Major Technique: Compression

Version 14/06/00 16:34 - Charles Weir 7

How can you fit a quart of data into a pint pot of memory?

- The *memory requirements* of the code and data appear greater than the memory available, whether primary memory, secondary storage, read-only memory or some combination of these
- You cannot reduce the functionality and omit some of the data or code
- You need to transmit information across a communications link as *quickly* as possible.
- You cannot choose **SMALL DATA STRUCTURES** to reduce the memory requirements further

Sometimes you just don't have enough memory to go around. The most usual problem is that you need to store more data than the space available, but sometimes the executable code can be too large. You can often choose suitable **DATA STRUCTURES** to ensure that the right amount of memory is allocated to store the data; you can also use **SECONDARY STORAGE** and **READ-ONLY STORAGE** move the data out of RAM. These techniques have one important limitation, however: they don't reduce the total amount of storage, of all kinds, needed to support the whole system.

For example, the Strap-It-On wrist-mounted PC needs to store the data for the documents the user is working on. It also needs sound files recorded by the internal microphone, data traces from optional body well-being monitors, and a large amount of executable code downloaded by the user to support "optional applications" (typically Tetris, Doom and Hunt-the-Wumpus, but sometimes work-related programs as well!). This information can certainly exceed the capacity of the Strap-It-On's primary memory and secondary storage combined. How can we improve the Strap-It-On's usability without forcing every user to carry around the optional 2 Gb disk back-pack?

No matter how much memory such a system may have, you will always find users who need more. Extra storage is expensive, so you should use what you have as effectively as possible.

Therefore: *Use a compressed representation to reduce the memory required.*

Store the information in a compressed form and decompress it when you need to access it. There are a wide variety of compression algorithms and approaches you can choose from, each with different space and time trade-offs.

So, for example, the Strap-It-On PC stores its voice sound files using GSM compression; its music uses MP3; its data traces use **DIFFERENCE COMPRESSION**; its databases use **TABLE COMPRESSION**; and its documents are stored using GZIP. The device drivers for Strap-It-On's secondary storage choose the appropriate **ADAPTIVE COMPRESSION** technique based on the file type, ensuring all files are stored in a compressed form.

Consequences

The *memory requirements* of your system decrease because compressed code and data need less space than uncompressed code or data. Some forms of *time performance* may also improve – for example, reading from slow secondary storage devices or over a network.

However: Compressed information is often more difficult to process from within the program. Some compression techniques prevent random access to the compressed information. You may have to decompress an entire data stream to access any part of it – requiring enough *main*

memory to store all the decompressed information, in addition to the *memory* needed for the decompression itself.

The program has to provide compression and decompression support, making it *more complex to maintain*, requiring a fair amount of *programmer effort* to implement, increasing the *testing cost* of the program and reducing the *realtime responsiveness*.

The compression process also *takes time* and *extra temporary memory* increasing the possibilities for failure; compression can also increase a program's *power consumption*. In some cases – program code, resource file data, and information received via telecommunications – the compression cost may be paid once by large powerful machines better able to handle it. The amount of memory required to store a given amount of data becomes *less predictable*, because it depends upon well the data can be compressed.

❖ ❖ ❖

Implementation

The key idea behind compression is that most data contains a large amount of *redundancy* — information that is not strictly required [Bell, Cleary, Whitten, 1990]. The following sections explore several types of redundancy, and discuss compression techniques to exploit each type.

1. Mechanical Redundancy

Consider the ASCII character set. ASCII defines around 100 printable characters, yet most text formats use eight, sixteen, or even thirty-two bits to store characters for processing on modern processors. You can store ASCII text using just seven bits per character; this would reduce memory used at a cost of increased processing time, because most processors handle eight or thirty-two bit quantities much more easily than seven bit quantities. Thus, 1 bit in a single byte encoding, or 9 bits in a sixteen bit UNICODE encoding are redundant. This kind of redundancy is called mechanical redundancy.

For text compression, the amount of compression is usually expressed by the number of (compressed) bits required per character in a larger text. For example, storing ASCII characters in seven bit-bytes would give a compression of 7 bits per character. For other forms of data we talk about the *compression ratio* – the compressed size divided by the decompressed size. Using 7 bit ASCII to encode a normal 8-bit ASCII file would give a compression ratio of 7/8, or 87.5%.

TABLE COMPRESSION and **SEQUENCE CODING** explore other related forms of mechanical redundancy.

2. Semantic Redundancy

Consider the traditional English song:

Verse 1:
Voice: Whear 'as tha been sin' I saw thee?
Reply: I saw thee
Chorus: On Ilkley Moor Bah t'at
Voice: Whear 'as tha been sin' I saw thee?
Reply: I saw thee
Voice: Whear 'as tha been sin' I saw thee?
Chorus: On Ilkley Moor Bah t'at
Reply: Bah t'at
Chorus: On Ilkley Moor Bah t'at
On Ilkley Moor Bah t'at

Verse 2:
Voice: Tha's been a-coortin' Mary Jane
Reply: Mary Jane
Chorus: On Ilkley Moor Bah t'at

… etc., for 7 more verses.

This song has plenty of redundancy because of all the repeats and choruses; you don't need to store every single word sung to reproduce the song. The songbook '*Rise up singing*' [Blood and Paterson 1992] uses bold type, parentheses and repetition marks to compress the complete song to 15 short lines, occupying a mere 6 square inches on the page without compromising readability:

1. Whear 'ast tha been sin' I saw thee (I saw thee)
On Ilkley Moor Bah T'at
Whear 'ast tha been sin' I saw thee (2x)

On Ilkley Moor bah t'at (bah t'at) on Ilkley Moor bah t'at
On Ilkely Moor bah t'at.
2. Tha's been a-coortin' Mary Jane
3. Tha'll go an' get thee death o' cowld

…etc., for 6 more lines

LZ compression (see **ADAPTIVE COMPRESSION**) and its variants use a similar but mechanical technique to remove redundancy in text or binary data.

3. Lossy Compression

Compression techniques that ensure that the result of decompression is exactly the same data as before compression are known as *lossless*. Many of the more powerful forms of compression are *lossy*. With lossy compression, decompression will produce an approximation to the original information rather than an exact copy. Lossy compression uses knowledge of the specific kind of data being stored, and of the required uses for the data.

The key to understanding lossy compression is the difference between *data* and *information*. Suppose you wish to communicate the information represented by the word "elephant" to an audience. Sent as a text string, 'elephant' occupies 8 7-bit ASCII characters, or 56 bits. Alternatively, as a spoken word encoded in 16 bit samples 8000 times per second, 'elephant' requires 1 second of samples, i.e. 128 KBits. A full-screen colour image of an elephant at 640*480 pixels might require 2.5 Mbits, and a video displayed for one second at 50 frames per second could take 50 times that, or 125 Mbits. None of the more expensive techniques convey much more information than just the text of "elephant", however. If all you are interested in the basic concept of an "elephant", most of the data required by the other techniques is redundant. You can exploit this redundancy in various ways.

Simplest, you can omit irrelevant data. For example you might be receiving uncompressed sound data represented as 16 bit samples. If your sound sampler isn't accurate enough to record 16-bit samples, the least significant 2 bits in each sample will be random, so you could achieve a simple compression by just storing 14 instead of 16 bits for each sample.

You can also exploit the nature of human perception to omit data that's less important. For example, we perceive sound on a 'log scale'; the ear is much less sensitive to differences in intensity when the intensity is high than when intensity is low. You can effectively compress sound samples by converting them to a log scale, and supporting only a small number of logarithmic intensities. This is the principle of some simple sound compression techniques,

particularly mu-law and a-law, which compress 14 bit samples to 8 bits in this way [CCITT G.711, Brokish and Lewis 1997].

You can take this idea of omitting data further, and transform the data into a different form to remove data irrelevant to human perception. Many of the most effective techniques do this:

JPEG The most commonly used variants of the JPEG standard represent each 8x8 pixel square as a composite of a standard set of 64 'standard pictures' – a fraction of each picture. The transformation is known as the 'cosine transform'. Then the fractions are represented in more or less detail according to the importance of each to human perception. This gives a format that compresses photographic data very effectively. [ITU T.87, Gonzalez and Woods 1992]

GSM GSM compression represents voice data in terms of a mathematical model of the human voice (Regular Pulse Excited Linear Predictive Coding)[1]. In this way it encodes separate 20mS samples in just 260 bits, allowing voice telephony over a digital link of only 13 Kbps. [Degener 1994]

GIF, PNG The proprietary GIF and standard PNG formats both map all colours in an image to a fixed-size palette before encoding. [CompuServe 1990, Boutell 1996].

MP3 MP3 represents sound data in terms of its composite frequencies – known as the 'Fourier Transformation'. The MP3 standard specifies the granularity of representation of each frequency according to its importance to the human ear and the amount of compression required, allowing FM radio-quality sound in 56 Kb per second [MP3].

MPEG The MPEG standard for video compression uses JPEG coding for initial frames. It then uses a variety of specific techniques – to spot motion in a variety of axes, changes of light, etc. – to encode the differences between successive frames in minimal data forms that fit in with the human perception of a video image [MPEG].

Some of these techniques exploit mechanical redundancy in the resulting data as well, using **TABLE COMPRESSION**, **DIFFERENCE CODING** and **ADAPTIVE** techniques.

Specialised Patterns

The rest of this chapter contains specialised patterns describing compression and packing techniques. Each of these patterns removes different kinds of mechanical and semantic redundancy, with different consequences for accessing the compressed data.

[1] GSM was developed using Scandinavian voices; hence all voices tend to sound Scandinavian on a mobile phone.

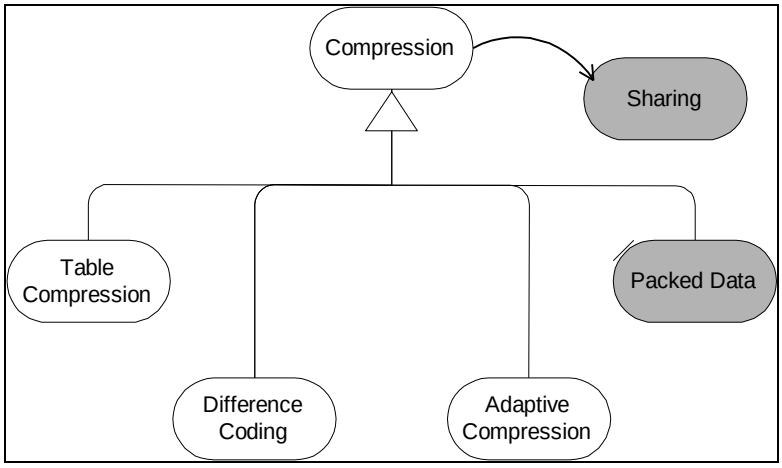


Figure 1: Compression Patterns

The patterns are as follows:

- **TABLE COMPRESSION** reduces the average number of bits to store each character (or value) by mapping it to a variable number of bits, such that the most common characters require the fewest bits.
- **DIFFERENCE COMPRESSION** addresses data series or sequences, by storing only the differences between successive items. Alternatively or additionally, if several successive items are the same it stores simply a count of the number of identical items.
- **ADAPTIVE COMPRESSION** analyses the data before or while compressing it to produce a more efficient encoding, storing the resulting parameters along with the compressed data – or uses the data itself as a table to support the compression.

The **PACKED DATA** pattern, that reduces the amount of memory allocated to store random-access data structures, can also been seen as a special kind of compression.

1. Evaluating Compression Techniques

There are many possible compression techniques. Here are some of the things to consider when choosing an appropriate technique:

1.1 Processing and Memory Required. Different techniques vary significantly in the processing and memory costs they impose. In general, **DIFFERENCE CODING** has the lowest costs, followed by fixed **TABLE COMPRESSION**, and most forms of **ADAPTIVE COMPRESSION** have quite high costs on both counts – but there are many exceptions to this rule. *Managing Gigabytes* [Witten, Moffat, Bell 1999] examines the costs in some detail.

1.2. Encoding vs. Decoding. Some compression algorithms reduce the processing cost of decoding the data by increasing the cost of encoding. This is particularly advantageous if there is one large and powerful encoding system and many decoders with a lower specification. This is a situation common in broadcast systems.

MP3 and MPEG, for example, require much more processing, code and memory to encode than decode, which suits them for broadcast transmission [MP3,MPEG]. Interestingly LZ **ADAPTIVE COMPRESSION** has the same feature, so ZIP archives can be distributed with their relatively simple decoding software built-in [Ziv and Lempel 1977].

Some compressed representations can be used directly without any decompression. For example, Java and Smalltalk use byte coding to reduce the size of executable programs; intermediate codes can be smaller than full machine language [Goldberg and Robson 1983, Lindholm and Yellin 1999]. These byte codes are designed so that they can be interpreted directly by a virtual machine, without a separate decompression step.

1.3. Programming Cost. Some techniques are simple to implement; others have efficient public domain or commercial implementations. Rolling your own complicated compression or decompression algorithm is unlikely to be a sensible option for many projects.

1.4. Random Access and Resynchronisation. Most compression algorithms produce a stream of bits. If this stream is stored in memory or in a file, can you access individual items within that file randomly, without reading the whole stream from the beginning? If you're receiving the stream over a serial line and the some is corrupted or deleted, can you resynchronise that data stream, that is, can you identify the start of a meaningful piece of data and continue decompression? In general, most forms of **TABLE COMPRESSION** can provide both random access and resynchronisation; **DIFFERENCE CODING** can also be tailored to handle both; **ADAPTIVE COMPRESSION**, however, is unlikely to work well for either.

Known Uses

Compression is used very widely. Operating systems use compression to store more information on secondary storage, communications protocols use compression to transmit information more quickly, virtual machines use compression to reduce the memory requirements of programs, and general purpose file compression tools are use ubiquitously for file archives.

See Also

As well as compressing information, you may be able to store it in cheaper **SECONDARY STORAGE** or **READ ONLY MEMORY.** You can also remove redundant data using **SHARING**

The excellent book 'Managing Gigabytes' [Witten, Moffat, Bell, 1999] explains all of this chapter's techniques for compressing text and images in much greater detail. 'Text Compression' [Bell, Cleary, Witten 1990] focuses on text compression.

The online book, 'Information Engineering Across the Professions' [Cyganski, Orr, and Vaz 1998] has explanations of many different kinds of Text, Audio, Graphical and Video compression.

The FAQ of the newsgroup `comp.compression` describes many of the most common compression techniques. Steven Kinnear's web page [1999] provides an introduction to multimedia compression, with an excellent set of links to other sites with more detail.

'Digital Video and Audio Compression' [Solari 1997] has a good description of techniques for multimedia compression.

Table Compression Pattern

Also know as: Nibble Coding, Huffman Coding.

How do you compress many short strings?

- You have lots of small-to-medium sized strings in your program — all different
- You need to reduce your program's *memory requirements.*
- You need random access to individual strings.
- You don't want to expend too much extra *programmer effort*, *memory space*, or *processing time* on managing the strings.

Many programs use a large number of strings — stored in databases, read from **RESOURCE FILES**, received via telecommunications links or hard-coded in the program. All these strings increase the program's *memory requirements* for main memory, read-only memory, and secondary storage.

Programs need to be able to perform common string operations such as determining their length and internal characters, concatenating strings, and substituting parameters into format strings, however strings are represented. Similarly, each string in a collection of strings needs to be individually accessible. If the strings are stored in a file on secondary storage, for example, we need *random access* to each string in the file.

Although storing strings is important, it is seldom the most significant memory use in the system. Typically you don't want to put to much *programmer effort* into the problem. Equally, you may not want to demand too much *temporary memory* to decompress each string.

For example, the Strap-It-On PC needs to store and display a large number of information and error messages to the user. The messages need to be stored in scarce main memory or read-only memory, and there isn't really enough space to store all the strings directly. The Programs must be able to access each string individually, to display them to the user when appropriate. Given that many of the strings describe exceptional situations such a memory shortage, they need to be able to be retrieved and displayed quickly, efficiently, and without requiring extra memory.

Therefore: *Encode each element in a variable number of bits so that the more common elements require fewer bits.*

The key to table compression is that some characters are statistically much more likely to occur than others. You can easily map from a standard fixed size representation to one where each character takes a different number of bits. If you analyse the kind of text you're compressing to find which characters are most probable, and map these characters to the most compact encoding, then on average you'll end up with smaller text.

For example the chart below shows the character frequencies for all the lower-case characters and spaces in a draft of this chapter.

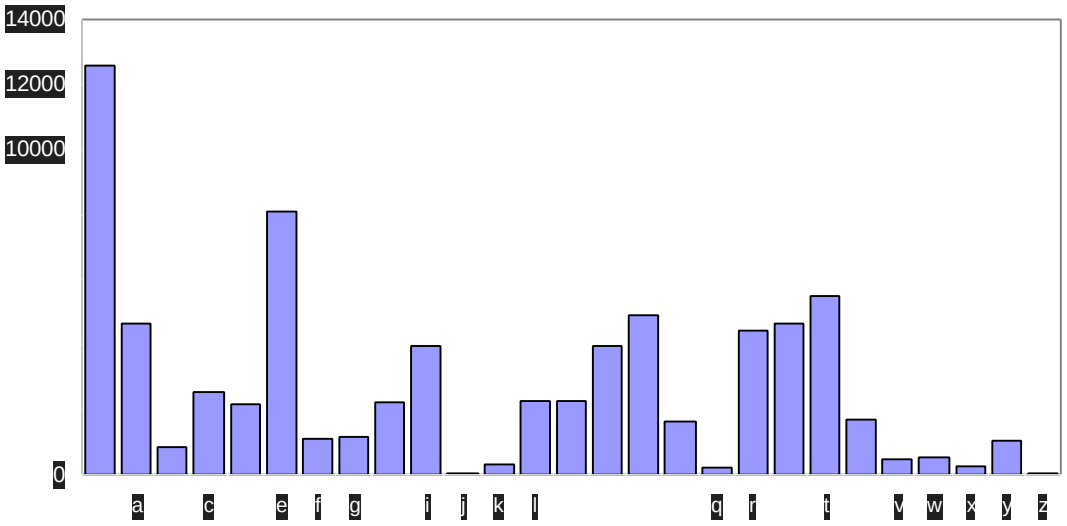


Figure 2 Distribution of Characters in this Chapter

Obviously some characters (space, 'e') appear much more often than others do ('z', 'j'). A surprisingly large proportion of the characters is never used at all[2]. The most common character, space, occupies 15% of the memory required to hold the chapter. The 15 most common characters occupy 75% of the total memory.

Given that the Strap-It-On's information and error messages have a similar distribution of characters, its designers get a significant reduction in the storage space required by encoding the most common characters in fewer bits, and the less common characters in more bits. Using the Huffman Encoding as described below, the designers of the Strap-it-on have achieved compression of 5 bits per character for its error messages, with negligible costs in run-time performance and temporary memory costs.

Consequences

Typically you get a reasonable compression for the strings themselves, reducing the program's *memory requirements*. Sequential operations on compressed strings execute almost as fast as operations on native strings, preserving *time performance*. String compression is quite easy to implement, so it does not take much *programmer effort*. Each string in a collection of compressed strings can be accessed individually, without decompressing all proceeding strings.

However: the total compression of the program data – including non-string data – isn't high, so the program's *memory requirements* may not be greatly reduced**.**

String operations that rely on random access to the characters in the string may execute up to an order of magnitude slower than the same operations on decompressed strings, reducing the program's *time performance*. Because characters may have variable lengths, you can only access a specific character by scanning from the start of the string. If you want operations that change the characters in the string you have to uncompress the string, make the changes, and recompress it.

It requires *programmer discipline* to use compressed strings, especially for string literals within the program code. Compressed strings require either manual encoding or a string pre-processing pass, either of which increases complexity.

[2] Well, hardly ever. [kjxcheck reference:Gilbert&Sullivan]

You have to *test* the compressed string operations, but these tests are quite straightforward.

❖ ❖ ❖

Implementation

There are many techniques used for table compression [Whitten et al 2000]. The following sections explore a few common ones.

1. Simple coding

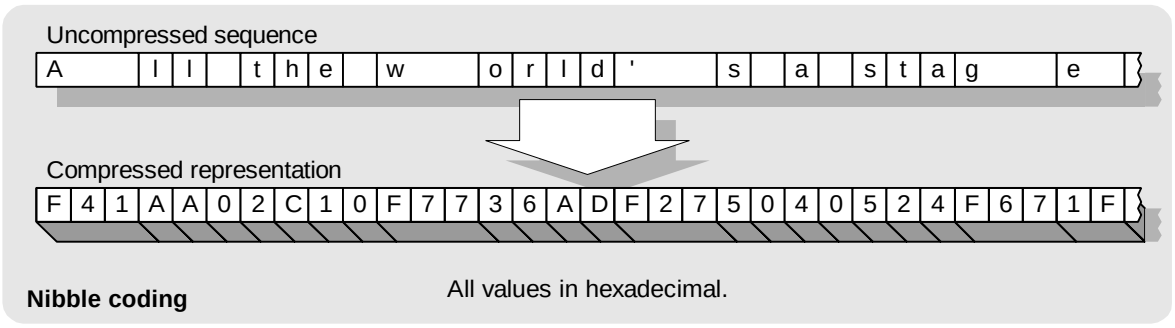
If the underlying character set has only 128 characters, such as ASCII, it certainly makes sense to store each character in seven bits, rather than eight, sixteen, or thirty-two bits. But, as we discussed above, in fact a large proportion of normal text could be encoded in just four bits. Other non-European languages might be better with five or six bits.

If you encode most of the text into, say, small fixed size characters, what do you do with the characters not within the most common set? The answer is to use 'escape codes'. An escape code is a special character that changes the meaning of the following character (or sometimes of the characters up to the next escape code).

For example, a common simple coding technique is to use a *nibble code*, where each character is coded into four bits. A nibble code is a easy to implement, because a nibble is always half a byte, making it easy to write the packed data. In a basic nibble code, we might have only one escape code, which is followed by the eight-bit ASCII code of the next character. So using the data in figure xx above to deduce the most common characters, we can construct an encoding and decoding table as follows:

Plain text	Encoded Nibbles	Encoded Bits
?	0	0000
a	4	0100
b	F 6 1	1111 0110 0001
c	F 6 2	1111 0110 0010
d	D	1101
e	1	0001
f	F 6 5	1111 0110 0101
… etc		

Thus the phrase "All the world's a stage" would encode as follows:



Nibble coding

Using this nibble code, 75% of characters in this chapter can be encoded in a 4 bits; the remainder all require 12 bits. On this simple calculation the average number of bits required per character is 6 bits; when we implemented the nibble code and tested it on the file, we achieved 5.4 bits per character in practice.

2. Huffman Coding

Why choose specifically 4 bits for the most common characters and 12 bits for the escaped characters? It would seem more sensible to have a more even spread, so that the most common characters (e.g. space) use less than four bits, fairly common characters ('u', 'w') require between four and eight bits, and only the least common ones ('Q', '&') require more. Huffman Coding takes this reasoning to its extreme. With Huffman coding, the 'Encoded bits' column in table xxx becomes will contain bit values of arbitrary lengths instead of either 4 or 12 [Huffman 1952].

Decoding Huffman data is a little trickier. Since you can't just look up an unknown length bit string in an array, Huffman tables are often represented as trees for decoding; each terminal node in the tree is a decoded character. To decode a bit string, you start at the root, then take the left node for each 1 and the right node for each 0. So, for example, if you had the following simple encoding table for a 4-character alphabet, where A is the most frequent character in your text, then D, then B and C:

Plain Text	Encoded Bits
A	1
B	010
C	011
D	00

This can be represented as a Huffman Tree as:

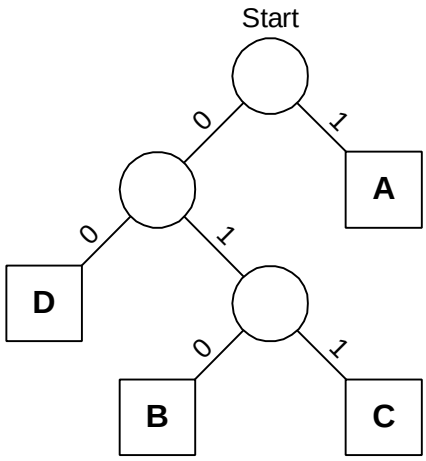


Figure 3: Huffman Tree

For more about Huffman coding – more efficient decoding techniques and a discussion on generating the Huffman encoding tables – see 'Managing Gigabytes' [Witten, et al 1999] or any other standard reference on text compression.

3. Encoding more than just characters

There's no need to limit **TABLE COMPRESSION** to strings; anything that contains data items of fixed numbers of bytes can be compressed in this way. Other compression techniques, for example, often apply Huffman coding to their intermediate representations to increase their compression ratios (see, for example, the **ADAPTIVE COMPRESSION** technique GZIP).

TABLE COMPRESSION does not have to be restricted to compressing fixed-length items, as long as each item has a clearly defined end. For example, Huffman Word Compression achieves very high compression ratios (3 bits per character or so) by encoding each word separately [Witten, et al 1999]. To achieve this ratio, Huffman Word Compression requires a very large compression table – the size of the dictionary used.

4. Compressing String Literals

Compressed strings are more difficult to handle in program code. While programming languages provide string literals for normal strings, they do not generally support compressed strings. Most languages support escape codes (such as "\x34") that allow any numeric characters to be stored into the string. Escape codes can be used store compressed strings in standard string literals. For example, here's a C++ string that stores the nibble codes for the "All the world's a stage" encoding in figure XX.

```
const char* AllTheWorldsAStage =
          "\xf4\x1a\xa0\x2c\x10\xf7\x73\x6a\xdf\x27\x50\x40\x52\x4f\x67\x1f";
```

You can also write a pre-processor to work through program texts, and replace standard encoded strings with compressed strings. This works particularly well when compressed strings can be written as standard string or array literals. Alternatively, in systems that store strings in **RESOURCE FILES**, the resource file compiler can compress the string, and the resource file reader can decompress it.

5. UTF8 Encoding

To support internationalisation, an increasing number of applications do all their internal string handling using two-byte character sets – typically the UNICODE standard [Unicode 1996]. Given that the character sets for most European languages require less than 128 characters, the extra byte is clearly redundant. For storage and transmission, many environments encode UNICODE strings using the UTF8 encoding.

In UTF8, each UNICODE double byte is encoded into one, two or three bytes (though the standard supports further extensions). The coding encodes the bits as follows:

UNICODE value	1st Byte	2nd Byte	3rd Byte
000000000xxxxxxx	0xxxxxxx		
00000yyyyyxxxxxx	110yyyyy	10xxxxxx	
Zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx

So in UTF8, the standard 7-bit ASCII characters are encoded in a single byte; in fact, the UNICODE encoding is exactly the same as the one byte ASCII encoding for these characters. Common extended characters are encoded as two bytes, with only the least common characters requiring three bytes. Every UTF8 character starts and ends on a byte boundary, so you can identify a substring within a larger buffer of UTF8 characters using just a byte offset and a length.

UTF8 wasdesigned to be especially suitable for transmission along serial connections. A terminal receiving UTF8 characters can always determine which byte represents the start of a UNICODE character, because either their top bit is 0, or the top two bits are '11'. Any UTF8 bytes with the top bits equal to "10" are always 2nd or 3rd in sequence and should be ignored unless the terminal has received the initial byte.

Example

This example implements a nibble code to compress the text in this chapter. Figure 3 below shows the distribution of characters in the text, sorted by frequency, with the 15 most common characters to the left of the vertical line.

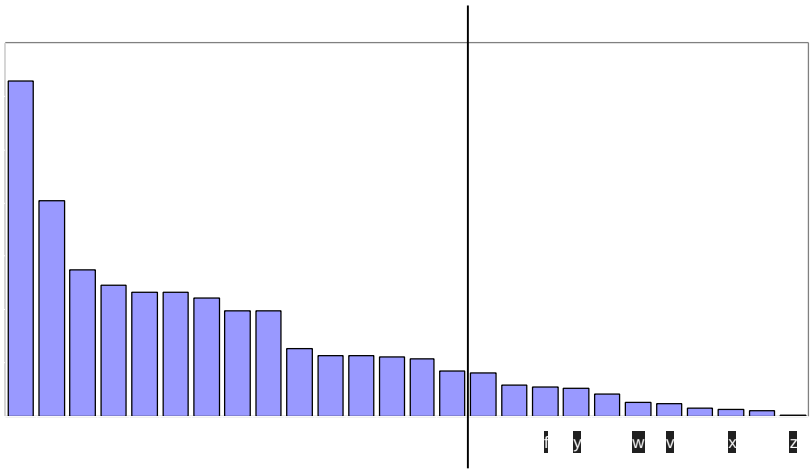


Figure 4 Choosing the 15 most common characters.

Here is a Java example that implements this nibble code. The `StringCompression` class uses a byte array output stream to simplify creating the compressed string — the equivalent in C++ would use an `ostrstream` instance. The most common characters are represented in the string `NibbleChars`:

```
protected final String NibbleChars = " etoasrinclmhdu";
protected final int NibbleEscape = 0xf;
protected int lastNibble;
protected ByteArrayOutputStream outStream;
```

The `encodeString` method takes each fixed-size character and encodes it, character by character. This function has to deal with end effects, ensuring the last nibble gets written to the output file by padding it with an escape character.

```
protected byte[] encodeString(String string) {
        outStream = new ByteArrayOutputStream();
        lastNibble = -1;
        for (int i = 0; i < string.length(); i++) {
            encodeChar(string.charAt(i));
        }
        if (lastNibble != -1) {
            putNibble(NibbleEscape);
        }
        byte[] result = outStream.toByteArray();
        outStream = null;
        return result;
}
```

The most important routine encodes a specific character. The `encodeChar` method searches the `NibbleChars` string directly; if the character to be encoded is in the string it is output as a nibble, otherwise we output an escape code and a high and low nibble. A more efficient implementation could use a 256-entry table lookup.

```
protected void encodeChar(int charCode) {
        int possibleNibble = NibbleChars.indexOf(charCode);
        if (possibleNibble != -1) {
            putNibble(possibleNibble);
        } else {
            putNibble(NibbleEscape);
            putNibble(charCode >>> 4);
            putNibble(charCode & 0xf);
        }
    }
```

The `putNibble` method simply adds one nibble to the output stream. We can only write whole bytes, rather than nibbles, so the `lastNibble` variable stores a nibble than has not been output. When another nibble is received, both `lastNibble` and the current nibble `n` can be written as a single byte:

```
protected void putNibble(int nibble) {
        if (lastNibble == -1) {
            lastNibble = nibble;
        } else {
            outStream.write((lastNibble << 4) + nibble);
            lastNibble = -1;
        }
    }
}
```

Decoding is similar to encoding. For convenience, the decoding methods belong to the same class; they use a `ByteArrayInputStream` to retrieve data. The `decodeString` method reads a character at a time and appends it to the output:

```
    protected ByteArrayInputStream inStream;

    protected String decodeString(byte [] inBuffer) {
        inStream = new ByteArrayInputStream(inBuffer);
        StringBuffer outString = new StringBuffer();
        lastNibble = -1;
        int charRead;
        while ((charRead = decodeChar()) != -1) {
            outString.append( (char)charRead );
        }
        return outString.toString();
    }
```

The `decodeChar` method reads as many input nibbles as are required to compose a single character.

```
    protected int decodeChar() {
        int s = getNibble();
        if (s == -1) return -1;
        if (s != NibbleEscape) {
            return NibbleChars.charAt(s);
        } else {
            s = getNibble();
            if (s == -1) return -1;
            return (s << 4) + getNibble(); }
    }
```

Method `getNibble` actually returns one nibble from the input stream, again keeping the extra nibble in the `lastNibble` field when a full byte is read by only one nibble returned.


```
protected int decodeChar() {
    int nibble = getNibble();
    if (nibble == -1) {
        return -1;
    }
    if (nibble != NibbleEscape) {
        return NibbleChars.charAt(nibble);
    } else {
        nibble = getNibble();
        if (nibble == -1) {
            return -1;
        }
        return (nibble << 4) + getNibble();
    }
}
```

Nibble encoding can be surprisingly effective. For example a text-only version of this chapter compresses to just 5.4 bits per char (67%) using this technique. Similarly, the complete set of text resources for a release of the EPOC32 operating system would compress to 5.7 bits per character (though as the total space occupied by the strings is only 44 Kb, the effort and extra code required have so far not been worthwhile).

❖ ❖ ❖

Known Uses

Reuters worldwide IDN network uses Huffman encoding to reduce the bandwidth required to transmit all the world's financial market prices worldwide. The IDN Huffman code table is reproduced and re-implemented in many different systems.

GZIP uses Huffman encoding as part of the compression process, though their main compression gains are from **ADAPTIVE COMPRESSION**. [Deutsch 1996]

The MNP5 and V42.bis modem compression protocols uses Huffman Encoding to get compression ratios of 75% to 50% on typical transmitted data [Held 1994].

Nibble codes were widely used in versions of text adventure games for small machines [Blank and Galley 1980]. Philip Gage used a similar technique to compress an entire string table [Gage 97]. Symbian's EPOC16 operating system for the Series 3 used table compression for its **RESOURCE FILES** [Edwards 1997].

Variants of UTF8 encoding are used in Java, Plan/9, and Windows NT to store Unicode characters [Unicode 1996, Lindholm and Yellin 1999, Pike and Thompson 1993].

See Also

Many variants of table compression are also **ADAPTIVE**, calculating the optimal table for each large section of data and including the table with the compressed data.

Compressed strings can be stored in **RESOURCE FILES** in **SECONDARY STORAGE** or **READ-ONLY MEMORY**, as well as primary storage. Information stored in **DATA FILES** can also be compressed.

Witten, Moffat, and Bell [1999] and Cyganski, Orr, and Vaz [1998] discuss Huffman Encoding and other forms of table compression in much more detail than we can give here. Witten, Moffat and Bell also includes discussions of memory and time costs for each technique.

Business Data Communications and Networking [Fitzgerald and Dennis 1995] provides a good overview of modem communications. Sharon Tabor's course materials for '*Data Transmission*' [2000] provide a good terse summary. The '*Ultimate Modem Handbook*' includes an outline of various modem compression standards [Lewart 1999].

Difference Coding Pattern

Also know as: Delta Coding, Run Length Encoding

How can you reduce the memory used by sequences of data?

- You need to reduce your program's *memory requirements*
- You have *large* streams of data in your program
- The data streams which will be accessed sequentially
- There are significant time or financial costs of storing or transferring data.

Many programs use sequences or series of data — for example, sequential data such as audio files or animations, time series such as stock market prices, values read by a sensor, or simply the sequence of bytes making up program code. All these sequences increase the program's *memory requirements*, or worsen the *transmission time* using a telecommunications link.

Typically this sort of streamed data is accessed sequentially, beginning at the first item and then processing each item in turn. Programs rarely or never require random access into the middle of the data. Although storing the data is important, it often isn't the largest problem you have to face — gathering the data is often much more work than simply storing it. Typically, you don't want to devote too much *programmer effort*, *processing time*, or *temporary memory* to the compression operations.

For example, the Strap-It-On PC needs to store results collected from the Snoop-Tronic series of body wellbeing monitors. These monitors are attached onto strategic points on the wearer's body, and regularly measure and record various physiological, psychological, psychiatric and psychotronic metrics (heartbeats, blood-sugar levels, alpha-waves, influences from the planet Gorkon, etc). This information needs to be stored in the background while the Strap-It-On is doing other work, so the recording process cannot require much processor time or memory space. The recording is continuous, gathering data whenever the Strap-It-On PC and Snoop-Tronic sensors are worn and the wearer is alive, so large amounts of data are recorded. Somehow, we must reduce the memory requirements of this data.

Therefore: *represent sequences according to the differences between each item.*

Continuous data sequences are rarely truly random — the recent past is often an excellent guide to the near future. So in many sequences:

- The values don't change very much between adjacent items, and
- There are 'runs', where is no change for several elements.

These features result in two complementary techniques to reduce the number of bits stored per item:

- Delta Coding stores just differences between each successive item.
- Run-length Encoding (RLE) replaces a run of identical elements with a repeat count.

For example, in the data stored by the Snoop-Tronic monitors, the values read are very close or the same for long periods of time. The Strap-It-On PCs driver for the Snoop-Tronic sensors uses sequence coding on the data streams as they arrive from each sensor, buffers the data, and stores it to secondary storage — without imposing a noticeable overhead on the performance of the system.

Consequences

Difference compression can achieve an excellent compression ratio for many kinds of data (particularly cartoon-style picture data and some forms of sound data) reducing the program's *memory requirements*. Sequential operations on the compressed data can execute almost as fast as operations on native values preserving *time performance* and *real-time responsiveness,* and considerably improving *time performance* if there are slow disk or telecommunication links involved.

Difference compression is quite easy to implement, so it does not take much *programmer effort*, or extra *temporary memory*.

However: The compressed sequences are more to difficult to manage than sequences of absolute data values. In particular, it is difficult to provide random access into the middle of compressed sequences without first uncompressing them, requiring *temporary memory* and *processing time*.

Some kinds of data – such as hi-fi sound or photographic images – don't reduce significantly with **DIFFERENCE COMPRESSION**.

You have to *test* the compressed sequence operations, but these tests are quite straightforward.

❖ ❖ ❖

Implementation

Here are several difference coding techniques that you can consider:

1. Delta Coding

Delta coding (or difference coding) stores differences between adjacent items, rather than the absolute values of the items themselves [Bell et all 1990]. Delta coding saves memory space because deltas can often be stored in smaller amounts of memory than absolute values. For example, you may be able to encode a slowly varying stream of sixteen bit values using only eight-bit delta codes.

Of course, the range of values stored in the delta code is less than the range of the absolute item values (16-bit items range from 0 to 65536, while 8 bit deltas give you +- 127). If the difference to be stored is larger than the range of delta codes, typically the encoder uses an escape code (a special delta value, say –128 for an 8-bit code) followed by the full absolute value of the data item.

Figure xx below shows such a sequence represented using delta coding. All values are in decimal, and the escape code is represented as '?':

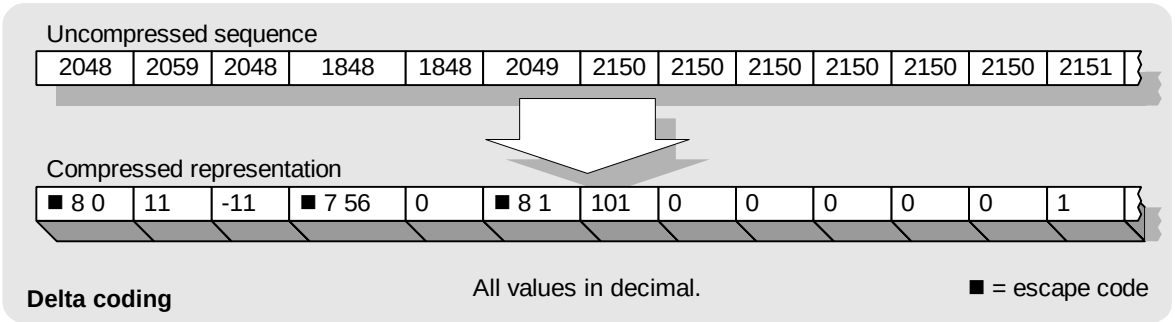


Figure 5: Delta Coding

2. Run Length Encoding

Run-length encoding compresses a run of duplicated items by storing the value of the duplicated items once, followed by the length of the run [Bell et all 1990]. For example, we can extend the delta code above to compress runs by always following the escape code by a count byte as well as the absolute value. Runs of between 4 and 256 items can be compressed as the escape code, the absolute value of the repeated item, and the count. Runs of longer than 256 items can be stored as repeated runs of 256 characters, plus one more run of the remainder. Figure XX shows RLE added to the previous example:

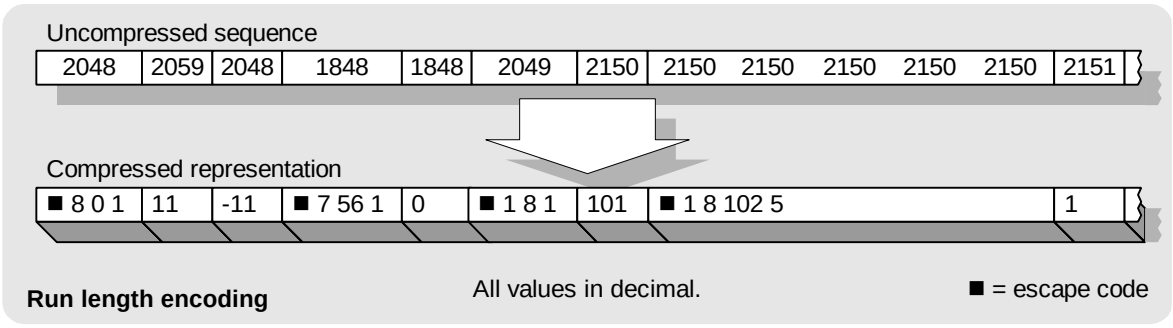


Figure 6: Run Length Encoding and Delta Coding

3. Lossy Difference Compression

Here are some common techniques that increase the compression ratio of sequence compression by losing some of the information compressed:

1. You can treat a sequence with only negligible differences in values as if they were a run of items with identical values. For example, in the data series above, differences within a quarter of a percent of the absolute value of the data items may not be significant in the analysis. Quite possibly they could be due to noise in the recording sensor or the ambient temperature when the data item was recorded. A quarter of one percent of 2000 is 20 — so we can code the first three items as a run.
2. You can handle large jumps in delta values by allowing a lag in catching up. Thus, for example, the difference of 200 between 2048 to 1848 can be represented as two deltas, rather than an escape code.

Using these two techniques, we can code the example sequence as shown in figure XX:

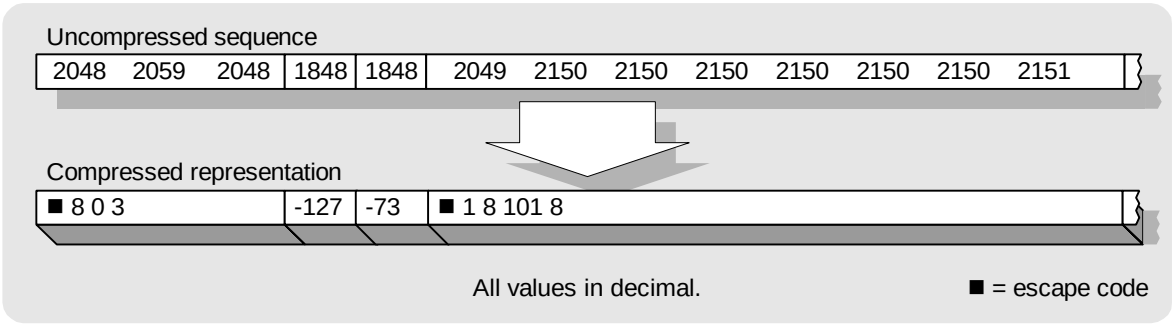


Figure 7: Lossy Sequence Compression

3. You can increase the granularity of the delta values, so that each delta value is scaled by the magnitude of the items they are representing. So, for example, each delta step could be the nearest integer below 0.25% of the previous item's value, allowing much larger deltas.

4. Resynchronisation

Sequence compression algorithms are often used for broadcast communications and serial or network connections. In many cases, particularly with multimedia data streams, it doesn't matter very much if part of the sequence is lost or corrupted, so long as later data can be read correctly. Because difference codes assume the receiver knows the correct value for the last item (so that the next item can be computed by adding the difference), one wrong delta means that every subsequent delta will produce the wrong value. To avoid this problem, you can include resynchronisation information; every now and again you can send a complete value as escape code, instead of a delta. The escape code resets the value of the current item, correcting any accumulated error due to corruption or lost data.

5. Non-numeric data

Difference Coding can also be very effective at compressing non-numeric data structures. In Delta Coding, the deltas will be structures themselves; for RLE represents events where the structures haven't changed. For example, you can think some forms of the Observer pattern [Gamma et al 1995] as examples of delta compression: the observer is told only the changes that have happened.

Similarly you can do run-length encoding using a count of a number of identical structures. For example the X Window System can return a single compressed mouse movement event that represents a number of smaller movements — the compressed event contains a count of the number of uncompressed movements it represents [Scheifler and Gettys 1986].

Examples

The following Java example compresses a sequence of two-byte values into a sequence of bytes using both difference compression and run length encoding. The compression is lossless, and the only escape sequence contains both the complete value and the sequence length. As above, the bytes of the escape sequence are:

```
<escape> <high byte of repeated value> <low byte> <sequence count>
```

The `encodeSequence` method takes a sequence of shorts, and passes each one to the `encodeShort` method, which will actually encode them:

```
protected final int SequenceEscape =  0xff;
protected final int MaxSequenceLength =  0xFE;
protected short lastShort;
protected short runLength;

protected void encodeSequence(short[] inputSequence) {
    lastShort = 0;
    runLength = 0;

    for (int i = 0; i < inputSequence.length; i++) {
        encodeShort(inputSequence[i]);
    }
    flushSequence();

}
```

The `encodeShort` method does most of the work. It first checks if its argument is part of a sequence of identical values, and if so, simply increases the run length count for the sequence — if the sequence is now the maximum length that can be represented, an escape code is written. If its argument is within the range of the delta coding (± 128 from the last value) an escape code

is written if necessary, and a delta code is written. Finally, if the argument is outside the range, an escape code is written to terminate the current run length encoded sequence if necessary. In any event, the current argument is remembered in the `lastShort` variable.

```
protected void encodeShort(short s) {
        if (s == lastShort) {
            runLength++;
            if (runLength >= MaxSequenceLength) {
                flushSequence();
            }
        } else if (Math.abs(s - lastShort) < 128 ) {
            flushSequence();
            writeEncodedByte(s - lastShort + 128);
        } else {
            flushSequence();
            runLength++;
        }
        lastShort = s;
}
```

The `flushSequence` method simply writes out the escape codes, if required, and resets the run length. It is called whenever a sequence may need to be written out — whenever `encodeShort` detects the end of the current sequence, or that the current sequence the longest that can be represented by the run length escape code.

```
    protected void flushSequence() {
        if (runLength == 0) return;
        writeEncodedByte(SequenceEscape);
        writeEncodedByte(lastShort >>> 8);
        writeEncodedByte(lastShort & 0xff);
        writeEncodedByte(runLength);
        runLength = 0;
}
```

The corresponding decoding functions are straightforward. If an escape code is read, a run of output values is written, and if a delta code is read, a single output is written which differs from the last output value by the delta.

```
protected void decodeSequence(byte[] inBuffer) {
        ByteArrayInputStream inStream =
            new ByteArrayInputStream(inBuffer);
        lastShort = 0;
        int byteRead;

        while ((byteRead = inStream.read()) != -1) {
            byteRead = byteRead & 0xff;

            if (byteRead == SequenceEscape) {
                lastShort = (short) (((inStream.read() &0xff ) << 8) +
                                     (inStream.read() & 0xff));
                for (int c = inStream.read(); c > 0; c--) {
                    writeDecodedShort(lastShort);
                }
            } else {
                writeDecodedShort(lastShort += byteRead -128);
            }
        }
    }
```

❖ ❖ ❖

Known Uses

Many image compression techniques use Different Compression. The TIFF image file format uses RLE to encode runs of identical pixels [Adobe 1992. The GIF and PNG formats do the same after (lossy) colour mapping [CompuServe 1987, Boutell 1996]. The Group 3 and 4 Fax transmission protocols uses RLE to encode the pixels on a line [Gonzalez and Woods 1992]; the next line (in fine mode) or three lines (in standard mode) are encoded as differences from the first line.

MPEG video compression uses a variety of techniques to express each picture as a set of differences from the previous one [MPEG, Kinnear 1999]. The V.42bis modem compression standard includes RLE and **TABLE COMPRESSION** (Huffman Coding), achieving a total compression ratio of up to 33% [Held 1994].

Many window systems in addition to X use Run Length encoding to compress events. For example MS Windows represents multiple mouse movements and key auto-repeats in this way, and EPOC's Window Server does the same [Petzold 1998, Symbian 1999].

Reuters IDN system broadcasts the financial prices from virtually every financial exchange and bank in the world, aiming – and almost always succeeding – in transmitting every update to every interested subscriber in under a second. To make this possible, IDN represents each 'instrument' as a logical data structure identified by a unique name (Reuters Identification Code); when the contents of the instrument (prices, trading volume etc.) change, IDN transmits only the changes. To save expensive satellite bandwidth further, these changes are transmitted in binary form using Huffman Coding (see **TABLE COMPRESSION**), and to ensure synchronisation of all the Reuters systems worldwide, the system also transmits a background 'refresh' stream of the complete state of every instrument.

See Also

You may want to use **TABLE COMPRESSION** in addition to, or instead of **DIFFERENCE CODING**. If you have a large amount of data, you may be able to tailor your compression parameters (**ADAPTIVE COMPRESSION**), or to use a more powerful **ADAPTIVE** algorithm.

The references discussed in the previous patterns are equally helpful on the subject of **DIFFERENCE COMPRESSION**. Witten, Moffat and Bell [1999] explain image compression techniques and tradeoffs; Cyganski, Orr, and Vaz [1998] and Solari [1997] explain audio, graphical and video compression techniques, and Held [1994] discusses modem compression.

Adaptive Compression Pattern

How can you reduce the memory needed to store a large amount of bulk data?

- You have a large amount of data to store, transmit or receive.
- You don't have enough persistent memory space to store the information long term, or you need to communicate the data across a slow telecommunications link
- You have transient memory space for processing the data.
- You don't need random access to the data

A high proportion of the memory requirements of many programs is devoted to bulk data. For example, the latest application planned for the Strap-It-On PC is ThemePark:UK, a tourist guide being produce in conjunction with the Unfriendly Asteroid travel consultancy. ThemePark:UK is based on existing ThemePark products, which guide users around theme parks in Southern California. ThemePark:UK will treat the whole of the UK as a single theme park; the Strap-It-On will use its Global Positioning System together an internal database to present interactive travel guides containing videos, music, voice-overs, and genuine people personalities for cute interactive cartoon characters. Unfortunately the UK is a little larger than most theme parks, and the designers have found that using **TABLE COMPRESSION** and **DIFFERENCE COMPRESSION** together cannot cram enough information into the Strap-It-On's memory.

This kind of problem is common in applications requiring very large amounts of data, whether collections of documents and emails or representations of books and multimedia Even if systems have sufficient main memory to be able to process or display the parts of the data they need at any given time, they may not have enough memory to store all the information they will ever need, either in main memory or secondary storage.

Therefore: *Use an adaptive compression algorithm.*

Adaptive compression algorithms can analyse the data they are compressing and modify their behaviour accordingly. These adaptive compression algorithms can provide high compression ratios, and work in several ways:

- Many compression mechanisms require parameters, such as the table required for table compression or the parameters to decide what data to discard with lossy forms of compression. An adaptive algorithm can analyse the data it's about to compress, choose parameters accordingly, and store the parameters at the start of the compressed data.
- Other adaptive techniques adjust their parameters on the fly, according to the data compressed so far. For example Move-to-front (or MTF) transformations change the table used in, say Nibble Compression, so that the table of codes translating to the minimum (4-bit) representation is always the set of most recently seen characters.
- Further techniques, predominantly the Lempel-Ziv family of algorithms, use the stream of data already encoded as a string table to provide a compact encoding for each string newly received.

Implementations of many adaptive compression algorithms are available publicly, either as free or open source software, or from commercial providers.

For example, ThemePark:UK uses the *gzip* adaptive file compression algorithm for its text pages, which achieves typical compressions of 2.5 bits per character for English text, and requires fairly small amounts of RAM memory for decoding. ThemePark:UK also uses JPEG

compression for its images, PNG compression for its maps and cartoons, and MP3 compression for sounds.

Consequences

Modern adaptive compression algorithms provide excellent compression ratios, reducing your *memory requirements*. They are widely used and are incorporated into *popular industry standards* for bulk data.

Adaptive compression can also reduce *data transmission times* for telecommunication. File compression can also reduce the *secondary storage* requirements or *data transmission times* for program code.

However: File compression can require a significant *processing time* to compress and decompress large bulk data sets, and so they are generally unsuitable for *real-time work*. Some *temporary memory* (primary and secondary storage) will be necessary to store the decompressed results and to hold intermediate structures.

The performance of compression algorithms can vary depending on the type of data being compressed, so you have to select your algorithm carefully, requiring *programmer effort*. If you cannot reuse an existing implementation you will need significant further *programmer effort* to code up one of these algorithms, because they can be quite complex. Some of the most important algorithms are *patented*, although you may able to use non-patented alternatives.

❖ ❖ ❖

Implementation

Designing efficient and effective adaptive compression algorithms is a very specialised task, especially as the compression algorithms must be tailored to the type of data being compressed. For most practical uses, however, you do not need to design you own text compression algorithms, as libraries of compression algorithms are available both commercially and under various open source licences. Sun's Java, for example, now officially includes a version of the *zlib* compression library, implementing the same compression algorithm as the *pkzip* and *gzip* compression utilities. In most programs, compressing and decompressing files or blocks of data is as simple as calling the appropriate routine from one of these libraries.

1. LZ Compression

Many of the most effective compression schemes are variants of a technique devised by Zip and Lempel [1977]. Lempel-Ziv (LZ77) compression uses the data already encoded as a table to allow a compact representation of following data. LZ compression is easy to implement; and decoding is fast and requires little extra temporary memory.

LZ77 works by encoding sequences of tuples. In each tuple, the first two items reference a string previously coded – as an offset from the current position, and a length. The third item is a single character. If there's no suitable string previously coded, the first two items are zero. For example, the following shows the LZ77 encoding of the song chorus "do do ron ron ron do do ron ron".

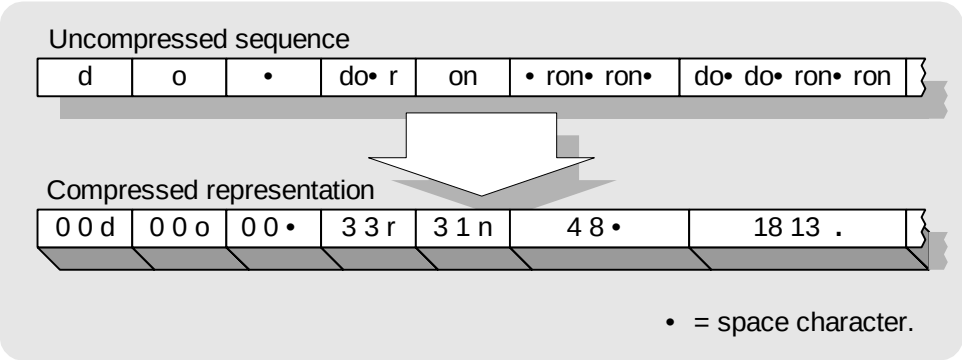


Figure 8: LZ77 Compression

Note how the repeating sequence " ron ron" is encoded as a single tuple; this works fine for decompression and requires only a small amount of extra effort in the compression code.

There are many variants of LZ compression, adding other forms of compression to the output, or tailored for fast or low-memory compression or decompression. For example GZIP encodes blocks of 64Kb at a time, and uses Huffman Coding to compress the offset and length fields of each tuple still further.

Examples

We examine two examples of adaptive compression. The first, MTF compression, is a simple adaptive extension of Table Compression. The second, more typical of real-world applications, simply uses a library to do compression and decompression for us.

1. MTF Compression

Move-To-Front (MTF) compression can adapt Nibble Compression to the data being encoded, by changing the compression table dynamically so that it always contains the 15 most recently used characters [Bell et al 1990]. The following code shows only the significant changes from the Nibble Coding example in **TABLE COMPRESSION**.

First, we need a modifiable version of the table. As with the fixed version, it can be a simple string.

```
protected String NibbleChars = " etoasrinclmhdu";
```

To start off, we set the table to be a best guess, so both the `encodeString` and `decodeString` methods start by resetting `currentChars` to the value `NibbleChars` (not shown here). Then we simply need to modify the table after encoding each character, by calling the new method `updateCurrent` in `encodeChar`:

```
protected void encodeChar(int charCode) {
        int possibleNibble = NibbleChars.indexOf(charCode);
        if (possibleNibble != -1) {
            putNibble(possibleNibble);
        } else {
            putNibble(NibbleEscape);
            putNibble(charCode >>> 4);
            putNibble(charCode & 0xf);
        }
        updateCurrent((char) charCode);
    }
```

The `updateCurrent` method updates the current table, either by moving the current character to the front of the table. If that character is already in the table, it gets pushed to the front; if not, then the last (least recently used) character is discarded:

```
protected void updateCurrent(int c)
    {
        int position = NibbleChars.indexOf(c);
        if (position != -1) {
            NibbleChars = "" + c + NibbleChars.substring(0, position) +
                NibbleChars.substring(position+1);
        } else {
            position = NibbleChars.length() - 1;
            NibbleChars = "" + c + NibbleChars.substring(0, position);
        }
    }
```

The `decodeChar` needs to do the same update for each character decoded:

```
    protected int decodeChar() {
        int result;
        int nibble = getNibble();
        if (nibble == -1) {
            return -1;
        }
        if (nibble != NibbleEscape) {
            result = NibbleChars.charAt(nibble);
        } else {
            nibble = getNibble();
            if (nibble == -1) {
                return -1;
            }
            result = (nibble << 4) + getNibble();
        }
        updateCurrent(result);
        return result;
    }
```

This example doesn't achieve as much compression as the fixed table for typical English text; for the text of this chapter it achieves only 6.2 bits per character. The MTF version does achieve some degree of compression on almost any non-random form of text, however, including executable binaries.

2. ZLIB Compression

This example uses an existing compression library, and so is more typical of real-world applications of adaptive compression. The Java Zlib libraries provide compressing streams that are **DECORATORS** of existing streams [Gamma et al 1995, Chan, Lee and Kramer 1998]. This makes it easy to compress any data that can be implemented as a stream. To compress some data, we open a stream on that data, and pass it through a compressing stream and then to an output stream.

```
protected static byte[] encodeSequence(byte[] inputSequence)
            throws IOException {
        InputStream inputStream = new ByteArrayInputStream(inputSequence);
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        GZIPOutputStream out = new GZIPOutputStream(outputStream);

        byte[] buf = new byte[1024];
        int len;
        while ((len = inputStream.read(buf)) > 0) {
            out.write(buf, 0, len);
        }
        out.close();
        return outputStream.toByteArray();
    }
```

In this model, decompressing is much like compressing. This time, the compressing stream is on the reading side; but in all other respects the code is virtually the same.

```
protected static byte [] decodeSequence(byte [] s) throws IOException {
        GZIPInputStream inputStream =
            new GZIPInputStream(new ByteArrayInputStream(s));
        ByteArrayOutputStream outputStream =
            new ByteArrayOutputStream();

        byte[] buf = new byte[1024];
        int len;
        while ((len = inputStream.read(buf)) > 0) {
            outputStream.write(buf, 0, len);
        }
        outputStream.close();
        return outputStream.toByteArray();
}
```

❖ ❖ ❖

Known Uses

Lempel-Ziv and variant compression algorithms are an industry standard, evidenced by the many `PKZip` and `gzip` file compression utilities used to reduce the size of email attachments, or to archive little-used or old versions of files and directories [Ziv and Lempel 1977, Deutsch 1996].

The PDF format for device-independent images uses LZ compression to reduce its file sizes [Adobe 1999] . Each PDF file contains one or more streams, each of which may be compressed with LZ.

File compression is also used architecturally in many systems. Linux kernels can be stored compressed and are decompressed when the system boots, and Windows NT supports optional file compression for each disk volume [Ward 1999, Microsoft NT 1996]. Java's JAR format uses gzip compression [Chan et al 1998] although designing alternative class file formats specially to be compressed can give two to five times better compression than gzip applied to the standard class file format [Horspool and Corless 1998, Pugh 1999]. Some backup tape formats use compression, notably the Zip and Jaz drives, and the HTTP protocol allows any web server to compress data, though as far as we are aware this feature is little used [Fielding 1999].

The current state-of-the-art library for adaptive file compression, Bzip2, achieves typical compressions of 2.3 bits per character on English text by transforming the text data before using LZ compression [Burroughs Wheeler 1994]. BZip2 requires a couple of Mbytes of RAM to compress effectively. See Witten et al [1999] and BZip2's home page [BZip2] for more detail.

See Also

TABLE COMPRESSION and **DIFFERENCE CODING** are often used with, or as part of, adaptive compression algorithms. You may also need to read a file a bit at a time (**DATA FILES)** to compress it.

Text Compression [Bell et al 1990] and *Managing Gigabytes* [Witten et al 1999] describe and analyse many forms of adaptive compression, including LZ compression, arithmetic coding and many others.
