol) specification that is passed to the voice synthesis component. Obviously, there can be a huge variety of particular architectures and components involved in the conversion process. Most systems, however, have some analog to each of the components presented above.

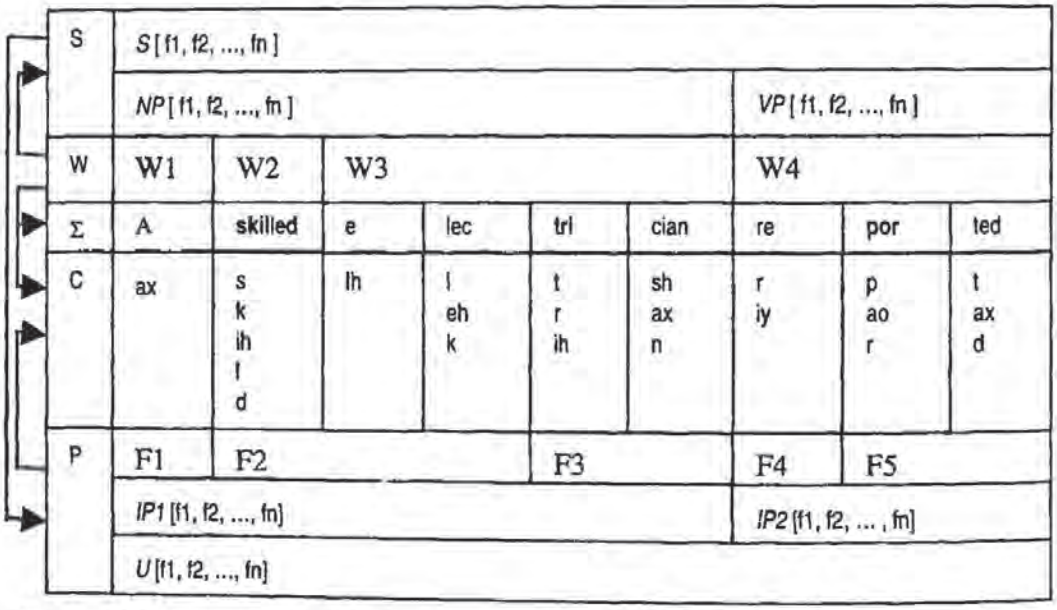


Figure 14.2 Annotation tiers indicating incremental analysis based on an input (text) sentence "A skilled electrician reported." Flow of incremental annotation is indicated by arrows on the left side.

14.1.3. Localization Issues

A major issue in the text and phonetic analysis components of a TTS system is internationalization and localization. While most of the language processing technologies in this book are exemplified by English case studies, an internationalized TTS architecture enabling minimal expense in localization is highly desirable. From a technological point of view, the text conventions and writing systems of language communities may differ substantially in arbitrary ways, necessitating serious effort in both specifying an internationalized architec-

ture for text and phonetic analysis, and localizing that architecture for any particular language.

For example, in Japanese and Chinese, the unit of *word* is not clearly identified by spaces in text. In French, interword dependencies in pronunciation realization exist (liaison). Conventions for writing numerical forms of dates, times, money, etc. may differ across languages. In French, number groups separated by spaces may need to be integrated as single amounts, which rarely occurs in English. Some of these issues may be more serious for certain types of TTS architectures than others. In general, it is best to specify a rule architecture for text processing and phonetic analysis based on some fundamental formalism that allows for language-particular data tables, and which is powerful enough to handle a wide range of relations and alternatives.

14.2. LEXICON

The most important resource for text and phonetic analysis is the TTS system lexicon (also referred to as a dictionary). As illustrated in Figure 14.1, the TTS system lexicon is shared with almost all components. The lexical service should provide the following kinds of content in order to support a TTS system:

- Inflected forms of lexicon entries
- Phonetic pronunciations (support multiple pronunciations), stress and syllabic structure features for each lexicon entry
- Morphological analysis capability
- Abbreviation and acronym expansion and pronunciation
- Attributes indicating word status, including proper-name tagging, and other special properties
- List of speakable names of all common single characters. Under modern operating systems, the characters should include all Unicode characters.
- Word part-of-speech (POS) and other syntactic/semantic attributes
- Other special features, e.g., how likely a word is to be accented, etc.

It should be clear that the requirements for a TTS system lexical service overlap heavily with those for more general-purpose NLP.

Traditionally, TTS systems have been rule oriented, in particular for grapheme-to-phoneme conversion. Often, tens of so called *letter-to-sound* (LTS) rules (described in detail in Section 14.8) are used first for grapheme-to-phoneme conversion, and the role of the lexicon has been minimized as an *exception list*, whose pronunciations cannot be predicted on the basis of such LTS rules. However, this view of the lexicon's role has increasingly been adjusted as the requirement of a sophisticated NLP analysis for high-quality TTS systems has become apparent. There are a number of ways to optimize a dictionary system. For a good overview of lexical organization issues, please see [4].

To expose different contents about a lexicon entry listed above for different TTS module, it calls for a consistent mechanism. It can be done either through a database query or a function call in which the caller sends a key (usually the orthographic representation of a word) and the desired attribute. For example, a TTS module can use the following function call to look up a particular attribute (like phonetic pronunciations or POS) by passing the attribute *att* and the result will be stored in the pointer *val* upon successful lookup. Moreover, when the lookup is successful (the word is found in the dictionary) the function returns true, otherwise it will return false instead.

```
BOOLEAN DictLookup (string word, ATTTYPE att, (VOID *) val)
```

We should also point out that this functional view of dictionary could further expand the physical dictionary as a service. The morphological analysis and letter-to-sound modules (described in Sections 14.7 and 14.8) can all be incorporated into the same lexical service. That is, underneath dictionary lookup, operation and analysis is encapsulated from users to form a uniform service.

Another consideration in the system's runtime dictionary is compression. While many standard compression algorithms exist, and should be judiciously applied, the organization and extent of the vocabulary itself can also be optimized for small space and quick search. The kinds of American English vocabulary relevant to a TTS system include:

- Grammatical function words (closed class)—about several hundred
- Very common vocabulary—about 5,000 or more
- College-level core vocabulary base forms—about 60,000 or more
- College-level core vocabulary inflected form—about 120,000 or more
- Scientific and technical vocabulary, by field—e.g., legal, medical, engineering, etc.
- Personal names—e.g., family, given, male, female, national origin, etc.
- Place names—e.g., countries, cities, rivers, mountains, planets, stars, etc.
- Slang
- Archaisms

The typical sizes of reasonably complete lists of the above types of vocabulary run from a few hundred function or closed-class words (such as prepositions and pronouns) to 120,000 or so inflected forms of college-level vocabulary items, up to several million surnames and place names. Careful analysis of the likely needs of typical target applications can potentially reduce the size of the runtime dictionary. In general, most TTS systems maintain a system dictionary with a size between 5000 and 200,000 entries. With advanced technologies in database and hashing, search is typically a nonissue for dictionary lookup. In addition, since new forms are constantly produced by various creative processes, such as acronyms, borrowing, slang acceptance, compounding, and morphological manipulation, some means of analyzing words that have not been stored must be provided. This is the topic of Sections 14.7 and 14.8.

14.3. DOCUMENT STRUCTURE DETECTION

For the purpose of discussion, we assume that all input to the TAM is an XML document, though perhaps largely unmarked, and the output is also a (more extensively marked) XML document. That is to say, all the knowledge recovered during the TAM phase is to be expressed as XML markup. This confirms the independence of the TAM from phonetic and prosodic considerations, allowing a variety of resources, some perhaps not crafted with TTS in mind, to be brought to bear by the TAM on the text. It also implies that that output of the TAM is potentially usable by other, non-TTS processes, such as normalization of language-model training data for building statistical language models (see Chapter 11). This fully modular and transparent view of TTS allows the greatest flexibility in document analysis, provides for direct *authoring* of structure and other customization, while allowing a split between expensive, multipurpose natural language analysis and the core TTS functionality. Although other text format or markup language, such as Adobe Acrobat or Microsoft Word, can be used for the same purpose, the choice of XML is obvious because it is the widely open standard, particularly for the Internet.

XML is a set of conventions for indicating the semantics and scope of various entities that combine to constitute a document. It is conceptually somewhat similar to Hypertext Markup Language (HTML), which is the exchange code for the World Wide Web. In these markup systems, properties are identified by tags with explicit scope, such as `"<b>make this phrase bold</b>"` to indicate a heavy, dark print display. XML in particular attempts to enforce a principled separation between document structure and content, on one hand, and the detailed formatting or presentation requirements of various *uses* of documents, on the other. Since we cannot provide a tutorial on XML here, we freely introduce example tags that indicate document and linguistic structure. The interpretations of these are intuitive to most readers, though, of course, the analytic knowledge underlying decisions to insert tags may be very sophisticated. It will be some time before commercial TTS engines come to a common understanding on the wide variety of text attributes that should be marked, and accept a common set of conventions. Nevertheless, it is reasonable to adopt the idea that TAM should be independent and reusable, thus allowing XML documents (which are expected to proliferate) to function for speech just as for other modalities, as indicated schematically in Figure 14.3.

TTS is regarded in Figure 14.3 as a factored process, with the text analysis perhaps carried out by human editors or by natural language analysis systems. The role of the TTS engine per se may eventually be reduced to the interpretation of structural tags and provision of phonetic information. While commercial engines of the present day are not structured with these assumptions in mind, modularity and transparency are likely to become increasingly important. The increasing acceptance of the basic ideas underlying an XML documentcentric approach to text and phonetic analysis for TTS can be seen in the recent proliferation of XML-like speech markup proposals [24, 33]. While not presenting any of these in detail, in the discussion below we adopt informal conventions that reflect and extend their basic assumptions. The structural markup exploited by the TTS systems of the

future may be imposed by XML authoring systems at document creation time, or may be inserted by independent analytical procedures. In any case the distinction between purely automatic structure creation/detection and human annotation and authoring will increasingly blur—just as in natural language translation and information retrieval domains, the distinction between machine-produced results and human-produced results has begun to blur.

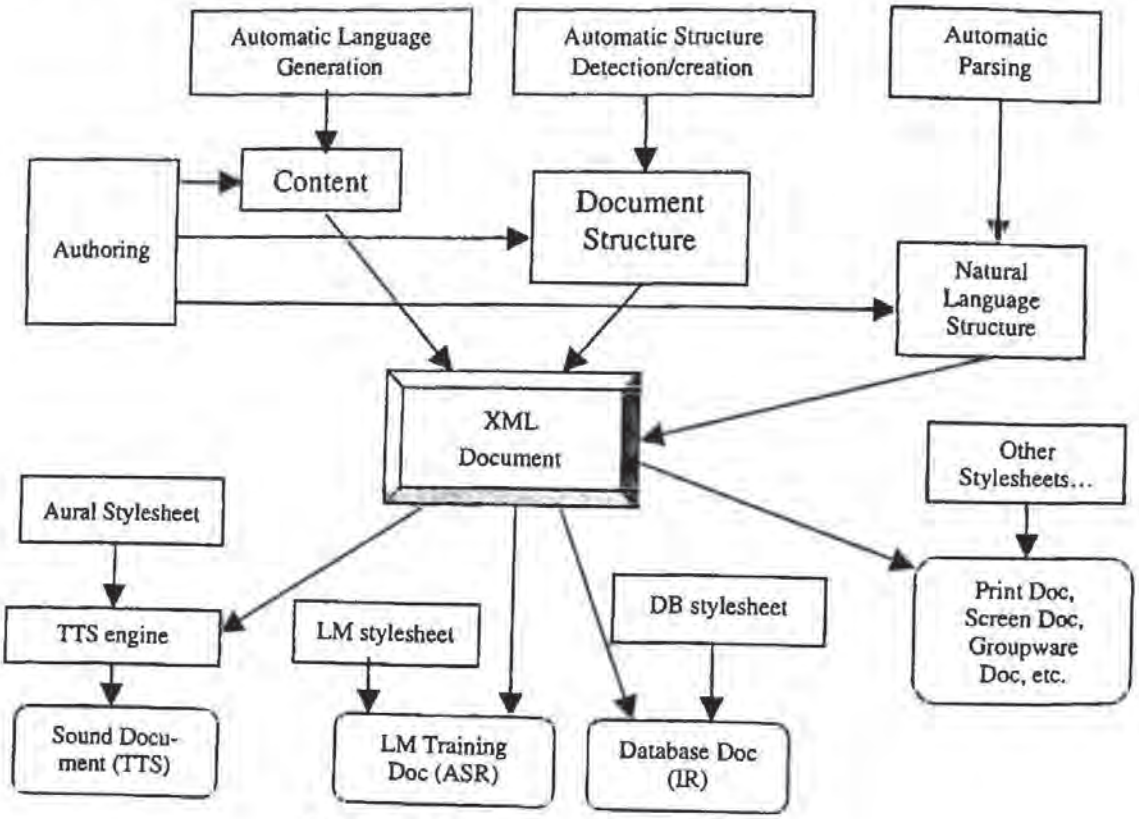


Figure 14.3 A documentcentric view of TTS.

14.3.1. Chapter and Section Headers

Section headers are a standard convention in XML document markup, and TTS systems can use the structural indications to control prosody and to regulate prosodic style, just as a professional reader might treat chapter headings differently. Increasingly, a document created on computer or intended for any kind of electronic circulation incorporates structural markup, and the TTS and audio human-computer-interface systems of the future learn to exploit this (in longer documents, the document structure markup assists in audio navigation, speedup, and skipping). For example, the XML annotation of a book at a high level might follow conventions as shown in Figure 14.4. Viewing a document in this way might lead a TTS system to insert pauses and emphasis correctly, in accordance with the structure

marked. Furthermore, an audio interface system would work jointly with a TTS system to allow easy navigation and orientation within such a structure. If future documents are marked up in this fashion, the concept of audio books, for example, would change to rely less on unstructured prerecorded speech and more on smart, XML-aware, high-quality audio navigation and TTS systems, with the output customization flexibility they provide.

For documents without explicit markup information for section and chapter headers, it is in general a nontrivial task to detect them automatically. Therefore, most TTS systems today do not make such an attempt.

```
<Book>
   <Title>The Pity of War</Title>
    <Subtitle>Explaining World War I</Subtitle>
   <Author>Niall Ferguson</Author>
   <TableOfContents>...</TableOfContents>
   <Introduction>
         <Para>...</Para>
         ...
   </Introduction>
   <Chapter>
   <ChapterTitle>The Myths of Militarism</ChapterTitle>
         <Section>
               <SectionTitle>Prophets</SectionTitle>
               <Para> ... </Para>
               ...
         </Section>
    </Chapter>
    ...
</Book>
```

Figure 14.4 An example of the XML annotation of a book.

14.3.2. Lists

Lists or bulleted items may be rendered with distinct intonational contours to indicate aurally their special status. This kind of structure might be indicated in XML as shown in Figure 14.5. Again, TTS engine designers need to get used to the idea of accepting such markup for interpretation, or incorporating technologies that can detect and insert such markup as needed by the downstream phonetic processing modules. Similar to chapter and section headers, most TTS systems today do not make an attempt to detect list structures automatically.

```
<UL>
<LI>compression</LI>
<LI>flexibility</LI>
<LI>text-waveform correspondence</LI>
</UL>
<Caption>The advantages of TTS</Caption>
```

Figure 14.5 An example of a list marked by XML.

14.3.3. Paragraphs

The paragraph has been shown to have direct and distinctive implications for pitch assignment in TTS [26]. The pitch range of good readers or speakers in the first few clauses at the start of a new paragraph is typically substantially higher than that for mid-paragraph sentences, and it narrows further in the final few clauses, before resetting for the next paragraph. Thus, to mimic a high-quality reading style in future TTS systems, the paragraph structure has to be detected from XML tagging or inferred from inspection of raw formatting. Obviously, relying on independently motivated XML tagging is, as always, the superior option, especially since this is a very common structural annotation in XML documents.

In contrast to other document structure information, paragraphs are probably among the easiest to detect automatically. The character <CR> (carriage return) or <NL> (new line) is usually a reliable clue for paragraphs.

14.3.4. Sentences

While sentence breaks are not normally indicated in XML markup today, there is no reason to exclude them, and knowledge of the sentence unit can be crucial for high-quality TTS. In fact, some XML-like conventions for text markup of documents to be rendered by synthesizers (e.g., SABLE) provide for a DIV (division) tag that could take paragraph, sentence, clause, etc. as attribute [24]. If we define sentence broadly as a primal linguistic unit that makes up paragraphs, attributes could be added to a `Sent` tag to express whatever linguistic knowledge exists about the type of the sentence as a whole:

```
<Sent type="yes-no question">
Is life so dear, or peace so sweet, as to be purchased at the price of chains and slavery?
</Sent>
```

Again, as emphasized throughout this section, such annotation could be either applied during creation of the XML documents (of the future) or inserted by independent processes. Such structure-detection processes may be motivated by a variety of needs and may exist outside the TTS system per se.

If no independent markup of sentence structure is available from an external, independently motivated document analysis or natural language system, a TTS system typically relies on simple internal heuristics to guess at sentence divisions. In email and other relatively informal written communications, sentence boundaries may be very hard to detect. In contrast to English, sentence breaking could be trivial for some other written languages. In Chinese, there is a designated symbol (a small circle ◦) for marking the end of a sentence, so the sentence breaking could be done in a totally straightforward way. However, for most Asian languages, such as Chinese, Japanese, and Thai, there is in general no space within a sentence. Thus, tokenization is an important issue for Asian languages.

In more formal English writing, sentence boundaries are often signaled by terminal punctuation from the set: { . ! ? } followed by whitespaces and an upper-case initial word. Sometimes additional punctuation may trail the '?' and '!' characters, such as close quotation marks and/or close parenthesis. The character '.' is particularly troubling, because it is, in programming terms, heavily *overloaded*. Apart from its uses in numerical expressions and Internet addresses, its other main use is as a marker of abbreviation, itself a difficult problem for text normalization (see Section 14.4). Consider this pathological jumble of potentially ambiguous cases:

> Mr. Smith came by. He knows that it costs $1.99, but I don't know when he'll be back (he didn't ask, "when should I return?")... His Web site is www.mrsmithhhhhh.com. The car is 72.5 in. long (we don't know which parking space he'll put his car in.) but he said "...and the truth shall set you free," an interesting quote.

Some of these can be resolved in the linguistic analysis module. However for some cases, only probabilistic guesses can be made, and even a human reader may have difficulty. The ambiguous sentence breaking can also be resolved in an abbreviation-processing module (described in Section 14.4.1). Any period punctuation that is not taken to signal an abbreviation and is not part of a number can be taken as end-of-sentence. Of course, as we have seen above, abbreviations are also confusable with words that can naturally end sentences, e.g., "*in*." For the measure abbreviations, an examination of the left context (checking for numeric) may be sufficient. In any case, the complexity of sentence breaking illustrates the value of passing multiple hypotheses and letting later, more knowledgeable modules (such as an abbreviation or linguistic analysis module) make decisions. Algorithm 14.1 shows a simple sentence-breaking algorithm that should be able to handle most cases.

For advanced sentence breakers, a weighted combination of the following kinds of considerations may be used in constructing algorithms for determining sentence boundaries (ordered from easiest/most common to most sophisticated):

- Abbreviation processing—Abbreviation processing is one of the most important tasks in text normalization and will be described in detail in Section 14.4.
- Rules or CART built (Chapter 4) upon features based on: document structure, whitespace, case conventions, etc.
- Statistical frequencies on sentence-initial word likelihood

- Statistical frequencies of typical lengths of sentences for various genres
- Streaming syntactic/semantic (linguistic) analysis—Syntactic/semantic analysis is also essential for providing critical information for phonetic and prosodic analysis. Linguistic analysis will be described in Section 14.5.

As you can see, a deliberate sentence breaking requires a fair amount of linguistic processing, like abbreviation processing and syntactic/semantic analysis. Since this type of analysis is typically included in the later modules (text normalization or linguistic analysis), it might be a sensible decision to delay the decision for sentence breaking until later modules, either text normalization or linguistic analysis. In effect, this arrangement can be treated as the document structure module passing along multiple hypotheses of sentence boundaries, and it allows later modules with deeper linguistic knowledge (text normalization or linguistic analysis) to make more intelligent decisions.

Finally, if a long buffer of unpunctuated words is presented, TTS systems may impose arbitrary limits on the length of a sentence for later processing. For example, the writings of the French author Marcel Proust contain some sentences that are several hundred words long (average sentence length for ordinary prose is about 15 to 25 words).

ALGORITHM 14.1: *A SIMPLE SENTENCE-BREAKING ALGORITHM*

1. **If** found punctuation ./!/? advance one character and **goto 2.**
   else advance one character and **goto 1.**
2. **If** not found whitespace advance one character and **goto 1.**
3. **If** the character is period (.) **goto 4.**
   else goto 5.
4. Perform abbreviation analysis.
   If not an abbreviation **goto 5.**
   else advance one character and **goto 1.**
5. Declare a sentence boundary and sentence type ./!/?
   Advance one character and **goto 1.**

14.3.5. Email

TTS could be ideal for reading email over the phone or in an eyes-busy situation such as when driving a motor vehicle. Here again we can speculate that XML-tagged email structure, minimally something like the example in Figure 14.6, will be essential for high-quality prosody, and for controlling the audio interface, allowing skips and speedups of areas the user has defined as less critical, and allowing the system to announce the function of each block. For example, the `sig` (signature) portion of email certainly has a different semantic function than the main message text and should be clearly identified as such, or skipped, at the listener's discretion. Modern email systems are providing increasingly sophisticated

support for structure annotation such as that exemplified in Figure 14.6. Obviously, the email document structure can be detected only with appropriate tags (like XML). It is very difficult for a TTS system to detect it automatically.

```
<message>
  <header>
      <date>11 June 1998</date>
      <from>Leslie</from>
      <to>Jo</to>
      <subject>Surf's Up!</subject>
  </header>
  <body> ... </body>
  <sig>Freedom's just another word for nothing left to lose</sig>
</message>
```

Figure 14.6 An example of email marked by XML.

14.3.6. Web Pages

All the comments about TTS reliance on XML markup of document structure can be applied to the case of HTML-marked Web page content as well. In addition to sections, headers, lists, paragraphs, etc., the TTS systems should be aware of XML/HTML conventions such as links (link name) and perhaps apply some distinctive voice quality or prosodic pitch contour to highlight these. The size and color of the section of text also provides useful hints for emphasis. Moreover, the TTS system should also integrate the rendering of audio and video contents on the Web page to create a genuine multimedia experience for the users. More could be said about the rendition of Web content, whether from underlying XML documents or HTML-marked documents prepared specifically for Web presentation. In addition, the World Wide Web Consortium has begun work on standards for aural stylesheets that can work in conjunction with standard HTML to provide special direction in aural rendition [33].

14.3.7. Dialog Turns and Speech Acts

Not all text to be rendered by a TTS system is standard written prose. The more expressive TTS systems could be tasked with rendering natural conversation and dialog in a spontaneous style. As with written documents, the TTS system has to be guided by XML markup of its input. Various systems for marking *dialog turns* (change of speaker) and *speech acts* (the mood and functional intent of an utterance)[3] are used for this purpose, and these annotations will trigger particular phonetic and prosodic rules in TTS systems. The speech act coding

[3] Dialog modeling and the concepts of *dialog turns* and *speech acts* are described in detail in Chapter 17.

schemes can help, for example, in identifying the speaker's intent with respect to an utterance, as opposed to the utterance's structural attributes. The prosodic contour and voice quality selected by the TTS system might be highly dependent on this functional knowledge.

For example, a syntactically well-formed question might be used as information solicitation, with the typical utterance-final pitch upturn as shown in the following:

<REQUEST_INFO>Can you hand me the wrench?</REQUEST_INFO>

But if the same utterance is used as a command, the prosody may change drastically.

<DIRECTIVE>Can you hand me the wrench.</DIRECTIVE>

Research on speech act markup-tag inventories (see Chapter 17) and automatic methods for speech act annotation of dialog is ongoing, and this research has the property considered desirable here, in that it is independently motivated (useful for enhancing speech recognition and language understanding systems). Thus, an advanced TTS system should be expected to exploit dialog and speech act markups extensively.

14.4. TEXT NORMALIZATION

Text often include abbreviations (e.g., FDA for *Food and Drug Administration*) and acronyms (SWAT for *Special Weapons And Tactics*). Novels and short stories may include *spoken* dialog interspersed with exposition; technical manuals may include mathematical formulae, graphs, figures, charts and tables, with associated captions and numbers; email may require interpretation of special conventional symbols such as *emoticons* [e.g., :-) means smileys], as well as Web and Internet address formats, and special abbreviations (e.g., IMHO means *in my humble opinion*). Again, any text source may include part numbers, stock quotes, dates, times, money and currency, and mathematical expressions, as well as standard ordinal and cardinal formats. Without context analysis or prior knowledge, even a human reader would sometimes be hard pressed to give a perfect rendition of every sequence of nonalphabetic characters or of every abbreviation. *Text normalization* (TN) is the process of generating normalized orthography (or, for some systems, direct generation of phones) from text containing words, numbers, punctuation, and other symbols. For example, a simple example is given as follows:

The 7% Solution → THE SEVEN PER CENT SOLUTION

Text normalization is an essential requirement not only for TTS, but also for the preparation of training text corpora for acoustic-model and language-model construction.[4] In addition, speech dictation systems face an analogous problem of *inverse* text normalization for document creation from recognized words, and such systems may depend on knowledge sources similar to those described in this section. The example of an inverse text normalization for the example above is given as follows:

[4] For details of acoustic and language modeling, please refer to Chapters 9 and 11.

THE SEVEN PER CENT SOLUTION → The 7% Solution

Modular text normalization components, which may produce output for multiple downstream consumers, mark up the exemplary text along the following lines:

The <tn snor="SEVEN PER CENT">7%</tn> Solution

The `snor` tag stands for *Standard Normalized Orthographic Representation.*[5] For TTS, input text may include multisentence paragraphs, numbers, dates, times, punctuation, symbols of all kinds, as well as interpretive annotations in a TTS markup language, such as tags for word emphasis or pitch range. Text analysis for TTS is the work of converting such text into a stream of normalized orthography, with all relevant input tagging preserved and new markup added to guide the subsequent modules. Such interpretive annotations added by text analysis are critical for phonetic and prosodic generation phases to produce desired output. The output of the text normalizer may be deterministic, or may preserve a full set of interpretations and processing history with or without probabilistic information to be passed along to later stages. We once again assume that XML markup is an appropriate format for expressing knowledge that can be created by a variety of external processes and exploited by a number of technologies in addition to TTS.

Since today's TTS systems typically cannot expect that their input be independently marked up for text normalization, they incorporate internal technology to perform this function. Future systems may piggyback on full natural language processing solutions developed for independent purposes. Presently, many incorporate minimal, TTS-specific hand-written rules [1], while others are loose agglomerations of modular, task-specific statistical evaluators [3].

For some purposes, an architecture that allows for a set or lattice of possible alternative expansions may be preferable to deterministic text normalization, like the *n-best* lists or *word graph* offered by the speech recognizers described in Chapter 13. Alternatives known to the system can be listed and ranked by probabilities that may be learnable from data. Later stages of processing (linguistic analysis or speech synthesis) can either add knowledge to the lattice structure or recover the best alternative, if needed. Consider the fragment "*at 8 am I . . .* " in some informal writing such as email. Given the flexibility of writing conventions for pronunciation, *am* could be realized as either *A. M.* (the numeric context seems to cue at times) or the auxiliary verb *am*. Both alternatives could be noted in a descriptive lattice of covering interpretations, with confidence measures if known (Table 14.2).

Table 14.2 Two alternative interpretations for sentence fragment "*At 8 am I ...*".

At 8 am I ...	At <time> eight am </time> I ...
At 8 am I ...	At <number> eight </number> am I ...

[5] SNOR, or Standard Normalized Orthographic Representation, is a uniform way of writing words and sentences that corresponds to spoken rendition. SNOR-format sentence texts are required as reference material for many Defense Advanced Research Project Agency and National Institutes of Standards and Technology-sponsored standard speech technology evaluation procedures.

If the potential ambiguity in the interpretation of *am* in the above pair of examples is simply noted, and the alternatives retained rather than suppressed, the choice can be made by a later stage of syntactic/semantic processing. Note another feature of this example—the rough irregular abbreviation form for antemeridian, which by prescriptive convention hopes that high-quality TTS processing can rely entirely on *standard* stylistic conventions. That observation also applies to the *obligatory* use of "?" for all questions.

Specific architectures for the text normalization component of TTS may be highly variable, depending on the system architect's answers to the following questions:

- Are cross-functional language processing resources mandated, or available?
- If so, are phonetic forms, with stress or accent, and normalized orthography, available?
- Is a full syntactic and semantic analysis of input text mandated, or available?
- Can the presenting application add interpretive knowledge to structure the input (text)?
- Are there interface or pipelining requirements that preclude lattice alternatives at every stage?

Because of this variability in requirements and resources, we do not attempt to formally specify a single, all-purpose architectural solution here. Rather, we concentrate on describing the text normalization challenges any system has to face. We note where solutions to these challenges are more readily realized under particular architectural assumptions.

All text normalization consists of two phases: identification of type, and expansion to SNOR or other unambiguous representation. Much of the identification phase, dealing with phenomena of sentence boundary determination, abbreviation expansion, number spell-out, etc., can be modeled as regular expression (see Chapter 11). This raises an interesting architectural issue. Imagine a system based entirely on regular *finite state transducers* (FST, see Chapter 11), as in [27], which enforces an appealing uniformity of processing mechanism and internal structure description. The FST permits a lattice-style representation that does not require premature resolution of any structural choice. An entire text analysis system can be based on such a representation. However, as long as a system confines its attention to issues that commonly come under the heading of text normalization, such as number formats, abbreviations, and sentence breaking, a simpler regular-expression-based uniform mechanism for rule specification and structure representation may be adequate.

Alternatively, TTS systems could make use of advanced tools such as, for example, the lex and yacc tools [17], which provide frameworks for writing customized lexical analyzers and context-free grammar parsers, respectively. In the discussion of typical text normalization requirements below, examples will be provided and then a fragment of Perl pattern-matching code will be shown that allows matching of the examples given. Perl notation [36] is used as a convenient short-hand representing any equivalent regular expressionparsing system and can be regarded as a subset of the functionality provided by any regular expression, FST, or context-free grammar tool set that a TTS software architect may choose to employ. Only a small subset of the simple, fairly standard Perl conventions

to employ. Only a small subset of the simple, fairly standard Perl conventions for regular expression matching are used, and comments are provided in our discussion of text normalization.

A text normalization system typically adds identification information to assist subsequent stages in their tasks. For example, if the TN subsystem has determined with some confidence that a given digit string is a phone number, it can associate XML-like tags with its output, identifying the corresponding normalized orthographic chunk as a candidate for special phone-number intonation. In addition, the identification tags can guide the lexical disambiguation of terms for other processes, like phonetic analysis in TTS systems and training data preparation for speech recognition.

Table 14.3 shows some examples of input fragments with a relaxed form of output normalized orthography. It illustrates a possible ambiguity in TN output. In the (contrived) example, the ambiguity is between a place name and a hypothetical individual named perhaps *Steve* or *Samuel* Asia. Two questions arise in such cases. The first is format of specification. The data between submodules in a TTS system can be passed (or be placed in a centrally viewable *blackboard* location) as tagged text or in a binary format. This is an implementation detail. Most important is that all possibilities known to the TN system be specified in the output, and that confidence measures from the TN, if any, be represented. For example, in many contexts, *South Asia* is the more likely spell-out of *S. Asia*, and this should be indicated implicitly by ordering output strings, or explicitly with probability numbers. The decision could then be delayed until one has enough information in the later module (like linguistic analysis) to make the decision in an informed manner.

Table 14.3 Examples of the normalized output using XML-like tags for text normalization.

Dr. King	<title> DOCTOR </title> KING
7%	<number>SEVEN<ratio>PERCENT</ratio> </number>
S. Asia	<toponym> SOUTH ASIA </toponym> OR <psn_name><initial>S</initial>ASIA</psn_name>

14.4.1. Abbreviations and Acronyms

As noted above, a period is an important but not completely reliable clue to the presence of an abbreviation. Periods may be omitted or misplaced in text for a variety of reasons. For similar reasons of stylistic variability and a writer's (lack of) care and skill, capitalization, another potentially important clue, can be variable as well. For example, all the representations of the abbreviation for *post script* listed below have been observed in actual mail and email. A system must therefore combine knowledge from a variety of contextual sources, such as document structure and origin, when resolving abbreviations:

PS. Don't forget your hat.
Ps. Don't forget your hat.
P.S. Don't forget your hat.
P.s. Don't forget your hat.

And *P.S.*, when examined out of context, could be personal name initials as well. Of course, a given TTS system's user may be satisfied with the simple spoken output */p iy ae s/* in cases such as the above, obviating the need for full interpretation. But at a minimum, when *fallback to letter pronunciation* is chosen, the TTS system must attempt to ensure that some obvious spell-out is not being overlooked. For example, a system should not render the title in *Dr. Jones* as letter names */d iy aa r/*.

Actually, any abbreviation is potentially ambiguous, and there are several distinct types of ambiguity. For example, there are abbreviations, typically quantity and measure terms, which can be realized in English as either plural or singular depending on their numeric coefficient, such as *mm* for *millimeter(s)*. This type of ambiguity can get especially tricky in the context of conventionally frozen items. For example, *9mm ammunition* is typically spoken as *nine millimeter ammunition* rather than *nine millimeters ammunition*.

Next, there are forms that can, with appropriate syntactic context, be interpreted either as abbreviations or as simple English words, such as in (inches), particularly at the end of sentences.

Finally, many, perhaps most, abbreviations have entirely different abbreviation spell-outs depending on semantic context, such as *DC* for *direct current* or *District of Columbia*. This variability makes it unlikely that any system ever performs perfectly. However, with sufficient training data, some statistical guidelines for interpretation of common abbreviations in context can be derived. Table 14.4 shows a few more examples of this most difficult type of ambiguity.

An advanced TTS system should attempt to convert reliably at least the following abbreviations:

- Title—Dr., MD, Mr., Mrs., Ms., St. (Saint), ... etc.
- Measure—ft., in., mm, cm (centimeter), kg (kilogram), ... etc.
- Place names—CO, LA, CA, DC, USA, St. (street), Dr. (drive), ... etc.

Table 14.4 Some ambiguous abbreviations.

CO	Colorado	commanding officer
	conscientious objector	carbon monoxide
IRA	Individual Retirement Account	Irish Republican Army
MD	Maryland	doctor of medicine
	muscular dystrophy	

Abbreviation disambiguation usually can be resolved by POS (part-of-speech) analysis. For example, whether *Dr.* is *Doctor* or *Drive* can be resolved by examining the POS features of the previous and following words. If the abbreviation is followed by a capitalized personal name, it can be expanded as *Doctor*, whereas if the abbreviation is preceded by a capitalized place name, a number, or an alphanumeric (like 120th), it will be expanded as *Drive*. Although the example above is resolved via a series of heuristic rules, the disambiguation (POS analysis) can also be done by a statistical approach. In [6], the POS tags are determined based on the most likely POS sequence using POS trigram and lexical-POS unigram. Since an abbreviation can often be distinguished by its POS feature, the most likely POS sequence of the sentence discovered by the trigram search then provides the best guess of the POS (thus the usage) for abbreviations. We describe POS tagging in more detail in Section 14.5.

Other than POS information, the lexical entries for abbreviations should include all features and alternatives necessary to generate a lattice of possible analyses. For example, a typical abbreviation's entry might include information as to whether it could be a word (like *in*), whether period(s) are optional or required, whether plural variants must be generated and if so under what circumstances, whether numerical specification is expected or required, etc.

Acronyms are words created from the first letters or parts of other words. For example, SCUBA is an acronym for *self-contained underwater breathing apparatus*. Generally, to qualify as a true acronym, a letter sequence should reflect normal language phonotactics, such as a reasonable alternation of consonants and vowels. From a TTS system's point of view, the distinctions between acronyms, abbreviations, and plain new or unknown words can be unclear. Many acronyms can be entered into the TTS system lexicon just as ordinary words would be. However, unknown acronyms (not listed in the lexicon) may occasionally be encountered. Although an acronym's case property can be a significant clue to identification, it is often unclear how to speak a given sequence of upper-case letters. Most TTS systems, failing to locate the sequence in the acronym dictionary, spell it out letter-by-letter. Other systems attempt to determine whether the sequence is inherently *speakable*. For example, DEC might be inherently speakable, while FCC is not formed according to normal word phonotactics. When something speakable is found, it is processed via the normal letter-to-sound rules, while something *unspeakable* would be spelled out letter-by-letter. Yet other systems might simply feed the sequence directly to the letter-to-sound rules (see Section 14.8), just as they would any other unknown word. As with all such problems, a larger lexicon usually provides superior results.

The general algorithm for abbreviations and acronyms expansion in text normalization is summarized in Algorithm 14.2. The algorithm assumes that tokenization and POS tagging have been done for the whole sentence. Abbreviation expansion is determined by the POS tags of the potential abbreviation candidates. Acronym expansion is done exclusively by table lookup, and letter-by-letter spell-out is used when acronyms cannot be found in the acronym table.

ALGORITHM 14.2: *ABBREVIATIONS AND ACRONYMS EXPANSION*

1. If word token *w* is not in abbreviation table and *w* contains only capital letters **goto 3.**
2. **Abbreviation Expansion**
   If the POS tag of *w* and the correspondent abbreviation match
   Abbreviation expansion by inserting SNOR and interpretive annotation tags
   Advance one word and **goto 1.**
3. **Acronym Expansion**
   If *w* is in the predefined acronym table
   Acronym expansion by inserting SNOR and interpretive annotation tags
   according to acronym expansion table
   else spell out *w* letter-by-letter
4. Advance one word and **goto 1.**

14.4.2. Number Formats

Numbers occur in a wide variety of formats and have a wide variety of contextually dependent reading styles. For example, the digits 370 in the context of the product name *IBM 370 mainframe computer* typically are read as *three seventy*, while in other contexts 370 would be read as *three hundred seventy* or *three hundred and seventy*. In a phone number, such as 370-1111, the string would normally be read as *three seven oh*, while in still other contexts it might be rendered as *three seven zero*. A text analysis system can incorporate rules, perhaps augmented by probabilities, for these situations, but might never achieve perfection in all cases. Phone numbers are a practical place to start, and their treatment illustrates some of the general issues relevant to the other number formats which are covered below.

14.4.2.1. Phone Numbers

Phone numbers may include prefixes and area codes and may have dashes and parentheses as separators. Examples are shown in Table 14.5.

The first two examples have *prefix* codes, while the next four have area codes with minor formatting differences. The final two examples are possible international-format phone numbers. A basic Perl regular expression pattern to subsume the commonality in all the local domestic numbers can be defined as follows:

```
$us_basic = '([0-9]{3}\-[0-9]{4})';
```

This defines a pattern subpart to match 3 digits, followed by a separator dash, followed by another 4 digits. Then the pattern to match the prefix type would be:

```
/([0-9]{1})[\/ -]($us_basic)/
```

Table 14.5 Some different written representations of phone numbers.

9-999-4118
9 345-5555
(617) 932-9209
(617) 932-9209
716-123-4568
409/845-2274
+49 (228) 550-381
+49-228-550-381

In the first example above, this leaves the system pattern variable $1 (corresponding to the first set of capture parentheses in the pattern) set to 9, and $2 (the second set of capture parentheses) set to 999-4118. Then a separate set of tables, indexed by the rule name and the pattern variable contents, could provide orthographic spell-outs for the digits. Clearly a balance has to be struck between the number of pattern variables provided in the expression and the overall complexity of the expression, vis-à-vis the complexity and sophistication of the indexing scheme of the spell-out tables. For example, the $us_basic could be defined to incorporate parentheses capture on the first three digits and the remaining four separately, which might lead to a simpler spell-out table in some cases.

The pattern to match the area code types could be:

```
/(\([0-9]{3}\))[\/ -]($us_basic)/
```

These patterns could be endlessly refined, expanded, and layered to match strings of almost arbitrary complexity. A balance has to be struck between number and complexity of distinct patterns. In any case, no matter how sophisticated the matching mechanism, arbitrary or at best probabilistic decisions have to be made in constructing a TTS system. For example, in matching an area code type, the rule architect must decide how much and what kind of whitespace separation the matching system tolerates between the area code and the rest of the number before a phone-number match is considered unlikely. Or, as another example, does the rule architect allow new lines or other formatting characters to appear between the area code and the basic phone number? These kinds of decisions must be explicitly considered, or made by default, and should be specified to a reasonable degree in user documentation. There are a great many other phone number formats and issues that are beyond the scope of this treatment.

Once a certain type of pattern requires a conversion to normalized orthography, the question of how to perform the conversion arises. The conversion characters can be aligned with the identification, so that conversion occurs implicitly during the pattern matching process. Another way is to separate the conversion from the identification phase. This may or may not lead to gains in efficiency and elimination of redundancy, depending on the

overall architecture of the system and whether and how components are expected to be reused. A version of this second approach is sketched here.

Suppose that the pattern match variable $1 has been set to 617 by one of the identification-phase pattern matches described above. Another list can provide pointers to conversion tables, indexed by the rule name or number and the variable name. So for the rule that can match area codes, the relevant entry would be:

```
Identification rule     Variable     Spellout table
Area-Phone              $1           LITERAL_DIGIT
```

The LITERAL_DIGIT spell-out rule set, when presented with the 617 character sequence (the value of $1), simply generates the normalized orthography *six one seven*, by table lookup. In this simple and straightforward approach, spell-out tables such as LITERAL_DIGIT can be reused for portions of a wide variety of identification rules. Other simple numeric spell-out tables would cover different styles of numeric reading, such as *pairwise* style (e.g., *six seventeen*), full decimal with tens, hundreds, thousands units (*six hundred seventeen*), and so on. Some spellout tables may require processing code to supplement the basic table lookup. Additional examples of spell-out tables are not provided for the various other types of text normalization entities exemplified below, but would function similarly.

14.4.2.2. Dates

Dates may be specified in a wide variety of formats, sometimes with a mixture of orthographic and numeric forms. Note that dates in TTS suffer from a mild form of the century-date-change uncertainty (the infamous Y2K bug), so a form such as 5/7/37 may in the future be ambiguous, in its full form, between 1937 and 2037. The safest course is to say as little as possible, i.e., "*five seven thirty seven*", or even "*May seventh, thirty seven*", rather than attempt "*May seventh, nineteen thirty seven*". Table 14.6 shows a variety of date formats and associated normalized orthography.

Table 14.6 Various date formats.

12/19/94 (US)	December nineteenth ninety four
19/12/94 (European)	December nineteenth ninety four
04/27/1992	April twenty seventh nineteen ninety two
May 27, 1995	May twenty seventh nineteen ninety five
July 4, 94	July fourth ninety four
1,994	one thousand nine hundred and ninety four
1994	nineteen ninety four

One issue that comes up with certain number formats, including dates, is range checking. A form like 13/19/94 is basically uninterpretable as a date. This kind of checking, if included in the initial pattern matching, may be slow and may increase formal requirements for power of the pattern matching system. Therefore, range checking can be done at spell-out time (see below) during normalized orthography generation, as long as a backtracking or redo option is present. If range checking is desired as part of the basic identification phase of text normalization, some regular expression matching systems allow for extensions. For example, the following pattern variable matches only numbers less than or equal to 12, the valid month specifications. It can be included as part of a larger, more complex date matching *pattern:*

```
$month = '/(0[123456789]/1[012]/'
```

14.4.2.3. Times

Times may include hours, minute, seconds, and duration specifications as shown in Table 14.7. Time formats exemplify yet another area where linguistic concerns have to intersect with architecture. If simple, flat normalized orthography is generated during a text normalization phase, a later stage may still find a form like *am* ambiguous in pronunciation. If a lattice of alternative interpretations is provided, it should be supplemented with interpretive information on the linguistic status of the alternative text analyses. Alternatively, a single best guess can be made, but even in this case, some kind of interpretive information indicating the status of the choice as, e.g., a time expression, should be provided for later stages of syntactic, semantic, and prosodic interpretation. This reiterates the importance of TTS text analysis systems to generate interpretive annotations tags for subsequent modules' use whenever possible, as discussed in Section 14.4. In some cases, unique text formatting of the choice, corresponding to the system's lexical contents, may be sufficient. That is, in some systems, generation of *A.M.*, for example, may uniquely correspond to the lexicon's entry for that portion of a time expression, which specifies the desired pronunciation and grammatical treatment.

Table 14.7 Several examples for time expressions.

11:15	eleven fifteen
8:30 pm	eight thirty pm
5:20 am	five twenty am
12:15:20	twelve hours fifteen minutes and twenty seconds
07:55:46	seven hours fifty-five minutes and forty-six seconds

14.4.2.4. Money and Currency

As illustrated in Table 14.8, money and currency processing should correctly handle at least the currency indications $, £, DM, ¥, and €, standing for dollars, British pounds, deutsche marks, Japanese yen, and euros, respectively. In general, $ and £ have to precede the numeral; DM, ¥, and € have to follow the numeral. Other currencies are often written in full words and have to follow the numeral, though abbreviations for these are sometimes found, such as *100 francs* and *20 lira*.

Table 14.8 Several money and currency expressions.

$40	forty dollars
£200	two hundred pounds
5 ¥	five yen
25 DM	twenty five deutsche marks
300 €	three hundred euros

14.4.2.5. Account Numbers

Account numbers may refer to bank accounts or social security numbers. Commercial product part numbers often have these kinds of formats as well. In some cases these cannot be readily distinguished from mathematical expressions or even phone numbers. Some examples are shown below:

```
123456-987-125456
000-1254887-87
049-85-5489
```

The other popular number format is that of credit card number, such as

```
4446-2289-2465-7065
3745-122267-22465
```

To process formats like these, it may eventually be desirable for TTS systems to provide customization capabilities analogous to the pronunciation customization features for words found in current TTS systems. Regular expression formalisms of the type exemplified above for phone number, would, if exposed to applications and developers through suitable editors, be adequate for most such needs.

14.4.2.6. Ordinal Numbers

Ordinal numbers are those referring to rank or placement in a series. Examples include:

1st, 2nd, 3rd, 4th, 10th, 11th, 12th, 20th, 100th, 1000th, etc.
1st, 2nd, 3rd, 4th, 10th, 11th, 12th, 20th, 21st, 32nd, 100th, 1000th, etc.

The system's ordinal processing may also be used to generate the denominators of fractions, except for halves, as shown in Table 14.9. Notice that the ordinal must be plural for numerators other than 1.

Table 14.9 Some examples of fractions.

1/2	one half
1/3	one third
1/4	one quarter or one fourth
1/10	one tenth
3/10	three tenths

14.4.2.7. Cardinal Numbers

Cardinal numbers are, loosely speaking, those forms used in simple counting or the statement of amounts. If a given sequence of digits fails to fit any of the more complex formats above, it may be a simple cardinal number. These may be explicitly negative or positive or assumed positive. They may include decimal or fractional specifications. They may be read in several different styles, depending on context and/or aesthetic preferences. Table 14.10 gives some examples of cardinal numbers and alternatives for normalized orthography.

The number-expansion algorithm is summarized in Algorithm 14.3. In this algorithm the text normalization module maintains an extensive pattern table. Each pattern in the table contains its associated pattern in regular expression or Perl format along with a pointer to a rule in the conversion table, which guides the expansion process.

Table 14.10 Some cardinal number types.

123	one two three	one hundred (and) twenty three
1,230	one thousand two hundred (and) thirty	
2426	two four two six	twenty four twenty six
	two thousand four hundred (and) twenty six	

A regular expression to match well-formed cardinals with commas grouping chunks of three digits of the type from 1,000,000 to 999,999,999 might appear as:

```
if ($item =~ /^([0-9]{1,3}),([0-9]{3}),([0-9]{3})/
 {      $NewFrame->{"millions"} = $1;
  $NewFrame->{"thousands"} = $2;
  $NewFrame->{"hundreds"} = $3;
   print "Grouped cardinal found: $item\n";
  return $NewFrame;   }
```

ALGORITHM 14.3: *NUMBER EXPANSION*

1. **Pattern Matching**
   If a match is found **goto 2.**
   else goto 3.
2. **Number Expansion**
   Insert SNOR and interpretive annotation tags according to the associated rule
   Advance the pointer to the right of the match pattern and **goto 1.**
3. **Finish**

14.4.3. Domain-Specific Tags

In keeping with the theme of this section—that is, the increasing importance of independently generated precise markup of text entities—we present a little-used but interesting example.

14.4.3.1. Mathematical Expressions

Mathematical expressions are regarded by some systems as the domain of special-purpose processors. It is a serious question how far to go in mathematical expression parsing, since providing some capability in this area may raise users' expectations to an unrealistic level. The World Wide Web Consortium has developed MathML (mathematical markup language) [34], which provides a standard way of describing math expressions. MathML is an XML extension for describing mathematical expression structure and content to enable mathematics to be served, received, and processed on the Web, similar to the function HTML has performed for text. As XML becomes increasingly pervasive, MathML could possibly be used to guide interpretation of mathematical expressions. For the notation $(x + 2)^2$ a possible MathML representation such as that below might serve as an initial guide for a spoken rendition.

```
<EXPR>
 <EXPR>
 x
 <PLUS/>
 2
 </EXPR>
 <POWER/>
 2
</EXPR>
```

This might be generated by an application or by a specialized preprocessor within the TTS system itself. Prosodic rules or data tables appropriate for math expressions could then be triggered.

14.4.3.2. Chemical Formulae

As XML becomes increasingly common and exploitable by TTS text normalization, other areas follow. For example, Chemical Markup Language (CML [22]) now provides a standard way to describe molecular structure or chemical formulae. CML is an example of how standard conventions for text markup are expected increasingly to replace ad hoc, TTS-internal heuristics.

In CML, the chemical formula C_2OCOH_4 would appear as:

```
<FORMULA>
  <XVAR BUILTIN="STOICH">
  C C O C O H H H H
  </XVAR>
</FORMULA>
```

It seems reasonable to expect that TTS engines of the future will be increasingly devoted to interpreting such precise conventions in high-quality speech renditions rather than endlessly replicating NL heuristics that fail as often as they succeed in guessing the identity of raw text strings.

14.4.4. Miscellaneous Formats

A random list illustrating the range of other types of phenomena for which an English-oriented TTS text analysis module must generate normalized orthography might include:

- Approximately/tilde: The symbol ~ is spoken as *approximately* before (Arabic) numeral or currency amount, otherwise it is the character named *tilde*.

- Folding of accented Roman characters to *nearest* plain version: If the TTS system has no knowledge of dealing with foreign languages, like French or German, a table of folding characters can be provided so that for a term such as *Über-mensch*, rather than spell out the word *Über*, or ignore it, the system can convert it to its *nearest* English-orthography equivalent: *Uber*. The ultimate way to process such foreign words should integrate a language identification module with a multi-lingual TTS system, so that language-specific knowledge can be utilized to produce appropriate text normalization of all text.
- Rather than simply ignore high ASCII characters in English (characters from 128 to 255), the text analysis lexicon can incorporate a table that gives *character names* to all the printable high ASCII characters. These names are either the full Unicode character names, or an abbreviated form of the Unicode names. This would allow speaking the names of characters like © (*copyright sign*), ™ (*trademark*), @ (*at*), ® (*registered mark*), and so on.
- Asterisk: in email, the symbol '*' may be used for emphasis and for setting off an item for special attention. The text analysis module can introduce a little pause to indicate possible emphasis when this situation is detected. For the example of "*Larry has *never* been here*," this may be suppressed for asterisks spanning two or more words. In some texts, a word or phrase appearing completely in UPPER CASE may also be a signal for special emphasis.
- Emoticons: There are several possible emoticons (emotion icons).

  1. :-) or :) SMILEY FACE (humor, laughter, friendliness, sarcasm)
  2. :-(or :(FROWNING FACE (sadness, anger, or disapproval)
  3. ;-) or ;) WINKING SMILEY FACE (naughty)
  4. :-D OPEN-MOUTHED SMILEY FACE (laughing out loud)

  Smileys, of which there are dozens of types, may be tacked onto word start or word end or even occur interword without spaces, as in the following examples.

  :)hi!
  Hi:)
  Hi:)Hi!

14.5. LINGUISTIC ANALYSIS

Linguistic analysis (sometimes also referred to as syntactic and semantic parsing) of natural language (NL) constitutes a major independent research field. Often commercial TTS systems incorporate some minimal parsing heuristics developed strictly for TTS. Alternatively, the TTS systems can also take advantage of independently motivated natural language proc-

essing (NLP) systems, which can produce structural and semantic information about sentences. Such linguistically analyzed documents can be used for many purposes other than TTS—information retrieval, machine translation system training, etc.

Provision of some parsing capability is useful to TTS systems in several areas. Parsers may be used in disambiguating the text normalization alternatives described above. Additionally, syntactic/semantic analysis can help to resolve grammatical features of individual words that may vary in pronunciation according to sense or abstract inflection, such as *read*. Finally, parsing can lay a foundation for derivation of a prosodic structure useful in determining segmental duration and pitch contour.

The fundamental types of information desired for TTS from a parsing analysis are summarized below:

- Word part of speech (POS) or word type, e.g., proper name or verb.
- Word sense, e.g., river *bank* vs. money *bank*.
- Phrasal cohesion of words, such as idioms, syntactic phrases, clauses, sentences.
- Modification relations among words.
- Anaphora (co-reference) and synonymy among words and phrases.
- Syntactic type identification, such as questions, quotes, commands, etc.
- Semantic focus identification (emphasis).
- Semantic type and speech act identification, such as requesting, informing, narrating, etc.
- Genre and style analysis.

Here we confine ourselves to discussion of the kind of information that a good parser could, in principle, provide to enable the TTS-specific functionality.

Linguistic analysis supports the phonetic analysis and prosodic generation phases. The modules of phonetic analysis are covered in Sections 14.6, 14.7, and 14.8. A linguistic parser can contribute in several ways to the process of generating (symbolic) phonetic forms from orthographic words found in text. One function of a parser is to provide accurate part-of-speech (POS) labels. This can aid in resolving the pronunciation of several hundred American English homographs, such as *object* and *absent*. Homographs are discussed in greater detail in Section 14.6. Parsers can also aid in identifying names and other special classes of vocabulary for which specialized pronunciation rule sets may exist [32].

Prosody generation deals mainly with assignment of segmental duration and pitch contour that have close relationship with prosodic phrasing (pause placement) and accentuation. Parsing can contribute useful information, such as the syntactic type of an utterance. (e.g., yes/no question contours typically differ from *wh*-question contours, though both are marked simply by '?' in text), as well as semantic relations of synonymy, anaphora, and focus that may affect accentuation and prosodic phrasing. Information from discourse analysis and text genre characterization may affect pitch range and voice quality settings. Further

examination of the contribution of parsing specifically to prosodic phrasing, accentuation, and other prosodic interpretation is provided in Chapter 15.

As mentioned earlier, TTS can employ either a general-purpose NL analysis engine or a pipeline of a number of very narrowly targeted, special-purpose NL modules together for the requirement of TTS linguistic analysis. Although we focus on linguistic information for supporting phonetic analysis and prosody generation here, a lot of the information and services are beneficial to document structure detection and text normalization described in previous sections.

The minimum requirement for such a linguistic analysis module is to include a lexicon of the *closed-class* function words, of which only several hundred exist in English (at most), and perhaps homographs. In addition, a minimal set of modular functions or services would include:

- *Sentence breaking*—Sentence breaking has been discussed in Section 14.3.4 above.
- *POS tagging*—POS tagging can be regarded as a two-stage process. The first is POS guessing, which is the process of determining, through a combination of a (possibly small) dictionary and some morphological heuristics or a specialized morphological parser, the POS categories that might be appropriate for a given input term in isolation. The second is POS choosing—that is, the resolution of the POS in context, via local short-window syntactic rules, perhaps combined with probabilistic distribution for the POS guesses of a given word. Sometimes the guessing and choosing functions are combined in a single statistical framework. In [6], lexical probabilities are unigram frequencies of assignments of categories to words estimated from corpora. In the original formulation of the model, the lexical probabilities [$P(c_i \mid w_i)$, where c_i is the hypothesized POS for word w_i], were estimated from the hand-tagged Brown corpus [8]. For Example, the word *see* appeared 771 times as a verb and once as an interjection. Thus the probability that *see* is a verb is estimated to be 771/772 or 0.99. Trigrams are used for contextual probability [$P(c_i \mid c_{i-1}c_{i-2}\cdots c_1) = P(c_i \mid c_{i-1}c_{i-2})$]. Lexical probabilities and trigrams over category sequences are used to score all possible assignments of categories to words for a given input word sequence. The entire set of possible assignments of categories to words in sequence is calculated, and the best-scoring sequence is used. Likewise, simple methods have been used to detect noun phrases (NPs), which can be useful in assigning pronunciation, stress, and prosody. The method described in [6] relies on a table of probabilities for inserting an NP begin bracket '[' between any two POS categories, and similarly for an NP end bracket ']'. This was also trained on the POS-labeled Brown corpus, with further augmentation for the NP labels. For example, the probability of inserting an NP begin bracket after an article was found to be

much lower than that of begin-bracket insertion between a verb and a noun, thus automatically replicating human intuition.

- *Homograph disambiguation*—Homograph disambiguation in general refers to the case of words with the same orthographic representation (written form) but having different semantic meanings and sometimes even different pronunciations. Sometimes it is also referred as sense disambiguation. Examples include "The boy used the *bat* to hit a home run" vs. "We saw a large *bat* in the zoo" (the pronunciation is the same for two *bat*) and "You *record* your voice" vs. "I'd like to buy that *record*" (the pronunciations are different for the two *record*). The linguistic analysis module should at least try to resolve the ambiguity for the case of different pronunciations because it is absolutely required for correct phonetic rendering. Typically, the ambiguity can be resolved based on POS and lexical features. Homograph disambiguation is described in detail in Section 14.6.
- *Noun phrase (NP) and clause detection*—Basic NP and clause information could be critical for a prosodic generation module to generate segmental durations. It also provides useful cues to introduce necessary pauses for intelligibility and naturalness. Phrase and clause structure are well covered in any parsing techniques.
- *Sentence type identification*—Sentence types (declarative, yes-no question, etc.) are critical for macro-level prosody for the sentence. Typical techniques for identifying sentence types have been covered in Section 14.3.4.

If a more sophisticated parser is available, a richer analysis can be derived. A so-called *shallow parse* is one that shows syntactic bracketing and phrase type, based on the POS of words contained in the phrases. A training corpus of shallow-parsed sentences has been created for the Linguistic Data Consortium [16]. The following example illustrates a shallow parse for sentence : "For six years, Marshall Hahn Jr. has made corporate acquisitions in the George Bush mode: kind and gentle."

```
For/IN[six/CD years/NNS],/,[T./NNP Marshall/NNP
Hahn/NNP Jr./NNP]has/VBZ made/VBN[corporate/JJ acquisi-
tions/NNS]in/IN[the/DT George/NNP Bush/NNP mode/NN]
:/:[kind/JJ]and/CC[gentle/JJ]./.
```

The POS labels used in this example are described in Chapter 2 (Table 2.14). A TTS system uses the POS labels in the parse to decide alternative pronunciations and to assign differing degrees of prosodic prominence. Additionally, the bracketing might assist in deciding where to place pauses for great intelligibility. A fuller parse would incorporate more higher-order structure, including sentence type identification, and more semantic analysis, including co-reference.

14.6. HOMOGRAPH DISAMBIGUATION

For written languages, *sense* ambiguities occur when words have different syntactic/semantic meanings. Those words with different senses are called *polysemous* words. For example, *bat* could mean either a kind of animal or the equipment to hit a baseball. Since the pronunciations for the two different senses of *bat* are identical, we are in general only concerned[6] about the other type of polysemous words that are homographs (spelled alike but vary in pronunciation), such as *bass* for a kind of fish (*/b ae s/*) or an instrument (*/b ey s/*).

Homograph variation can often be resolved on POS (grammatical) category. Examples include *object, minute, bow, bass, absent*, etc. Unfortunately, correct determination of POS (whether by a parsing system or statistical methods) is not always sufficient to resolve pronunciation alternatives. For example, simply knowing that the form *bow* is a noun does not allow us to distinguish the pronunciation appropriate for the instrument of archery from that for the front part of a boat. Even more subtle is the pronunciation of *read* in "If you *read* the book, he'll be angry." Without contextual clues, even human readers cannot resolve the pronunciation of *read* from the given sentence alone. Even though the past tense is more likely in some sense, deep semantic and/or discourse analysis would be required to resolve the tense ambiguity.

Several hundred English homographs extracted from the 1974 *Oxford Advanced Learners Dictionary* are listed in [10]. Here are some examples:

- Stress homographs: noun with front-stress vowel, verb with end-stress vowel
  "an *absent* boy" vs. "Do you choose to *absent* yourself?"
- Voicing: noun/verb or adjective/verb distinction made by voice final consonant
  "They will *abuse* him." vs. "They won't take *abuse*."
- –ate words: noun/adjective sense uses schwa, verb sense uses a full vowel
  "He will *graduate*." vs. "He is a *graduate*."
- Double stress: front-stressed before noun, end-stressed when final in phrase
  "an *overnight* bag" vs. "Are you staying *overnight*?"
- -ed adjectives with matching verb past tenses
  "He is a *learned* man." vs. "He *learned* to play piano."
- Ambiguous abbreviations: already described in Section 14.4.1
  in, am, SAT (Saturday vs. Standard Aptitude Test)
- Borrowed words from other languages—They could sometimes be distinguishable based on capitalization.
  "El Camino *Real* road in California" vs. "*real* world"
  "*polish* shoes" vs. "*Polish* accent"

[6] Sometimes, a polysemous word with the same pronunciation could have impact for prosodic generation because different semantic properties could have different accentuation effects. Therefore, a high-quality TTS system can definitely be benefited from word-sense disambiguation beyond homograph disambiguation.