Data Compression Using Long Common Strings

Jon Bentley
Bell Labs, Room 2C-514
600 Mountain Avenue
Murray Hill, NJ 07974
jlb@research.bell-labs.com

Douglas McIlroy
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755
doug@cs.dartmouth.edu

1. Introduction

White [1967] proposed compressing text by ‘‘replacing [a] repeated string by a reference to [an] earlier occurrence’’. Ziv and Lempel [1977, 1978] implemented this idea by cleverly representing strings that occur in a relatively small sliding window. We extend the basic idea to represent long common strings that may appear far apart in the input text.

On typical English text, our method provides little compression; there are few long common strings to be exploited. Some files, though, do contain repeated long strings. Baker [1995] documented significant repetition in the code of large software systems. In a mathematical subroutine library, we found many blocks of code repeated across functions of type `float`, `double`, `complex`, and `double complex`; our method combined with a standard compression algorithm reduced the file to half the size given by the standard algorithm alone. Corpora of real documents, such as correspondence, news articles, or netnews often contain long duplications due to quoting or republication, or even plagiarism.

We begin by illustrating the opportunity for compressing very long strings. We then survey Karp and Rabin’s [1987] algorithm for string matching, and apply it to data compression. Experiments show the efficacy of the new method on some classes of input, and analysis shows that it is efficient in run time.

2. The Opportunity

Data compression schemes usually employ a sliding window, typically a few kilobytes long. Repetitions of strings are discovered and compressed only if they both appear in the window. This approach is efficient in bits to encode a position, in the space to store the window, and in the time required to search the window. Furthermore, the resulting dynamic codes reflect the locality intrinsic in many documents.

Sliding windows do, however, present a significant drawback: they do not find repeated strings that occur far apart in the input text. An artificial experiment illustrates this point. We take two text files, the Constitution of the United States and the King James Bible, and compress both the original text and the text concatenated with itself. We use the GNU `gzip` implementation of LZ77 as a typical compression algorithm.

File	Text	gzip	Relative compressed size
Const	49523	13936	1.0
Const+Const	99046	26631	1.911
Bible	4460056	1321495	1.0
Bible+Bible	8920112	2642389	1.9995

While doubling adds almost no information, the doubled documents compress to nearly double the size of the original documents. A major opportunity has been missed, because the compression window no longer contains the first string.

We propose a data compression scheme that recognizes the second occurrence of the input text as a repetition. It then represents the second string with a reference to the first, using just a few bytes. Some compression algorithms do recognize such repetitions, such as Cleary and Teahan's [1997] PPM* and Nevill-Manning and Witten's [1997] Sequitur system. Those systems typically require roughly n words of primary memory to process a file of n characters. Our method introduces a block size, b, and then uses approximately n/b words of main memory. Typical values of b are near 100. By increasing b, one can dramatically decrease memory requirements by slightly decreasing the compression efficiency.

We will typically use our scheme in conjunction with a standard compression algorithm. We will first run our ''precompression'' algorithm to find long strings that are far apart, then feed its output into a standard algorithm that efficiently represents short (and near) repeated strings.

3. Finding Long Common Strings

One could use many different algorithms to search for long common strings in a text file. Obvious candidates include McCreight's [1976] suffix trees and Manber and Myers's [1990] suffix arrays. We will in fact use Karp and Rabin's [1987] method of fingerprints.

Karp and Rabin originally presented fingerprints as an aid to string searching: does a string of length n contain a search pattern of length m? Karp and Rabin interpret the m characters of the pattern as a polynomial modulo a large prime number; the resulting fingerprint can be stored as a 32-bit word. Their algorithm scans down the input string, computing the same fingerprint for each of the $n-m+1$ substrings of length m. If the fingerprints do not match, then we are certain that the substring does not match the pattern. If the fingerprints do match, then we check to see whether the substring in fact matches the pattern.

Karp and Rabin prove several useful properties of their fingerprints. They can be computed quickly: a fingerprint can be initialized in $O(m)$ time and updated by sliding one position in $O(1)$ time. Fingerprints yield few false matches: unequal strings are extremely likely to have unequal fingerprints. The probability of two unequal strings having the same 32-bit fingerprint is near 2^{-32}. Furthermore, one can choose the large primes at random to yield randomized algorithms for text searching. Technical details on these properties may be found in Karp and Rabin's original paper and in many standard algorithms texts.

Our compression algorithm uses a single parameter b, which is the compression ''block size''. Typical values of b will be between 20 and 1000. Ideally, we would like to assert that we ignore repeated strings of length less than b, and represent common strings longer than b. Our algorithm will instead make the weaker claim that it ignores repeated strings of length less than b, and discovers all repeated strings with length at least $2b-1$. String with length between b and $2b-1$ may or may not be represented.

Our primary data structure stores the fingerprint of each (non-overlapping) block of b bytes. That is, we store the fingerprint of bytes $1 \cdots b$, $b+1 \cdots 2b$, $2b+1 \cdots 3b$, and so forth. In a file of length n, our method stores approximately n/b fingerprints. We represent them in a hash table, together with an integer giving the sequence's location in the input text. As we scan through the input text, we will use the hash table to find common fingerprints and thereby locate common strings.

4. The Compression Algorithm

Because we represent only long common strings, we are free to use an inefficient representation. We will represent a repeated string by the sequence ''<*start*,*length*>'', where *start* is the initial position and *length* is the size of the common sequence. For instance, the Constitution begins:

> The Constitution of the United States PREAMBLE We, the people of the United States, in order to form a more perfect Union, ...

Its compressed form begins:

> The Constitution of the United States PREAMBLE We, the people <16,21>, in order to form a more perfect Union, ...

Literal ''<'' characters are quoted as ''<<''.

The main loop of our algorithm processes every character by updating the signature and checking the hash table for a match. Every b characters, it stores the fingerprint. If we let the variable `fp` represent the fingerprint, we can express the loop in pseudocode as:

```
initialize fp
for (i = b; i < n; i++)
    if (i % b == 0)
        store(fp, i)
    update fp to include a[i] and exclude a[i-b]
    checkformatch(fp, i)
```

The `checkformatch` function looks up `fp` in the hash table and encodes a match if one is found.

Exactly what do we do when we find a match? Suppose for concreteness that $b=100$ and that the current block of length b matches block 56 (that is, bytes 5600 through 5699). We could encode that single block as <5600,100>. This scheme is guaranteed not to encode any common sequences less than b. If a block is at least $2b-1$ long, though, at least one subsequence of b characters will fall on a block and be discovered.

We in fact implemented a slightly more clever matching scheme. After checking to

ensure that the block with matching fingerprints is not a false match, we greedily extend the match backwards as far as possible (but never more than $b-1$ characters, or it would have been found in the previous block) and forward as far as possible. If several blocks match the current fingerprint, we encode the largest match among them. These small examples illustrate our algorithm with block size $b=1$.

Input	Output
abcdefghijklmnopq<12345	abcdefghijklmnopq<<12345
abcdefghijabcdefghij	abcdefghij<0,10>
abcdefghijklmnopqrstuvwxijklmnopabcdefghqrstuvwx	abcdefghijklmnopqrstuvwx<8,8><0,8><16,8>
aaaaaaaaaaaaaaaaaaaa	a<0,20>

We were surprised and delighted to see the last line: our method encodes a run as a block that matches itself, shifted by one.

For a larger example, we return to the Bible concatenated with itself:

Compression	Bible	Bible+Bible
Input	4460056	8920112
gzip	1321495	2642389
com 50	4384403	4384414
com 20	3906771	3906782
com 50 | gzip	1318687	1318699
com 20 | gzip	1362413	1362422

For definiteness, we call the program that implements our algorithm `com`, and its single parameter is the block size `b`. We'll first study the middle column, which shows the performance of the compression algorithms on the Bible itself. By itself, `com` 50 provided a slight decrease in size, while `com` 20 gave a 12.4% reduction. When used as a precompressor, though, `com` 50 gave a slight overall reduction, while `com` 20 interacted badly with `gzip` to increase the file size. The right column, however, shows that `com` is incredibly effective when the file contains a huge repeated string (itself). The doubled file is represented in just eleven additional bytes, and `gzip` works as before.

Our compression program is implemented in about 150 lines of C, and the decompressor requires just 50 lines.

5. An Experiment: One File In Detail

Our simple experiment with doubling the Bible verified the obvious: `com` is very effective when the input data contains long repeated strings. But does real data in fact contain such strings?

Our next experiment concatenated all text files in the Project Gutenberg Compact Disc [1994] using this Unix command:

```
cat */*.txt >gut94all.txt
```

The files represented documents such as the Constitution, the Declaration of Independence, inaugural speeches of several presidents, fiction such as *Alice in Wonderland* and *O, Pioneers!*, and so forth. This table shows the result of applying `com` 1000, `gzip`, and both to the resulting file:

	Input		gzip
Input	66.122	34.64%	22.905
	85.68%		86.09%
com 1000	56.653	34.81%	19.721

The file sizes are given in megabytes, and the percentages show the effectiveness of each compression. By itself, `com` 1000 reduces the file to about 86% of its original size, while `gzip` reduces it to about 35% of its original size. If we apply `gzip` after applying `com` 1000, though, `gzip` is almost as effective as before: the two methods appear to factor. Our precompression algorithm compressed three different kinds of long common strings that went unnoticed by `gzip`:

Stern legal boilerplate was repeated at the front of each document.

finding repeated text in well-structured packages. We investigated the file he mentioned, and found it to be rich with duplication:

b	com b	com | gzip
∞	29.37	4.04
1000	24.38	3.34
500	20.24	2.88
200	12.79	2.21
100	8.84	1.92
50	6.50	1.84
20	5.66	2.09

File sizes are shown in megabytes. At the optimal block size $b = 50$, the 1.84 megabytes of the resulting file is a 55% reduction over the 4.04 megabytes of the file after `gzip`. Further investigation showed that many blocks of code were repeated across functions of type `float`, `double`, `complex`, and `double complex`.

Our next set of experiments dealt with a variety of files. The files included the ‘‘`.cab`’’ binaries of the Windows 95 operating system distribution, the `setup.exe` for AOL 3.0 for Windows 95, the `DIFFPACK` package for differential equations, and a set of Associated Press stories. A 1994 AT&T 800 number directory included lines like:

```
8002000887 Turf Mktg       Boise ID       GR0350 Grass - Artificial
8002002199 Quality Craft   Anaheim CA     IM0050 Importers
8002002499 Leather Factory Phoenix AZ     LE0150 Leather
8002002566 Floral Essence  San Diego CA   FL0750 Florists - California
8002005225 Iraqi Jack Inc  Canoga Park CA GU0150 Guns & Gunsmiths
```

The block size *b* is the best for the file, and file sizes are shown in megabytes:

File	Opt b	Input	gzip	com	com | gzip
Win 95 Binaries	20	33.286	33.217	33.399	33.353
AOL 3.0 Setup	200	11.910	11.745	11.419	11.278
800 Directory	20	12.157	4.089	7.528	3.554
Diffpack	50	7.526	1.162	2.624	0.658
AP Stories	20	1.241	0.444	0.766	0.319

Our algorithm found no long common strings in (the already compressed) Windows 95 binaries; indeed, quoting the ‘‘<’’ character increased the file length. While `gzip` alone squeezed about 1.5% from the AOL binaries, our algorithm found an extra 4%. Our algorithm was more effective on the other inputs.

For our final example, we felt obligated to examine the ‘‘Canterbury corpus’’ of Arnold and Bell [1997]. We retrieved the file `cantrbry.tar.gz` from the North American mirror site `ftp://dna.stanford.edu/pub/canterbury/`, and uncompressed it to form the input file `cantrbry.tar`. Before we present the results, we encourage the reader to guess: will our precompression algorithm find long common strings in that file?

b	com b	com	gzip
∞	2821120	739057	
1000	2723911	742258	
500	2704698	742287	
200	2643714	742448	
100	2514569	742620	
50	2445013	744447	
20	2396298	752818	

When we studied the file, we found that the majority of common strings were runs of binary zeros used by the `.tar` format and the Excel spreadsheet. Many of the remaining common strings occurred in the same sliding window, and were therefore discovered by `gzip`. Our simple coding of the sequences was much less efficient than that used by `gzip`, and therefore increased the file length.

7. Algorithmic Considerations

How fast is our algorithm? If the input contains no long matches, then our algorithm will spend linear time to compute fingerprints. Karp and Rabin prove that there will be relatively few false matches to check, and they will be identified as mismatches after a few comparisons.

If the file contains a few long matches, we can amortize the additional time required to check the matches against the reduction in writes due to compression.

Although we have not analyzed the worst-case behavior of our algorithm, we have constructed a pathological input that drives it to use $\Theta(N^{3/2})$ time. The input consists of $\sqrt{N}$ blocks of ones, each of length $\sqrt{N}$, separated by random strings of zeros and ones of length $2\log_2 N$. On all realistic data we have tried, though, the algorithm runs in close to linear time. Karp and Rabin [1987] use randomization to increase the speed of their text searching algorithm. Their techniques might be used to increase the expected performance of our compression scheme.

Our implementation of the algorithm stores the entire input file in main memory. An alternative implementation keeps the input file on disk, and uses roughly N/b 32-bit words of main memory to store the fingerprints. For typical files, the total number of bytes read and written sums to approximately $2N$.

8. From Prototype to Practice

We believe that our small experiments with the prototype program establish the feasibility of this compression scheme. When we started our experiments, we felt that long common strings would not occur frequently in ‘‘well-structured’’ file systems. We have been surprised to find long common strings in a variety of contexts, and in a manner unrecognized by most current compression algorithms.

The prototype compressor and decompressor might be useful in certain applications. Much work, however, stands between this prototype and a production tool.

We currently use the simplest possible <*start*,*length*> representation of a common sequence: both numbers are represented in decimal. This portable representation forces the quotation of ‘‘<’’ characters and uses 14 bytes to represent the sequence <1000000,1000>. This was a small price to pay for large blocks, but rendered small

block sizes (say, $b = 15$) ineffective. A production version of the algorithm should use a more sophisticated representation: a one-byte escape character followed by a three-byte *start* and a two-byte *length*, for instance, squeezes the representation from 14 bytes to 6.

Our prototype program reads its standard input and writes on its standard output. A production version would have a more powerful interface and would address issues inherent in real file systems (such as representing many files in subdirectories).

We mentioned in the previous section that our algorithm can be implemented using very little primary memory if the files to be encoded and decoded may be read from a disk. This might be useful in a production program.

Future experiments should measure how the precompression algorithm interacts with a variety of compression algorithms. Such experiments might provide insight into whether and how to represent a particular common sequence.

9. Conclusions

We have described a precompression algorithm that effectively represents any long common strings that appear in a file. The algorithm interacts well with standard compression algorithms that represent shorter strings that are near in the input. Our experiments show that some real data sets do indeed contain many long common strings.

We have extended the fingerprint mechanisms of our algorithm to a program that identifies long common strings in an input file. This program has given interesting insights into the structure of real data files that contain long common strings.

Many problems remain open. The previous section sketched several problems that must be addressed to build a useful compression program. Further experiments should investigate how our compression algorithm behaves on additional representative data sets. Open algorithmic problems include finding a compression algorithm with optimal worst-case run time, and finding compression patterns more effectively than our greedy method (although Storer and Szymanski [1982] prove that the general problem is NP-hard).

References

Arnold, R. and T. Bell [1997]. ‘‘A corpus for the evaluation of lossless compression algorithms’’,

McCreight, E. M. [1976]. ‘‘A space-economical suffix tree construction algorithm’’, *Journal of the ACM 23*, 12, pp. 262-272.

Nevill-Manning, C. G. and I. H. Witten [1997]. ‘‘Identifying hierarchical structure in sequences: a linear-time algorithm’’, *Journal of Artificial Intelligence Research 7*, September 1997, pp. 67-82.

Project Gutenberg [1994]. *Project Gutenberg CD ROM*, Walnut Creek CDROM, Walnut Creek, California.

Storer, J. A. and T. G. Szymanski [1982]. ‘‘Data compression via textual substitution’’, *Journal of the ACM 29*, 4, pp. 928-951.

White, H. E. [1967]. ‘‘Printed English compression by dictionary encoding’’, *Proceedings IEEE*, 55, 3, pp. 390-396. (Cited by Bell, Cleary and Witten [1990, pp. 214 and 242].)

Ziv, J. and A. Lempel [1977]. ‘‘A universal algorithm for sequential data compression’’, *IEEE T. Information Theory IT-23*, 3, 337-343.

Ziv, J. and A. Lempel [1978]. ‘‘Compression of individual sequences via variable-rate encoding’’, *IEEE T. Information Theory IT-24*, 5, 530-536.