Data Compression Project - LZ77

Breno Fatureto

I. Abstract

We describe the implementation of a LZ77-based file compressor in Swift. Thereafter, the performance of the compressor is compared with the widely available `gzip` utility, which also uses a technique based on the DEFLATE algorithm.

II. Introduction

LZ77 (Named after Abraham Lempel and Jacob Ziv) is a compression technique which substitutes repeated sequences of symbols with pointers to the previous occurring sequence.

The algorithm consists of, given an input s, back-buffer and lookahead-buffer-sizes b and f, respectively, and a current position i, look for the longest prefix of $s[i \ldots]$ occurring in $s[i-b \ldots i-1]$ limited to length f. This information is encoded in a triple $< C, l, d >$, where l is the length of the match, d is the offset from i where the match starts, and C is the first character ocurring after the match ends.

Since we substitute pieces of the input for pointers indicating where the fragment occurs in a buffer, LZ77 is classified as a *dictionary technique* [Sayood].

The algorithm here described is implemented in the Swift programming language. The project is available at GitHub. Usage instructions are available in the project's readme.

III. Implementation

A. Compression

The program reads the input file as an array of bytes. Then we execute the LZ77 encoding step. Parameters of the compressor (backbuffer size, lookahead buffer size, minimum/maximum match length) can be configured on runtime.

Prefix lookup is done by, first, cataloguing the position of each byte in a hash table. Then, we find the index of each occurrence of the current byte in the backbuffer by looking it up in the hash table. Finally, we find the longest match by scanning linearly starting from each index found. This technique is only efficient for reasonably short match lengths.

The LZ77 encoder outputs a binary file containing information of each triple. This output is then encoded using a Huffman tree encoder.

B. Decompression

The first stage of decompression is taking the Huffman-encoded file and decoding it. The output of this procedure is then fed into the LZ77 decoder. This decoding process involves processing each triple and outputting a piece of the decompressed file.

Given a current buffer s and a position i, decoding a triple $< C, l, d >$ is done by taking the substring $s[i-l...i-l+d]$ and appending it, as well as the byte C, to the buffer. This is done until there are no triples left.

IV. Evaluation

We evaluate this implementation by comparing it with the GNU `gzip` utility commonly available in Unix systems. This utility is used as a baseline since it implements a compression scheme based on LZ77, along with a Huffman encoding stage. The extracted data is presented in the 'Tables' section (VI)

We measure the original file size and compressed file size in bytes as reported in `ls`. We then compute the compression ratio obtained by dividing the compression file size by the original file size.

Despite using a similar algorithm, `gzip` obtained much better results overall. This can be explained by the choices in the buffer parameters used in the tool. While our compressor utilizes buffer sizes of up to 255, `gzip` uses a 32KB search buffer, along with a maximum match length of 258 bytes. Using such parameters in our implementation would result in unreasonable performance, since the prefix-lookup algorithm would have to be changed to accommodate such large search space. In fact, this limitation in buffer lookup makes the LZ77 compressor perform worse in some cases than the standalone Huffman encoder.

We can also see in the histogram of lengths and offsets that, for some files, triples with length 0 vastly outnumber any other lengths. This is worsens the compression rate significantly, since any gains with longer

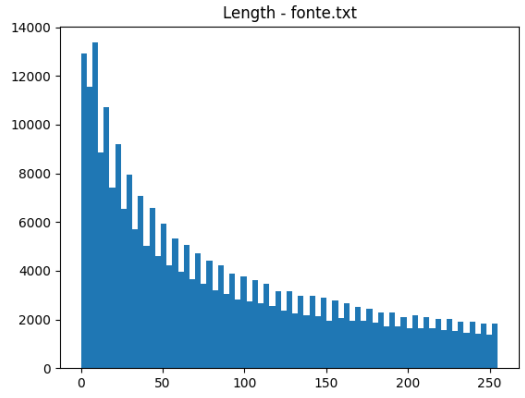

Fig. 1. Histogram of prefix lengths - fonte.txt

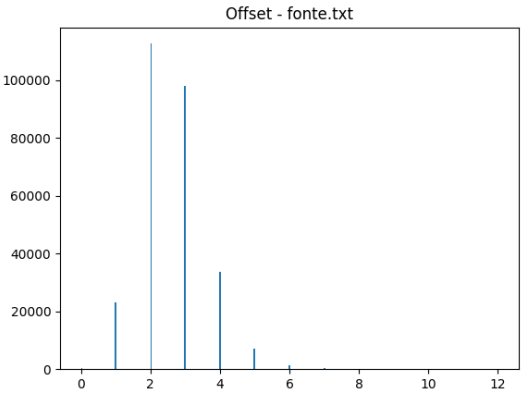

Fig. 2. Histogram of offsets - fonte.txt

prefixes may be offset by a large number of 0-length triples.

The reader may notice something odd going on with the file `fonte0.txt`. Although the LZ77 compression stage yields a 652 byte file, the Huffman encoder inflates this file's size to 2280 bytes.

V. Conclusion

We present an implementaion of a LZ77-based compressor in Swift. Although the compressor achieves reasonable compression for some files, the performance falls far below what is found in the `gzip` compressor, and still performs worse than a standalone Huffman encoder. A possible explanation is that this happens due to constraints involving the backbuffer size, which needs to be small to make prefix lookup viable. The main possible improvement is implementing a more efficient prefix lookup technique, similar to the one found in `gzip`.

VI. Tables

The *Size* column indicates the original file size. The *Compressed* column indicates the compressed file size. File sizes are reported in bytes.

TABLE I
LZ77 evaluation

Filename	Size	Compressed	Ratio
TEncEntropy.txt	19415	15150	0.7803
TEncSearch.txt	253012	139700	0.5521
domcasmurro.txt	389670	336679	0.8640
fonte.txt	1000000	689558	0.6896
fonte0.txt	1000	2280	2.2800
fonte1.txt	1000000	761677	0.7617

TABLE II
gzip evaluation

Filename	Size	Compressed	Ratio
TEncEntropy.txt	19415	4225	0.2176
TEncSearch.txt	253012	38677	0.1529
domcasmurro.txt	389670	153735	0.3945
fonte.txt	1000000	428408	0.4284
fonte0.txt	1000	322	0.3220
fonte1.txt	1000000	489795	0.4898

TABLE III
Huffman-only evaluation

Filename	Size	Compressed	Ratio
TEncEntropy.txt	19415	14356	0.7394
TEncSearch.txt	253012	165855	0.6555
domcasumurro.txt	389670	219091	0.5622
fonte.txt	1000000	368845	0.3688
fonte0.txt	1000	292	0.2920
fonte1.txt	1000000	416105	0.4161