Understanding Lossless File Compression (.zip)

Author	josh
Date	2023-04-18

Overview

File compression is a common task in computer science. We computer users generate *massive* amounts of data. And although storage is cheap, it is good to have clever tools for reducing the size of data on disk or in transit. Reducing storage and bandwith costs for text data is usually done with *lossless* compression algorithms. Let's learn about the general principals for these algorithms.

Lossless Compression Algorithms

General Principals

The first part of understanding *lossless* compression is understanding what *lossless* means in this context. A compression algorithm is said to be lossless when all of the original data is retained when the file is decompressed. A lossy algorithm cuts out data in such a way that is imperceptible to the viewer, for example, in an image file. A JPEG is an example of a lossy compression algorithm – where some image data is cut out, but not so much that the user notices a difference. Lossless algorithms, on the other hand, *compress* down the data in such a way that the file can later be *decompressed* to its exact original state. These algorithms are most commonly used for text data formats – such as spreadsheets, text documents, etc. where it's important all the original information is accessible.

So how can we make a file smaller while retaining all the information? At their core, these algorithms use *pattern recognition* and substitutions. By recognizing patterns in our data and swapping in a smaller representation of the information, we can reduce the file size without losing data. For example, the word "the" could be swapped for the character "1". We have to store the dictionary as well – a table of our substitutions such as "the == 1". But given that we will likely see the word "the" hundreds or thousands of times in a large text document, the 2 character reduction adds up! Let's look at a more concrete example next to fully understand this concept.

An Example Compression Algorithms

Let's say we have a text document database that stores a record of students at a jiu jitsu school and their belt rank. Our table looks like this:

```
John Smith     Blue Belt
John Adams     White Belt
Jane Richards  Purple Belt
Sally Fields   Blue Belt
```

How can we apply a simple pattern recognition and compression algorithm to this? Let's look for repeated words, and swap those out with a simple integer index. We'll then store our dictionary in the file so we can decompress later. For each repeated word, we'll pick the next index and store it in the lookup table. To decompress, we simply reverse the process using out dictionary.

```
0 Smith        2 1
0 Adams        White 1
Jane Richards  Purple 1
Sally Fields   2 1
***
John           (0)
Belt           (1)
Blue           (2)
```

Now in this very small example, we end up having to store close to, if not more, data in our file – since we have to store our swapped-out data *and* the compression dictionary. But we must think about the overall gain for a larger database. Swapping "John" for "0" is a 3 character reduction. Swapping "Belt" for "1" is also a 3 character reduction. For every "Belt" occurrence in a database of 100 students, we save quite a few bytes!

Swap and Store Less!

This is the crux of lossless algorithms. The more *repetition* in the original data, the higher *compression ratio* we can achieve. Using typical lossless algorithms on images, for example, usually does not yield useful levels of data reduction (which is why we use lossy algorithms instead). In a typical text document, there's a lot of repetitive words, leading to higher compression ratios. Our simple pattern-recognition is pretty naive, but sophisticated algorithms like those found in the popular ".zip" format do a great job at reducing data size on disk and in transit. Some simple swaps can really save!