LOSSLESS DATA COMPRESSION

by: Rukako

01 INTRODUCTION

Computers are amazing: you can automate your work, compute like you had a thousand years of free time, and store a lot of data. However, the space to store all this data is unfortunately limited. What happens when you run out of space? Buy a new disk? Remove some of your precious information? No, you need to compress your files.

Many files are already compressed — like pictures or music — but the algorithms used to compress them are not suitable to handle text files or executables. The reason is simple: these are lossy compression algorithms. This means that it loses a bit of information in order to gain disk space. In the case of pictures, you can encode pixel colors using fewer bits (e.g. 24 instead of 32) or modify the image in order to make very similar colors the same. Then you can group them together and encode the color once for the whole group.

Lossy algorithms are not suitable to compress text files or executables since you don't want a word or an instruction to be changed in the process. To compress these files you need a lossless data-compression algorithm.

01.01 KEEP THE SAME DATA USING LESS SPACE

Lossless compression algorithms are fascinating. Large parts of these algorithms come from very simple ideas and end up very simple to implement. For example, a common idea is that data often repeats itself. Did you count how many times the words "you" and "compress" are written in the introduction? Great, you get the point! The core idea is to reduce the space taken by these repetitions of the same piece of data. In the second section of this article, I will present you an algorithm named LZ77 that allows you to avoid repetitions and to use back-references instead.

It is also possible to change the size of the symbols composing the input in order to reduce significantly the memory space taken by the most frequent symbols. To illustrate this, I will show you the famous Huffman coding in the third section of the article.

02 SAME-WORD-REPETITION REPLACEMENT

In the previous section, I asked you how many times the words “you” and “compress” were written. Obviously, they are not written only once, but a fair number of times. This means the total space taken by every “compress” is the number of “compress”es multiplied by the word length: 8 times the total! What if we wrote this word once and then only write a short reference to it, instead of the whole word? What if we applied the same logic to the whole text? We would end up with a text that contains only one occurrence of each word and then only references to the first time it appears. Here is an example:

The sky is blue. The ocean is blue. The trees are green. The grass is green. But sand is neither blue nor green.
Becomes something like:

```
The sky is blue. !0 ocean !2 !3. !0 trees are green.
!0 grass !2 !7. But sand !2 neither !3 nor !7.
```

This example is a very repetitive text I wrote on purpose. Nonetheless, texts that you want to compress are much longer, which increase greatly the chances to introduce new repetitions. For instance, source code is very repetitive: C code contains a lot of occurrences of the same typenames, variables are used more than once, preprocesssor directives repeat, et cetera. My point is that the space we gained in my small example has nothing in common with the space you may gain compressing a much longer input.

02.01 INTRODUCING DICTIONARIES

In a perfect world, the output of our compression scheme would be 100% references. But in practice, we need at least one occurrence of each word in order to keep the value behind our short references. The set of original values is called the dictionary.

Until now, I used words as the basic unit of our input data for the sake of simplicity. However, real-life algorithms do not split the data into words, they operate on sequences of bytes. This way, we can compress any kind of data using a dictionary-based algorithm. There is another advantage to this: we don’t need to match entire words anymore. The algorithm will try to find patterns that are repeated later in the data regardless of the meaning or the structure of the token it manipulates. For example, compress appears in compression, so the algorithm should be able to replace it, resulting in “!0ion”.

Now that you have a good insight of how dictionary-based compression behaves, we will study the algorithm LZ77 which is one of the simplest algorithms of its kind.

03 LEMPEL-ZIV 1977

LZ77 is an algorithm created in 1977 by Abraham Lempel and Jacob Ziv. It features a sliding window as dictionary to compress the input data. This means that the dictionary is not the same for the entire data. This sliding window is composed of two parts: a search window and a lookahead buffer. Figure 3 shows the spacial structure of the sliding window.

When the algorithm begins, it places the sliding window at the beginning of the input data stream. The goal of the algorithm is to find matches between the search window and the lookahead buffer. It finishes when the sliding window reaches the end of the stream and the lookahead buffer ends up empty.

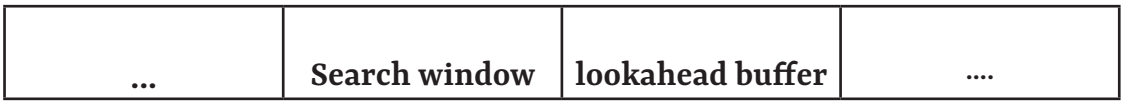

Figure 1: LZ77 sliding window

The lookahead buffer contains the sequence of bytes to be matched in the search window. The algorithm must find the longest prefix of the buffer inside the search window. If a prefix of the buffer appears in the search window, then we have a repetition. The reason why the algorithm explicitly needs a prefix of the lookahead buffer is the need of a deterministic behavior of the algorithm. Otherwise, the decompression would be implementation-dependent.

In the case of a match, the selected prefix is replaced by a triplet (D, L, b) where D is the backward distance, L is the length of the prefix and b is the first byte right after said prefix in the lookahead buffer. The backward distance is the distance in number of bytes between the prefix of the buffer and its match inside the search window. Otherwise, in the lack of a match, the triplet (0, 0, b), where b is the first byte of the lookahead buffer, is generated. In fact, this triplet is a special case of the (D, L, b) triplet. It features a length of 0, a useless backward distance (0 by convention) and the next byte. When the triplet is emitted, the window can move L + 1 bytes ahead – the prefix length plus the next byte also stored in the triplet. This way, the prefix becomes part of the search window and the lookahead buffer can match further repetition against it.

3.1 EFFICIENCY MATTERS

This algorithm is conceptually pretty good, but does it work? Well, usually the sizes of both the search window and lookahead buffer are big – some kilobytes – so the chance to find repetitions is very high. However, the real pitfall of LZ77 is that the repetitions have to be relatively close to each other in order to appear together in the sliding window. In the case of a very long file with huge but very distant repetitions, this algorithm alone may not be able to achieve an acceptable compression rate.

But fear not! hope is not lost. You can transform your input stream before using a dictionary-based algorithm. Transforms such as the Burrows-Wheeler help pack similar bytes together, forming big chunks of bytes repeating themselves. Other transforms, like the move-to-front transform, transform a sequence of similar bytes into a sequence of zeros. With these two transforms, the input stream ends up being a huge collection of sequences of zeros separated by some less important bytes. This configuration is nearly optimal for an algorithm like LZ77.

An algorithm named LZMA — and more recently LZMA2 — that is based on LZ77 is used alone in 7z and xz archives. The two main reasons why it has replaced LZ77 are the humongous size of its dictionaries and a powerful range encoder that contributes a lot to the compression rate. LZMA is now among the lossless algorithms that give the best compression rates.