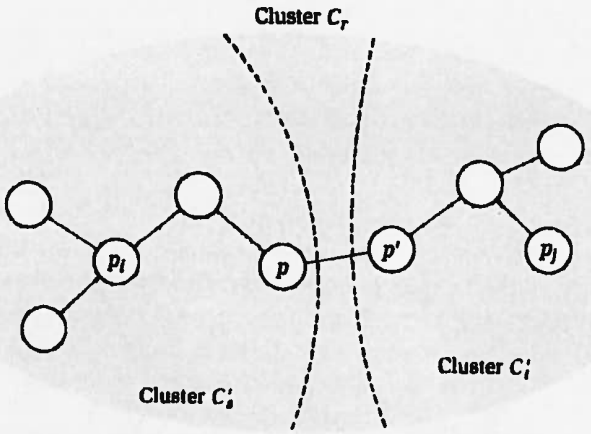

Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.

P has length at most d^*. Now, we know that $p_i \in C_s'$ but $p_j \notin C_s'$; so let p' be the first node on P that does not belong to C_s', and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \leq d^*$, since the edge (p, p') was added by Kruskal's Algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \leq d^*$. This completes the proof. ■

4.8 Huffman Codes and Data Compression

In the Shortest-Path and Minimum Spanning Tree Problems, we've seen how greedy algorithms can be used to commit to certain parts of a solution (edges in a graph, in these cases), based entirely on relatively short-sighted considerations. We now consider a problem in which this style of "committing" is carried out in an even looser sense: a greedy rule is used, essentially, to shrink the size of the problem instance, so that an equivalent smaller problem can then be solved by recursion. The greedy operation here is proved to be "safe," in the sense that solving the smaller instance still leads to an optimal solution for the original instance, but the global consequences of the initial greedy decision do not become fully apparent until the full recursion is complete.

The problem itself is one of the basic questions in the area of *data compression*, an area that forms part of the foundations for digital communication.

The Problem

Encoding Symbols Using Bits Since computers ultimately operate on sequences of *bits* (i.e., sequences consisting only of the symbols 0 and 1), one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into long strings of bits.

The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. To take a basic example, suppose we wanted to encode the 26 letters of English, plus the space (to separate words) and five punctuation characters: comma, period, question mark, exclamation point, and apostrophe. This would give us 32 symbols in total to be encoded. Now, you can form 2^b different sequences out of b bits, and so if we use 5 bits per symbol, then we can encode $2^5 = 32$ symbols—just enough for our purposes. So, for example, we could let the bit string 00000 represent a, the bit string 00001 represent b, and so forth up to 11111, which could represent the apostrophe. Note that the mapping of bit strings to symbols is arbitrary; the point is simply that five bits per symbol is sufficient. In fact, encoding schemes like ASCII work precisely this way, except that they use a larger number of bits per symbol so as to handle larger character sets, including capital letters, parentheses, and all those other special symbols you see on a typewriter or computer keyboard.

Let's think about our bare-bones example with just 32 symbols. Is there anything more we could ask for from an encoding scheme? We couldn't ask to encode each symbol using just four bits, since 2^4 is only 16—not enough for the number of symbols we have. Nevertheless, it's not clear that over large stretches of text, we really need to be spending an *average* of five bits per symbol. If we think about it, the letters in most human alphabets do not get used equally frequently. In English, for example, the letters e, t, a, o, i, and n get used much more frequently than q, j, x, and z (by more than an order of magnitude). So it's really a tremendous waste to translate them all into the same number of bits; instead we could use a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones, and hope to end up using fewer than five bits per letter when we average over a long string of typical text.

This issue of reducing the average number of bits per letter is a fundamental problem in the area of *data compression*. When large files need to be shipped across communication networks, or stored on hard disks, it's important to represent them as compactly as possible, subject to the requirement that a subsequent reader of the file should be able to correctly reconstruct it. A huge amount of research is devoted to the design of *compression algorithms*

that can take files as input and reduce their space through efficient encoding schemes.

We now describe one of the fundamental ways of formulating this issue, building up to the question of how we might construct the *optimal* way to take advantage of the nonuniform frequencies of the letters. In one sense, such an optimal solution is a very appealing answer to the problem of compressing data: it squeezes all the available gains out of nonuniformities in the frequencies. At the end of the section, we will discuss how one can make further progress in compression, taking advantage of features other than nonuniform frequencies. .

Variable-Length Encoding Schemes Before the Internet, before the digital computer, before the radio and telephone, there was the telegraph. Communicating by telegraph was a lot faster than the contemporary alternatives of hand-delivering messages by railroad or on horseback. But telegraphs were only capable of transmitting pulses down a wire, and so if you wanted to send a message, you needed a way to encode the text of your message as a sequence of pulses.

To deal with this issue, the pioneer of telegraphic communication, Samuel Morse, developed *Morse code*, translating each letter into a sequence of *dots* (short pulses) and *dashes* (long pulses). For our purposes, we can think of dots and dashes as zeros and ones, and so this is simply a mapping of symbols into bit strings, just as in ASCII. Morse understood the point that one could communicate more efficiently by encoding frequent letters with short strings, and so this is the approach he took. (He consulted local printing presses to get frequency estimates for the letters in English.) Thus, Morse code maps e to 0 (a single dot), t to 1 (a single dash), a to 01 (dot-dash), and in general maps more frequent letters to shorter bit strings.

In fact, Morse code uses such short strings for the letters that the encoding of words becomes ambiguous. For example, just using what we know about the encoding of e, t, and a, we see that the string 0101 could correspond to any of the sequences of letters *eta*, *aa*, *etet*, or *aet*. (There are other possibilities as well, involving other letters.) To deal with this ambiguity, Morse code transmissions involve short pauses between letters (so the encoding of *aa* would actually be dot-dash-pause-dot-dash-pause). This is a reasonable solution—using very short bit strings and then introducing pauses—but it means that we haven't actually encoded the letters using just 0 and 1; we've actually encoded it using a three-letter alphabet of 0, 1, and "pause." Thus, if we really needed to encode everything using only the bits 0 and 1, there would need to be some further encoding in which the pause got mapped to bits.

Prefix Codes The ambiguity problem in Morse code arises because there exist pairs of letters where the bit string that encodes one letter is a *prefix* of the bit string that encodes another. To eliminate this problem, and hence to obtain an encoding scheme that has a well-defined interpretation for every sequence of bits, it is enough to map letters to bit strings in such a way that no encoding is a prefix of any other.

Concretely, we say that a *prefix code* for a set S of letters is a function γ that maps each letter $x \in S$ to some sequence of zeros and ones, in such a way that for distinct $x, y \in S$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Now suppose we have a text consisting of a sequence of letters $x_1x_2x_3 \cdots x_n$. We can convert this to a sequence of bits by simply encoding each letter as a bit sequence using γ and then concatenating all these bit sequences together: $\gamma(x_1)\gamma(x_2) \cdots \gamma(x_n)$. If we then hand this message to a recipient who knows the function γ, they will be able to reconstruct the text according to the following rule.

- Scan the bit sequence from left to right.
- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.
- Now delete the corresponding set of bits from the front of the message and iterate.

In this way, the recipient can produce the correct set of letters without our having to resort to artificial devices like pauses to separate the letters.

For example, suppose we are trying to encode the set of five letters $S = \{a, b, c, d, e\}$. The encoding γ_1 specified by

$$\gamma_1(a) = 11$$
$$\gamma_1(b) = 01$$
$$\gamma_1(c) = 001$$
$$\gamma_1(d) = 10$$
$$\gamma_1(e) = 000$$

is a prefix code, since we can check that no encoding is a prefix of any other. Now, for example, the string *cecab* would be encoded as 0010000011101. A recipient of this message, knowing γ_1, would begin reading from left to right. Neither 0 nor 00 encodes a letter, but 001 does, so the recipient concludes that the first letter is c. This is a safe decision, since no longer sequence of bits beginning with 001 could encode a different letter. The recipient now iterates

on the rest of the message, 0000011101; next they will conclude that the second letter is e, encoded as 000.

Optimal Prefix Codes We've been doing all this because some letters are more frequent than others, and we want to take advantage of the fact that more frequent letters can have shorter encodings. To make this objective precise, we now introduce some notation to express the frequencies of letters.

Suppose that for each letter $x \in S$, there is a frequency f_x, representing the fraction of letters in the text that are equal to x. In other words, assuming there are n letters total, nf_x of these letters are equal to x. We notice that the frequencies sum to 1; that is, $\sum_{x \in S} f_x = 1$.

Now, if we use a prefix code γ to encode the given text, what is the total length of our encoding? This is simply the sum, over all letters $x \in S$, of the number of times x occurs times the length of the bit string $\gamma(x)$ used to encode x. Using $|\gamma(x)|$ to denote the length $\gamma(x)$, we can write this as

$$\text{encoding length} = \sum_{x \in S} nf_x|\gamma(x)| = n \sum_{x \in S} f_x|\gamma(x)|.$$

Dropping the leading coefficient of n from the final expression gives us $\sum_{x \in S} f_x|\gamma(x)|$, the *average* number of bits required per letter. We denote this quantity by $\text{ABL}(\gamma)$.

To continue the earlier example, suppose we have a text with the letters $S = \{a, b, c, d, e\}$, and their frequencies are as follows:

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

Then the average number of bits per letter using the prefix code γ_1 defined previously is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25.$$

It is interesting to compare this to the average number of bits per letter using a fixed-length encoding. (Note that a fixed-length encoding is a prefix code: if all letters have encodings of the same length, then clearly no encoding can be a prefix of any other.) With a set S of five letters, we would need three bits per letter for a fixed-length encoding, since two bits could only encode four letters. Thus, using the code γ_1 reduces the bits per letter from 3 to 2.25, a savings of 25 percent.

And, in fact, γ_1 is not the best we can do in this example. Consider the prefix code γ_2 given by