An Algorithm for Decoding American Morse

Les Kerr

2 July 2006

American Morse is much more difficult to decode by computer than International Morse, mostly because of its internally spaced letters (C, O, R, Y, and Z). Consider, for example, the waveform in Figure 1. In International Morse, clearly, the word is SEEN.

Figure 1. SEEN, SON, or ZEN?

In American Morse, however, the situation is much more complicated. The *dididit dit* at the beginning could be a Z – making the word ZEN. Or the *dit dit* in the middle could be an O – making it SON. The timing differences among these three words can be quite subtle, and variations in sending style – even by the same operator within the same transmission – can make it almost impossible to correctly decode the word by using *a priori* timing thresholds.

The Morse KOB program's code reader uses a decoding algorithm that is relatively straightforward, and yet even in the face of ambiguities such as this it performs surprisingly well.

Sorting out the dots and dashes

The code reader's first task is to figure out which of the marks are dots and which are dashes. This is particularly easy when the code is being sent on a bug, because then the timing of the dots is so regular.

The nominal length of a dot, in milliseconds, is determined by the Code Speed setting[1] according to the formula

$$DotLength = 1200 / WPM$$

Since a dash is nominally three dots in length, a threshold of two dot lengths would seem to be the right choice for distinguishing dots from dashes. In practice, however, the length of the dots generated by a bug is very predictable, while the manually generated dashes are often considerably shorter than the standard 3 dot lengths. Therefore, a dot-dash threshold a bit less than 2 is a better choice.

Element space *vs*. internal or character space

The second task for the code reader is to decide which of the spaces are interelement spaces within a character, and which are longer spaces: an internal "Morse" space, a

[1] Early versions of the KOB code reader algorithm were more sophisticated and attempted to determine the speed of the incoming code dynamically by observing the lengths of the marks and spaces. In actual practice, however, I was surprised to find that I got better results by setting the approximate code speed manually and leaving it there. The setting is not critical, and rarely needs to be changed.

character space, or a word space. Since an internal space will generally be at least 2.5 dot units in length, a threshold of just less than 2 is appropriate. Since this is so similar to the dot-dash threshold, the algorithm uses the same parameter for both purposes.

Applying these first two tasks to the sample waveform in Figure 1, the algorithm produces the following:

[· · ·] <150> [·] <200> [·] <150> [— ·]

where the strings within square brackets are the dots and dashes of the "characters", and the numbers in angle brackets are the lengths of the spaces between the "characters". Of course, some of these spaces could be internal "Morse" spaces, in which case two of the strings in square brackets would in fact represent a single character. Resolving this question is the subject of the next section.

Internal space *vs*. character space

Correctly identifying the internally spaced letters is a problem unique to American Morse, and it's by far the most challenging aspect of the code reader. The key to the algorithm is to judge the length of a space based on its immediate context: specifically, the relative length of the immediately preceding space. Simply stated, if a given space is *equal* to or *longer* than the preceding space, then it's assumed to be a character (or word) space.[2] On the other hand, if it's *shorter* than the preceding space, then it *may* be an internal space, in which case the algorithm defers judgment until the following space can be examined.

Going back to our example in the previous section, here's how the algorithm would work. First we note that the 150 ms space between the [· · ·] and the [·] is *shorter* than the previous space (the long space preceding the sample waveform). Therefore, it *may* be an internal space, but we need to wait until we can compare it with the following space before making a decision.

Since the 150 ms space is, in fact, *shorter* than the following 200 ms space, we can assume that it's an internal space and merge the [· · ·] and [·] strings into a single string with an internal space: [· · · ·] . A search of the code definition table for American Morse tells us that this string corresponds to the letter Z.

Now we can move on to the second 150 ms space, the one between the [·] and [— ·] strings. Since it's *shorter* than the preceding 200 ms space, once again it's a candidate for being an internal space. We find that it's also shorter than the following space (the long space following the sample waveform), so we merge the two strings into a single string with an internal space: [· — ·] . This time, however, we find no such string in the code definition table. When this happens, the algorithm concludes that the 150 ms space has to be a word space, not an internal space, so it decodes the [·] and [— ·] strings separately as E and N.

In this way, the waveform in Figure 1 is decoded as the word ZEN. Very small differences in the lengths of the spaces could have yielded SEEN or SON instead. Such is the nature of American Morse.

[2] Actually, "equal to" means *approximately* equal to. The allowable tolerance, roughly plus or minus 15 percent, is determined by a parameter in the code reader algorithm.

What about word spaces?

Attempting to identify the word spaces in hand-sent code can be an exercise in frustration. When I tried to tackle this problem head on in early versions of the KOB program, the code reader would frequently break words apart or run them together. The resulting text would look something like this:

```
STR O NG BR EEZE , 2 5 TO 31 MPH, L ARGE BRANC HES INMOTION , TE
L EGRAPH WI RES WHIST LE, UMBRE LLASUSED WITH DIFFICU LTY
```

The current version of the code reader finesses the whole issue by not attempting to identify word spaces at all. Instead, it uses a proportional font to display the text and it inserts an amount of space between the printed characters that corresponds to the actual duration of the space. When decoding perfectly sent code, the code reader inserts a single space between characters and three spaces between words. With hand-sent code, there can be from zero to five spaces.[3]

Using this less ambitious algorithm, the above example might look like this:

```
STR O NG BR EEZE, 2 5 TO 31 MPH, L ARGE BRANC HES INMOTION,
TE L EGRAPH WI RES WHIST LE, UMBRE LLAS USED WITH DIFFICU LTY
```

It's surprising how much more readable the output from the simpler algorithm is. The secret to its success is presenting relevant information, namely the actual space between the letters, to the human eye and letting the reader's brain do the pattern matching – something it can do quite effortlessly.

Appendix A: Code reader parameters

Name	Value	Function
DotVsDash	1.9	Threshold for separating dots from dashes (in dot units)
DashVsLongDash	5	Threshold for separating dashes from long dashes[4]
Tolerance	15%	Tolerance for two spaces to compare as "equal"

[3] The algorithm uses *nonbreaking* spaces for the first two spaces between characters, so word wrap takes place only where there are three or more spaces. This prevents the accidental splitting of a word at the end of a line.

[4] Early versions of the KOB program decoded L and Ø as distinct characters, depending on the length of the dash. There's so much variation in the way different operators send L and Ø, however, that the program often misinterpreted one for the other. I found it less confusing to have an L instead of a Ø in the middle of a number than to have a Ø instead of an L in the middle of a word, so I changed the program to always display an L for any dash longer than a T.