Off-Line Dictionary-Based Compression

The dictionary-based compression methods described in Chapter 3 of the book are different, but have one thing in common; they generate the dictionary as they go along, reading data and compressing it. The dictionary is not included in the compressed file and is generated by the decoder in lockstep with the encoder. Thus, such methods can be termed "online." In contrast, the methods described here are also dictionary based, but can be considered "offline" because they include the dictionary in the compressed file.

The first method is byte pair encoding (BPE). This is a simple compression method, due to [Gage 94], that often features only mediocre performance. It is described here because (1) it is an example of a multipass method (two-pass compression algorithms are common, but multipasses are normally considered too slow) and (2) it eliminates only certain types of redundancy and should therefore be applied only to data files that feature this redundancy. (The second method, by [Larsson and Moffat 00], does not suffer from these restrictions and is much more efficient.) BPE is both an example of an offline dictionary-based compression algorithm and a simple example (perhaps the simplest) of a grammar-based compression method. In addition, the BPE decoder is very small, which makes it an ideal candidate for applications where memory size is restricted.

The BPE method is easy to understand. We assume that the data symbols are bytes and we use the term *bigram* for a pair of consecutive bytes. Each pass locates the most-common bigram and replaces it with an unused byte value. Thus, the method performs best on files that have many unused byte values, and one aim of this document is to show what types of data feature this kind of redundancy. First, however, a small example. Given the character set `A`, `B`, `C`, `D`, `X`, and `Y` and the data file `ABABCABCD` (where `X` and `Y` are unused bytes), the first pass identifies the pair `AB` as the most-common bigram and replaces each of its three occurrences with the single byte `X`. The result is `XXCXCD`. The second pass identifies the pair `XC` as the most-common bigram and replaces each of its two occurrences with the single byte `Y`. The result is `XYYD`, where every bigram occurs just once. Bigrams that occur just once can also be replaced, if more unused byte values are available. However, each replacement rule must be appended to the dictionary and thus ends up being included in the compressed file. As a result, the BPE encoder stops when no bigram occurs more than once.

What types of data tend to have many unused byte values? The first type that comes to mind is text. Currently, most text files use the well-known ASCII codes to encode text. An ASCII code occupies a byte, but only seven bits constitue the actual code. The eighth bit can be used for a parity check, but is often simply set to zero. Thus, we can expect 128 byte values out of the 256 possible ones to by unused in a typical ASCII text file. A quick glance at an ASCII code table shows that codes 0 through 32 (as well as code 127) are control codes, which are used for commands such as backspace, carriage return, escape, delete, and blank space. It therefore makes sense to expect only a few of those to appear in any given text file.

The validity of these arguments can be checked by a simple test. The following Mathematica code prints the unused byte values in a given text file.

```
scn = ReadList["Hobbit.txt", Byte];
btc = Table[0, {256}];
Do[btc[[scn[[i]]]] = btc[[scn[[i]]]] + 1, {i, 1, Length[scn]}];
btc
dis = Table[0, {256}]; j = 0;
Do[If[btc[[i]] == 0, {j = j + 1, dis[[j]] = i - 1}], {i, 1, 256}];
Take[dis, j]
```

It was executed on three chapters from the book *Data Compression: The Complete Reference* and on Tolkien's *The Hobbit*. The following results were obtained:

Dcomp3: 0–7, 9–11, 13–30, 126–255.
Dcomp4.1: 0–11, 13–30, 126–255.
Dcomp5: 0–7, 9–11, 13–30, 126–255.
Hobbit: 0–7, 9–11, 13–30, 34–36, 42, 46, 60, 63, 87, 89–92, 95, 122–123, 125–255.

The results of this experiment are clear. More than half the byte values are unused. The 128 values 128 through 255 are unused and the only ASCII control characters used are BS, FF, US, and Space.

Today, more and more text files are encoded in Unicode, but even such files should have many unused byte values. (Notice that most Unicodes are 16 bits, but there are also 8-bit and 32-bit Unicodes.) A typical Unicode text file in an alphabetic language consists of letters, digits, punctuation marks, and accented letters, so the total number of codes is around 100–150, leaving many unused byte values. As an example, the 128 Unicodes for Greek and Coptic [Greek-Unicode 07] are `0370` through `03FF`, and these do not use byte values `00`, `01`, and `04` through `6F`. Naturally, text files in an ideograph-based language, such as Bopomofo or Cuneiforms, easily use all 256 byte values.

Grayscale images constitute another example of data where many byte values may be unused. A typical image in grayscale may have millions of pixels, each in one of 256 shades of gray, but many shades may be unused. An experiment with the Mathematica code above indicates 41 unused byte values in the well-known `lena` image (raw, 128×128, 1-byte pixels), and 35 unused shades of gray (out of 256) in the familiar, raw format, `baboon` image of the same resolution.

On the other hand, the same images in color (in raw format, with three bytes per pixels) use all the 256 byte values and it is easy to see why. A typical color image may consist of 6–7 million pixels (this is typical for today's digital cameras) and may use only a few thousand colors. However, each color occupies three bytes, so even if a certain color, say (r, g, b)=(108, 56, 213), is unused, there is a good chance that some pixels have color components 108, 56, or 213.

Thus, Gage's method makes sense for compressing text and grayscale images. If applied to other types of data files, a large file should be chopped into small sections with unused byte values in each.

At any given time, the method looks only at one pair of bytes, but this algorithm also indirectly takes advantage of longer repeating patterns. Thus, an input file of the form `abcdeabcdfabcdg` compresses quite efficiently. The most-common byte pairs are `ab`, `bc`, and `cd`. If the algorithm selects the first pair and replaces it with the unused byte `x`, the file becomes `xcdexcdfxcdg`. If `xc` is next selected and is replaced by `y`, the result

is `ydeydfydg`. Now the pair `yd` is replaced by `z`, to produce `zezfzg`. The compression factor is $15/6 = 2.5$, comparable to (or even exceeding) more efficient methods.

The remainder of this section describes a possible implementation of this method (reference [Gage 94] includes C source code). To compress a file, the entire file must be input into a buffer in memory (if the file is too large, it has to be compressed in sections). As an example, we consider an alphabet of the eight symbols `a` through `h` (coded as 0 through 7) and the input file `ababcabcd` (with symbols `e` through `h` unused).

The program constructs the following dictionary (also referred to as a pair-table or a phrase-table), where each zero in the bottom row indicates an unused character.

```
0 1 2 3 4 5 6 7
a b c d e f g h
1 1 1 1 0 0 0 0
```

The first step is to locate the most-common bigram. The simplest approach is to set up a table of size 256×256, initialize it to all zeros, use the two bytes of the next input bigram as row and column indexes to the table, and increment the particular entry pointed to by this bigram. The implementation described in [Gage 94] is based on a hash table that hashes each pair of bytes into a 12-bit number. That number is used as an index to an array of size $2^{12} = 4{,}096$ and the array location pointed to by the index is incremented. This works if the number of bigrams in the data file is less than 4,096 (notice that it can be up to $256 \times 256 = 65{,}536$).

Once the most-common bigram is located (`ab` in our example), it is replaced by an unused character (`h`). The file in the buffer becomes `hhchcd` and the pair-table is updated to

```
0 1 2 3 4 5 6 7
a b c d e f g a
1 1 1 1 0 0 0 b
```

The last entry indicates that byte-pair `ab` has been replaced by `h` (code 7).

The next (and last) pass identifies pair `hc` as the most common bigram and replaces it with unused symbol `g`. The file in the buffer becomes `hggd` and the pair-table is updated to

```
0 1 2 3 4 5 6 7
a b c d e f h a
1 1 1 1 0 0 c b
```

The 7th entry indicates that bigram `hc` has been replaced by `g` (code 6).

The pair-table consists of two types of entries. One type (type 1) has a binary flag in the bottom row indicating used or unused symbols (1 or 0 flags). This type is easy to identify because the element in the top row is identical to its index (thus, the first element `a` had code 0 and is in column 0, while the last element, also `a`, has code 0 but is in column 7). The other type (type 2) indicates pair substitutions and entries of this type should be written on the compressed file. Notice that the two types of entries may be mixed and don't have to be contiguous. In our example, the pair-table consists of six type-1 entries followed by two type-2 entries and is written on the compressed file as the six bytes -6, `h`, `c`, `1`, `a`, `b`, where the first three bytes indicate six irrelevant type-1 entries (corresponding to codes 0 through 5) followed by one type-2 entry `hc`

(corresponding to code 6). The next three bytes indicate one type-2 entry `ab` (which corresponds to code 7). The encoding rule for this table is therefore the following: Each contiguous segment of n type-1 entries followed by a type-2 entry is encoded as $-n$ followed by the two bytes of the type-2 entry. Each segment of m consecutive type-2 entries is encoded as the byte m followed by the $2m$ bytes of the entries.

The last feature of the encoder has to do with replacing pairs of bytes in the buffer. When a pair of bytes `xy` is replaced by a single byte `p`, it becomes `p-`, where the `-` indicates an empty byte. There may be several ways to handle empty bytes. The straightforward way is to move bytes and compact the buffer each time a pair is replaced by a single byte. This is extremely slow because it may involve many thousands of byte movements for each replacement. A better approach is to have an auxiliary array of flags that indicate which byte positions in the buffer are empty. If the size of the buffer is n bytes, the size of the auxiliary array should be n bits, one-eighth (or 12.5%) the buffer size. If the buffer contains the 16 bytes `the␣la-t␣fea-ure`, then the auxiliary array should have the two bytes 00000010|00001000, where the two 1's indicate empty bytes. When the compressed file is written from the buffer to the output, only those bytes that correspond to zero bits in the auxiliary array should be written. Another way to handle empty bytes is to organize the buffer as a linked list and establish another list (initially empty) of empty bytes. When a byte becomes empty, pointers are updated to take this byte out of the buffer and append it to the list of empty bytes.

It is now clear that encoding may not be very efficient, but on the other hand the method never expands the data (except for the overhead from the pair-table). If no byte values are unused, the data is simply written on the compressed file as is, with the addition of the pair-table. This should be compared to other, more efficient methods where the "wrong" type of data may cause significant expansion.

Encoding is slow, requiring multiple passes and a large buffer. Decoding, on the other hand, is fast. The decoder first reads and expands the pair-table, where only the type-2 entries are relevant. Bytes are then read from the compressed file. If a byte is literal (i.e., if it does not appear in the type-2 entries, such as byte `d` in our example), it is written to the final output as is. Otherwise, the byte is one of the type-2 entries and it represents a pair. The pair is constructed and is pushed into a stack. If the stack is not empty, the next byte is popped from the stack and handled as described above (which may cause another byte pair to be pushed into the stack). If the stack is empty, the next byte is read from the compressed file. Being so simple, the decoder is also very small, which is an advantage (as has been mentioned earlier).

Re-Pair, an efficient offline dictionary algorithm

The performance of BPE depends on the number of unused byte values in the original data. The next method (recursive pairing, or Re-Pair, by [Larsson and Moffat 00]) is a much more sophisticated offline compression algorithm. It features high compression factors, a fast, small decoder, and it does not depend on the existence of unused byte values. It also employs sophisticated data structures that make it possible to select the most-common bigram in each pass without having to scan the entire data each time.

No unused byte values are required. In each pass, the most-common bigram is

identified and is replaced by a *new* symbol. We first illustrate this idea symbolically, using the well-known line from the television film *Yabba-Dabba Do!* (based on the 1960–66 animated sitcom *The Flintstones*). The original data is shown in lowercase and the new symbols are in uppercase.

Pair	String
	yabba␣dabba,␣yabba␣dabba␣dabba␣do
A → ba	yabA␣dabA,␣yabA␣dabA␣dabA␣do
B → ab	yBA␣dBA,␣yBA␣dBA␣dBA␣do
C → BA	yC␣dC,␣yC␣dC␣dC␣do
D → C␣	yDdC,␣yDdDdDdo
E → Dd	yEC,␣yEEEo
F → yE	FC,␣FEEo

The compressed file consists of the final string FC,␣FEEo and the six-entry dictionary (or phrase-table). Notice that the final string contains one bigram, but replacing it with a new symbol would require an extra dictionary entry and would therefore contribute nothing to the compression (and may even cause expansion).

The main question is how to add new symbols. Assuming that the original data consists of bytes and that all the 256 byte values are used, how can we add new symbols? We can start by placing each 8-bit byte in a pair of bytes (16 bits). Two-byte symbols can have values from 0 to $2^{16} - 1 = 65{,}535$, so there is room for the original 256 byte values and 65,280 new symbols. The bigram replacement algorithm is then executed and a phrase-table is constructed. Once this is done, the algorithm knows how many new symbols are needed, and the symbols (the 256 original ones and the new ones) are replaced with variable-length codes.

One way to assign reasonable variable-length codes to the symbols is to count the number of occurrences of each symbol in the final string and in the phrase-table and use this information to construct a set of Huffman codes. The table of Huffman codes becomes overhead that must be included in the compressed file for the decoder's use.

An alternative is to sort the list of symbols according to their frequencies of occurrence and assign them one of the many variable-length codes for the integers, such as Golomb, Rice, or Elias. The most-common symbols are assigned the shortest codes and the overhead in this case is the sorting information. This information—which is a permutation of the symbols, each represented by its variable-length code—must be included in the compressed file, for the decoder's use.

In either case, the compressed file consists of three parts, the final string (where each symbol is represented by its variable-length code), the phrase-table (which must also be compressed), and the overhead.

The Nakamura Murashima Method

The Nakamura Murashima method ([Nakamura and Murashima 91] and [Nakamura and Murashima 96]) also employs new symbols, but encodes the phrase-table differently. The authors propose two variants, dubbed SCT and SED. The former is

straightforward. Given the 4-symbol alphabet `a`, `b`, `c`, and `d` and the source data string `acabbadcaddbaaddcaaddb`, SCT encodes it as follows:

Pair	String
	`acabbadcaddbaaddcaaddb`
`A` → `ad`	`acabbAcAdbaAdcaAdb`
`B` → `Ad`	`acabbAcBbaBcaBb`
`C` → `ca`	`aCbbAcBbaBCBb`
`D` → `Bb`	`aCbbAcDaBCD`

SCT encodes the phrase-table in a simple way and prepends it to the output. It generates a string that includes two symbols for each table entry. The names of the new symbols are not included in this string and are assumed to be `A`, `B`, `C`, and so on. Thus, our phrase-table is encoded as the 8-symbol string `adAdcaBb` and the complete encoder output is the string `adAdcaBb`|`aCbbAcDaBCD`, where the vertical bar is a special separator symbol. The decoder reconstructs the phrase-table easily. It reads pairs of symbols, assigns the first pair to the new symbol `A`, the second pair to new symbol `B`, and so on until it reaches the separator.

The SED method is more complex. It constructs a different phrase-table and its output is a string of (original and new) symbols where flags distinguish between symbols and phrase-table entries. Each symbol in the output string is preceded by such a flag (a bit). A single symbol (either from the original alphabet or a new symbol) is preceded by a zero, while a phrase-table entry is preceded by a 1. Thus, the already-familiar input string `acabbadcaddbaaddcaaddb` becomes 0`a`10`c`0`a`0`b`0`b`10`a`0`d`0`c`110β0`d`0`b`0`a`0γ0α0δ (where Greek letters stand for the new symbols). Here is why. The first input symbol `a` becomes 0`a`, but the following pair `ca` appears several times, so it becomes 10`c`0`a`, where the "1" indicates a new symbol, α. The next pair `bb` appears only once, so it becomes 0`b`0`b`, but the following pair `ad` repeats several times, so it becomes 10`a`0`d`, where the "1" indicates the next new symbol β. The single `c` the follows becomes 0`c`. Notice that the encoder does not process the pair `ca`, because the `c` is followed by the pair `ad`, which has already been replaced by β. Thus, the next substring `addb` becomes β`db`, and then γ`b` (where the new symbol γ is `add`), and finally the new symbol $\delta = \gamma$`b` = β`db` = `addb`. This is encoded as the string 1(10β0`d`)0`b` (without the paretheses). The rest of the encoding is easy to follow.

There remains the question of how to write a hybrid string (with bits and symbols mixed up) such as 0`a`10`c`0`a`0`b`0`b`10`a`0`d`0`c`110β0`d`0`b`0`a`0γ0α0δ on the output. The authors mention three different methods that employ adaptive arithmetic coding and complete binary trees, but no details are given.

References:

Gage, Philip (1994) "A New Algorithm for Data Compression," *C/C++ Users Journal*, **12**(2)23–28, Feb 01. This is available online at `http://www.ddj.com/cpp/184402829;jsessionid=LGSEIODZNDHKIQSNDLRSKHSCJUNN2JVN?_requestid=927467`.

Greek-Unicode (2007) is `http://www.unicode.org/charts/PDF/U0370.pdf`.

Larsson and Moffat (2000) "Off-Line Dictionary-Based Compression," *Proceedings of the IEEE*, **88**(11)1722–1732. An earlier, shorter version was published in *Proceedings of the Conference on Data Compression* 1999, pages 296–305. An implementation is available at `http://www.bic.kyoto-u.ac.jp/pathway/rwan/software/restore.html`

Nakamura, Hirofumi and Sadayuki Murashima (1991) "The Data Compression Based on Concatenation of Frequentative Code Neighbor," *Proceedings of the 14th Symposium on Information Theory and its Applications (SITA '91)*, (Ibusuki, Japan), pp. 701–704, December 11–14 (in Japanese).

Nakamura, Hirofumi and Sadayuki Murashima (1996) "Data Compression by Concatenations of Symbol Pairs," *Proceedings of the IEEE International Symposium on Information Theory and its Applications*, (Victoria, BC, Canada), pp. 496–499, September.

I would like to acknowledge the help given me by Giovanni Motta and Hirofumi Nakamura in the preparation of this document.

David Salomon, `dsalomon@csun.edu`, November 2007