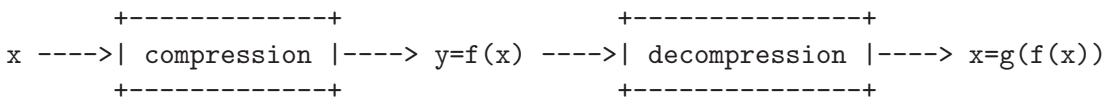
CMPS340 File Processing
Notes on Coding and Data Compression
R. McCloskey

Still under construction...

Coding has to do with representing data or information using a string of symbols over some alphabet.[1] Often the term is used in referring to the translation of data represented by a string over one alphabet into a string over a different alphabet. The widely used ASCII ("American Standard Code for Information Interchange"), for example, associates a bit string of length seven to each of the 2^7 members of what could be called the "ASCII alphabet", which contains just about every character commonly used in the English language, including the roman letters (both lower and upper case), decimal digits, and common punctuation symbols, as well as some **control characters** (e.g., line feed, carriage return, backspace).[2] Given that the character is a basic form of data and that electronic computers represent data in the form of bit strings, it is only natural that some such coding standard be adopted.

Data compression refers to a procedure by which, for some alphabet A, one derives a string $y \in A^*$ from a given string $x \in A^*$ such that y is (we hope) shorter than x but such that x is recoverable from y by an inverse procedure known, naturally, as **decompression**. This is depicted in the nearby figure, where we have denoted by f the function computed by the compression procedure and by g its inverse (the function computed by the decompression procedure). Note that it is vital for f to have an inverse, which is to say that f must be *injective* (or *one-to-one*, if you prefer), meaning that, for every pair of strings x_0 and x_1, $x_0 \neq x_1$ implies $f(x_0) \neq f(x_1)$ (or, equivalently, $f(x_0) = f(x_1)$ implies $x_0 = x_1$). Suppose, to the contrary, that x_0 and x_1 were distinct strings satisfying $f(x_0) = y = f(x_1)$. Then, given y, the decompression procedure would have no way of determining which of x_0 or x_1 should be its output.



Among the benefits of data compression are reduced use of storage space and faster transmission of data over computer networks.

To measure the effectiveness of compression function f when applied to a particular string x, we use the **compression ratio** $\frac{|f(x)|}{|x|}$, where $|z|$ denotes the length of z. The smaller the ratio,

---

[1] An *alphabet* is simply a set of symbols, such as $\{a, b, c\}$. If x is a string composed of symbols drawn from alphabet A, we say that x is a string *over* A. The set of all such strings is denoted by A^*; hence we can also express this by $x \in A^*$.

[2] There are various extensions to ASCII that encode each character using eight bits rather than only seven. This allows the alphabet to be expanded to 2^8 characters so as to include some Greek letters (e.g., α, β) and mathematical symbols (e.g., $\leq$, $\div$, $\surd$).

the more compression that was achieved. (One could just as well use the reciprocal $\frac{|x|}{|f(x)|}$, in which case a larger value would indicate better compression.)

Interestingly, it is not possible to devise an injective function f such that $|f(x)| < |x|$ for every string x. In fact, we can claim something even stronger:

Theorem: Let A be an alphabet with at least two members, and let $f : A^* \to A^*$ be an injective function. Then, for every $n > 0$, there exists a string $x \in A^*$ of length n such that $|f(x)| \geq |x|$.

Proof: To prove this, it suffices to show that the number of strings of length n is greater than the number of strings of length less than n, for then the Pigeonhole Principle tells us that, if f mapped every string of length n into a shorter string, it would have to map two strings of length n to the *same* shorter string, contradicting our assumption that f is injective. Now, the number of strings over A of length n is m^n, where $m = |A|$. Let $\#$ be the function such that $\#(n)$ is the number of distinct strings over A of length less than n. Then (using our assumption that $m > 1$), we have

$$\#(n) = \sum_{0 \leq i < n} m^i = \frac{m^n - 1}{m - 1} < m^n$$

That is, the number of strings of length n exceeds the number of all shorter strings, as required. *End of proof.*

More generally, compression schemes come in two main varieties. What we have just described is called **lossless** compression. The other kind is **lossy** compression, so called because the result $g(f(x))$ of decompressing $f(x)$ is not guaranteed to be x itself, but only something "resembling" x. (Hence, something is "lost".) Given the less strict requirements, it should not be surprising that lossy compression techniques typically are able to achieve a better rate of compression than lossless ones.

For some kinds of data (source code, machine code, most text), it is absolutely vital that the original string be recoverable from its compressed form. In these cases, lossy compression is intolerable. For data representing images and sounds, however, lossy compression is acceptable so long as the picture (or audio, or whatever) represented by $g(f(x))$ is sufficiently close to that represented by x. (Indeed, even if x and $g(f(x))$ differ significantly as strings, human vision and hearing may not be able to perceive any difference between the two when they are rendered as images or audio.)

Run-length encoding A rudimentary form of data compression is run-length encoding, in which *runs* of repeated values are encoded not by literally repeating the value but rather by identifying the repeated value and indicating how many times it repeats. This approach is effective only on data in which repeated values are very common, of course.

One possible scheme we could employ for doing (what one might call) byte-based run-length encoding is to follow the rule that, when there is a run of bytes having the same value, we encode it using three bytes:

(1) 00, which signals that what comes next is the description of a run,[3]
(2) a one-byte code (using standard binary representation) for the length of the run, and
(3) the byte value that occurs in the run

For example, a run of $4A$'s of length six would be encoded by 00 06 $4A$[4]. Because the encoding of a run takes three bytes, there is no benefit in encoding a run of length less than four in the manner prescribed above. (Indeed, for a run of length two (or one!), extra bytes would be used in doing so.)

There is one issue that needs to be addressed: How do we encode a single occurrence of byte value 00? If we simply include it in the compressed version of the file, the decompressor will interpret it as the first byte in the three-byte code for some run. For example if the original data contained, say, ... 37 00 13 $D2$ AF ..., the three bytes in the middle would be encoded (in the compressed version of the data) exactly as is. But then the decompressor would interpret this three-byte sequence as an encoding of a run of $D2$'s of length 19 (13 in hexadecimal is 16+3 in decimal, of course).

One way to solve the problem is to stipulate that a single occurrence of 00 should be encoded by 00 01 00, which describes a run of 00's of length one! It seems a pity to have to use three bytes to encode a single byte, however, especially when our goal is to achieve compression! A slightly better idea is to encode 00 by 00 00, which uses only two bytes rather than three. But it is not immediately clear why this should work, because, for example, the byte sequence 00 13 would end up being encoded as 00 00 13, which the decompressor would naturally interpret as a run of 13's of length zero! Ah, but the point is that we would never encode a run of length zero in this way! Hence, the decompressor is designed so that, if it encounters 00 immediately followed by another 00, it translates these two bytes to a single 00.

Note that, following this convention for encoding 00, a run of two (or more) 00's should be encoded using the three-byte scheme, as it saves space. (In contrast, a run of any other byte value must be of length at least four to be encoded using the three-byte scheme, because no space is saved on shorter runs.)

To illustrate the scheme just described, suppose that the first 17 bytes of data to be compressed are

47 32 $4A$ $4A$ $4A$ $4A$ $4A$ $4A$ $C2$ $C2$ $C2$ 63 00 26 00 00 $9D$

Compressing it, we would get (the 16 bytes)

47 32 00 06 $4A$ $C2$ $C2$ $C2$ 63 00 00 26 00 02 00 $9D$

Very little compression was achieved, because there really wasn't much repetition to exploit.

Symbolwise coding

[3]By convention, a byte value is expressed as a two-digit hexadecimal (i.e., base 16) numeral. This is quite natural, because each hexadecimal digit corresponds to four binary digits (due to the fact that $2^4 = 16$!), and a byte is eight binary digits. Note that the sixteen digits used in base 16 are 0 through 9 and A through F, the latter six corresponding to the values 10 through 16.

[4]The space between adjacent byte values is there simply to make reading easier.

The concept of (symbol-wise) coding can be described as follows: You are given a *source text* x over the *source alphabet* A. That is, you are given $x \in A^*$. The goal is to produce an encoding y over *code alphabet* B by replacing each symbol in x by a string of symbols from B.

The **Morse Code** is a **static** symbolwise coding scheme in which $A = \{\texttt{A}, \ldots, \texttt{Z}, \texttt{0}, \ldots, \texttt{9}, ?, :, \ldots\}$ and $B = \{-, .\}$. Each member of A has a corresponding **codeword** composed of dots and dashes, as depicted in the figure.

Symbol	Codeword	Symbol	Codeword	Symbol	Codeword
A	.-	M	--	Y	-.--
B	-...	N	-.	Z	--..
C	-.-.	O	---	1	.----
D	-..	P	.--.	2	..---
E	.	Q	--.-	3	...--
F	..-.	R	.-.	4	....-
G	--.	S	...	5	.....
H	....	T	-	6	-....
I	..	U	..-	7	--...
J	.---	V	...-	8	---..
K	-.-	W	.--	9	----.
L	.-..	X	-..-	0	-----

Figure 1: Morse Code

The Morse Code illustrates some important concepts relevant to coding and compression.

Notice that the codewords for `E` and `T` —the two most frequently occurring letters in English text— have length one, whereas those for `X`, `Y`, and `Z` —among the most rarely occurring letters— have length four. Is this just a coincidence? **NO!** By assigning shorter codewords to letters that tend to occur more frequently, we tend to shorten the encoding of the source text. This was one of Morse's goals.

As a simple example to illustrate this, suppose that the code alphabet is $B = \{0, 1\}$, the source alphabet is $A = \{a, b, c, d\}$, and, in the source text, a occurs 27 times, b occurs 12 times, c occurs 8 times, and d occurs 3 times (for a total of 50 occurrences of symbols). A fixed-length coding scheme assigns a distinct codeword of length $\lceil \log_{|B|} |A| \rceil$ to each symbol in A. In this example, with $|A| = 4$ and $|B| = 2$, that comes out to two, of course. For example, we might use the following fixed-length code:

$$a \to 00 \quad b \to 01 \quad c \to 10 \quad d \to 11$$

Because two bits will be used to encode each of the 50 occurrences of symbols, the length of the encoded text will be 100 bits.

More generally, the length, in bits, of the encoded text will be

$$\sum_{v \in A} (\text{len}_v \cdot \text{freq}_v)$$

where len_v is the length of v's codeword and freq_v is the frequency with which v occurs in the source text. For this example, we get

$$\begin{aligned}
\sum_{v \in A} \text{len}_v \cdot \text{freq}_v &= (\text{len}_a \cdot \text{freq}_a) + (\text{len}_b \cdot \text{freq}_b) + (\text{len}_c \cdot \text{freq}_c) + (\text{len}_d \cdot \text{freq}_d) && (1)\\
&= (2 \cdot 27) + (2 \cdot 12) + (2 \cdot 8) + (2 \cdot 3) && (2)\\
&= 2 \cdot (27 + 12 + 8 + 3) && (3)\\
&= 2 \cdot 50 && (4)\\
&= 100 && (5)
\end{aligned}$$

which is consistent with our earlier calculation.

Now consider this alternative (variable-length) coding scheme:

$$a \rightarrow 0 \quad b \rightarrow 10 \quad c \rightarrow 110 \quad d \rightarrow 111$$

Under this scheme, the length of the encoded text will be

$$\begin{aligned}
\sum_{v \in A} (\text{len}_v \cdot \text{freq}_v) &= (\text{len}_a \cdot \text{freq}_a) + (\text{len}_b \cdot \text{freq}_b) + (\text{len}_c \cdot \text{freq}_c) + (\text{len}_d \cdot \text{freq}_d) && (6)\\
&= (1 \cdot 27) + (2 \cdot 12) + (3 \cdot 8) + (3 \cdot 3) && (7)\\
&= 27 + 24 + 24 + 9 && (8)\\
&= 84 && (9)
\end{aligned}$$

Hence, 84 bits suffice to encode the source text under this alternative coding scheme, compared to 100 bits using the fixed-length code. This yields a compression ratio of $\frac{84}{100}$.

Morse Code is a **static** code, meaning that the mapping from the source alphabet to the codewords doesn't change, regardless of the source text being encoded. In contrast, in the just-completed example we implicitly suggested the use of **semi-static** coding, in which the mapping from the source alphabet to codewords is constructed based upon the number of occurrences of each symbol in the source text.

In **dynamic** coding, the mapping itself is subject to being changed as we are translating the source text. Hence, occurrences of a near the beginning of the source text, where a occurs frequently, might be encoded by 0 but occurrences of a near the end, where it occurs much less often, might be encoded by 1011. Typically, a dynamic coding scheme begins with a default symbol-to-codeword mapping and then adjusts it periodically according to how frequently each symbol has appeared so far (or how frequently it has appeared "recently"). Hence, one advantage of dynamic coding over semi-static coding is that the latter must make two passes over the source text (the first for the purpose of gathering whatever information (e.g., symbol

frequencies) is used in constructing the symbol-to-codeword mapping, and the second for applying that mapping to the source text to produce its coded version) whereas the former need only make one pass over the source text.

Unique Decipherability

Concatenation, denoted by $\cdot$, is the operation by which strings are formed from shorter strings. Consider, for example, the alphabet $\{a, b\}$. Each of a and b can be viewed as a string of length one. Using concatenation, we can generate the string $a \cdot b$. Applying it again, we can generate $(a \cdot b) \cdot a$. By definition, concatenation is associative, so we can omit the parentheses and write this as $a \cdot b \cdot a$. But the $\cdot$'s are cumbersome, so (as in algebra, where we often omit the symbol for multiplication) we prefer to write this as aba. In general, then, unless we wish to emphasize that a string is being viewed as a product of particular factors (as in $aba \cdot ba \cdot aba$), we write it without $\cdot$'s.

We are now ready to define the notion of **unique decipherability**: A set of codewords is said to be uniquely decipherable (or **u.d.**) if any string formed by concatenating codewords together can be factored into codewords in only one way. Or, to put it more precisely, to say that C is u.d. is to say that if

$$u_1 u_2 \cdots u_m = v_1 v_2 \cdots v_n$$

where each u_i and each v_i is a member of C, then necessarily $m = n$ and, for each k satisfying $1 \leq k \leq n$, $u_k = v_k$.

For example, the set of codewords $C = \{01, 10, 010\}$ is *not* u.d. because the string 01010 can be factored as either $01 \cdot 010$ or $010 \cdot 10$. (We call each of these a C-factorization, meaning that each factor is a member of C. We call a pair of distinct factorizations of the same string a *disagreeing pair* of factorizations.) If, say, we had a coding scheme in which the symbols a, b, and c from the source alphabet were mapped to the codewords 01, 10, and 010, respectively, then in decoding 01010 we could (correctly) obtain either ac (corresponding to the factorization $01 \cdot 010$) or cb (corresponding to the factorization $010 \cdot 10$). In other words, the encoding 01010 is ambiguous! For most applications, this would be unacceptable.

It turns out that Morse Code is not uniquely decipherable. For example, `.--` encodes three different strings: AT, EM, and ETT. If we insert a third symbol, say `/`, into the code alphabet and we append it to the end (or the beginning) of each codeword, we obtain a code that is u.d. Using this modified scheme, AT, EM, and ETT would be encoded as `.-/-/`, `./--/`, and `./-/-/`, respectively.

In about 1950, Sardinas and Patterson developed an algorithm that, given a finite set of codewords, determines whether or not that set is uniquely decipherable. Below we give an algorithm that augments theirs in that it not only answers the question of unique decipherability but also —in case the given set C of codewords is not u.d.— yields information sufficient to construct all possible disagreeing pairs of C-factorizations. Before describing the algorithm, we need to introduce a few more concepts related to strings.

Let u and x be strings. If there exists a string v such that $x = uv$, we say that u is a *prefix* of x, which is denoted by $u \preceq x$. If, in addition, $u \neq x$, we say that u is a *proper prefix* of x, which is denoted by $u \prec x$. For example, $u = abb$ is a proper prefix of $x = abbba$, as choosing

$v = ba$ gives us $x = uv$. (Every string is a prefix of itself, but not a proper prefix of itself.) Of course, there are the analogous concepts *suffix* and *proper suffix*.

If $u \preceq x$, then $u\backslash x$ (sometimes written $u^{-1}x$) is defined to be that string v such that $x = uv$. For example, $abb\backslash abbba = ba$ and $abb\backslash abb = \lambda$. (The string of length zero, called the *empty string* is denoted by λ.) If u is not a prefix of x, $u\backslash x$ is undefined. The $\backslash$ operator is called *left quotient*.

The augmented Sardinas-Patterson algorithm (see Figure 4) takes as input a set of codewords $C = \{w_1, w_2, \cdots, w_n\}$ and constructs from it a directed graph $SP(C)$, the Sardinas-Patterson graph for C. Each vertex in $SP(C)$ is identified with either a member of C or some proper suffix thereof, and every edge is labeled by a member of C. The program variable V represents the set of vertices in the graph, E represents the set of edges, and $Q \subseteq V$ is the set of vertices for which it remains to compute outgoing edges.

In Figure 2 you will find $SP(C)$ for

$$C = \{1, 00, 01, 010, 110, 0010\}$$

Vertices corresponding to members of C are indicated by two concentric circles. Call these *terminal* vertices. One can prove that, for any finite set C of codewords, C is uniquely decipherable if and only if there is no non-empty path in $SP(C)$ that begins and ends in terminal vertices. Indeed, if such paths do exist, each one corresponds to a *prime* disagreeing pair of C-factorizations. (By prime we mean that the pair of factorizations is such that there is no way to "peel" off one or more factors from either end and still end up with a disagreeing pair of factorizations.) Furthermore, by traversing that path —and making use of the labels on the terminal vertices and the edges along the way— we can construct the corresponding prime disagreeing pair of factorizations.

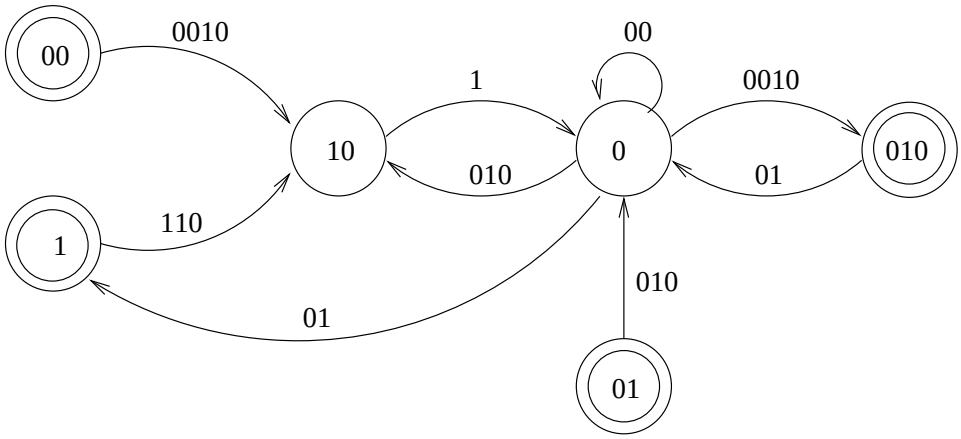


Figure 2: Sardinas-Patterson Graph for $\{1, 00, 01, 010, 110, 0010\}$

We now provide instructions for doing this: Let $(v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{k-1}, l_{k-1}, v_k)$ be a path in $SP(C)$, where $k > 0$ and $v_0, v_k \in C$. Then use the algorithm in Figure 3 to obtain a prime disagreeing pair (f, g) of C-factorizations.

$f := v_0$;
$g := l_0$
$i := 1$
do while $i \neq k$
  extend shorter of f and g by "$\cdot$" l_i
  $i := i + 1$
od
extend shorter of f and g by "$\cdot$" v_k

Figure 3: Using a Path in $SP(C)$ to Construct a Prime Disagreeing Pair of C-Factorizations

As an example of constructing a prime disagreeing pair of factorizations, consider the Sardinas-Patterson graph in Figure 2. Take the path

$$(00, 0010, 10), (10, 1, 0), (0, 0010, 010)$$

which begins in (terminal) vertex 00 and ends in (terminal) vertex 010. Following the algorithm in Figure 3, we get the disagreeing pair of factorizations

$$00 \cdot 1 \cdot 0010$$

$$0010 \cdot 010$$

Or we could take a circuitous route, as by the path

$$(01, 010, 0), (0, 010, 10), (10, 1, 0), (0, 00, 0), (0, 0010, 010), (010, 01, 0), (0, 0010, 010)$$

This give rise to the prime pair of disagreeing factorizations

$$01 \cdot 010 \cdot 0010 \cdot 010$$

$$010 \cdot 1 \cdot 00 \cdot 01 \cdot 0010$$

Deciphering Delay:

For a coding scheme to function properly, at the very least it must be u.d. But, for the sake of efficiency, we would like to demand even more from it.

Consider the u.d. set of codewords $\{01, 10, 011\}$ and consider the bit string $y = 01(10)^n$, where n is some positive integer. The only factorization is

$$01 \ \cdot \ 10 \ \cdot \ 10 \ \cdot \ 10 \ \cdot \ \ldots \ \cdot \ 10$$

On the other hand, the string $x = y1 = 01(10)^n1 = 011(01)^n$ has as its unique factorization

$$011 \ \cdot \ 01 \ \cdot \ 01 \ \cdot \ \ldots \ \cdot \ 01$$

// Input is the set of codewords $C = \{w_1, w_2, \cdots, w_n\}$

isUD := $true$ //final value answers the question: Is C uniquely decipherable?
$V, E, Q := \emptyset, \emptyset, \emptyset$ // Sets of vertices, edges, and the "queue" all begin empty

do for each i in 1..n
  do for each j in 1..n
    if $w_i \prec w_j$ **then**
      $V := V + \{w_i\}$
      $y := w_i \backslash w_j$
      $addEdge(w_i, w_j, y)$
    fi
  od
od

do while $Q \neq \emptyset \wedge$ isUD // omit 2nd conjunct to build complete graph
  $v := Q.takeOne()$ // remove an element from Q and assign it to v
  do for each i in 1..n
    if $v \prec w_i$ **then**
      $addEdge(v, w_i, v \backslash w_i)$
    elsif $w_i \prec v$ **then**
      $addEdge(v, w_i, w_i \backslash v)$
    else
      // do nothing, as neither v nor w_i is a proper prefix of the other
    fi
  od
od

function addEdge(x, label, y)
  $E := E + \{(x, label, y)\}$
  if $y \notin V$ **then**
    $V := V + \{y\}$
    $Q := Q + \{y\}$
  fi
  if $y \in C$ **then**
    isUD := $false$
  fi

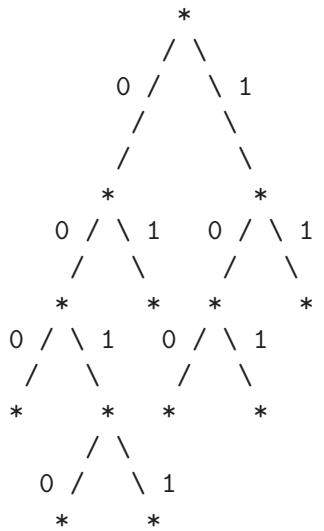
Figure 4: Algorithm for Constructing a Sardinas-Patterson Graph

Hence, in decoding y (or x, for that matter), there is no way to tell which of the two codewords 01 or 011 is the first factor until we have reached the end of the string (and thereby determined whether it ends with 0 or with 1)! (We assume that a string to be decoded is scanned from left to right.) Given that n can be chosen to be arbitrarily large, it follows that this set of codewords has **unbounded deciphering delay**. In a context in which a large quantity of coded data is being sent to us over a network and we want to decode it "on the fly", this is BAD!

A less drastic example of deciphering delay arises from the u.d. set of codewords $\{0, 01\}$. In attempting to decode a message formed from this set of codewords, we must follow the rule that an occurrence of 0 is a factor if and only if it is not immediately followed by 1. Hence, this set of codewords has a deciphering delay of one, meaning that, in the worst case, we must look one symbol beyond the end of a potential factor before we can determine whether or not it is, indeed, a factor. Obviously, a deciphering delay of one is far preferable to an unbounded delay, as we saw with the previous example. A deciphering delay of zero (i.e., *instantaneous decipherability*) is the best we could possibly hope for, of course.

Huffman Coding/Compression

Consider a **full** binary tree, meaning one in which every interior (i.e., non-leaf) node has two children. For each interior node, label the edges going to its children 0 and 1, respectively. Consider the string "spelled out" by the labels on the edges along the path from the root to a given node. We refer to this string as the node's label. Clearly, no two nodes in the tree have the same label. Moreover, for any nodes p and q, the label of p is a proper prefix of the label of q if and only if p is a proper ancestor of q. Hence, if we take the labels of all the nodes having no proper descendants (i.e., the leaves), we end up with a **prefix-free** set of binary codewords![5] The set of labels of the leaves in the tree pictured nearby is $\{000, 0010, 0011, 01, 100, 101, 11\}$.



[5] A set of strings is said to be prefix-free if none of them is a proper prefix of any of them.

It should be fairly obvious that any prefix-free set of codewords is not only u.d. (which we insist upon absolutely) but also has deciphering delay zero, which is (obviously) optimal. All else being equal, then, we would prefer to use prefix-free codeword sets to ones that are not prefix-free. But is all else necessarily equal? Perhaps not. After all, it is conceivable that constructing a prefix-free codeword set is much more expensive (in terms of computing resources—time and space) than constructing one that is simply u.d. (but possibly not prefix-free). Also, it is conceivable that, in some cases (e.g., for a particular set of symbol frequencies), one can achieve better compression (i.e., a lower compression ratio) by using a non-prefix-free codeword set than by using *any* prefix-free codeword set.

Remarkably, it has been proved that, for any u.d. codeword set, one can find a prefix-free codeword set having exactly the same number of members with exactly the same lengths! Hence, insisting upon a prefix-free codeword set leads to no sacrifice in terms of compression ratio. As for the expense of constructing a prefix-free codeword set, Huffman's algorithm (which constructs a full binary tree, as illustrated nearby, in order to obtain a prefix-free set of codewords) shows that, at least when the source alphabet is finite, this can be done quite efficiently.