LZ78 Compression Algorithm

Stanford EE274 Data Compression course project report

Nan Wu

Introduction

This report presents the LZ78 compression algorithm, detailing its functionality and practical implementation in Python. Additionally, it analyzes the performance of this implementation against the Stanford Compression Library's LZ77 algorithm.

We have discussed the LZ77 compression algorithm and its practical realization in depth during our class. To recap, LZ77 keeps previously seen inputs in memory, and replaces future repeated occurrences with references to the earlier data stream, hence achieving compression. However, there could be multiple copies of the same subsequences, and finding the longest matches could be a slow operation. We could employ sliding windows (or buffer) to avoid searching too far back in the input to speed up the operation. This approach, albeit effective, makes the selection of the buffer size critical. A smaller buffer results in reduced compression time but demands more space. Conversely, a larger buffer size reduces required space but prolongs the compression time. Consequently, the parameters of LZ77 need to be optimized according to the pattern of the input data.

The LZ78 algorithms, named after its inventors Abraham Lempel and Jacob Ziv with the '78' denoting its year of publication in 1978. Just like LZ77, it is also a lossless data compression algorithm that utilizes a dictionary-based matching algorithm. It forms the basis for several ubiquitous formats, including GIF and TIFF. The primary motivation for developing LZ78 was to create a faster universal compression algorithm that does not require any prior knowledge of the input and choosing of the buffer size, addressing an inherent drawback of LZ77.

Technical Details

Encode

LZ78 identifies and adds phrases to a dictionary. When a phrase recurs, LZ78 outputs a dictionary index token instead of repeating the phrase, along with one character that follows that phrase. The new phrase (the reoccurred phrase plus the character that follows) will be added to the dictionary as a new phrase. The dictionary index and the following character form a tuple, and the tuple is added to a list. The output of a LZ78 compression will be this list of tuples. These tuples are then entropy coded. Compression is achieved by replacing reoccurrences of previously seen phrases with indexes.

Algorithm

Overview

- keep a dictionary of previously seen string patterns to their positions in the output list.
- keep an output list of tuples of the dictionary index of the previously seen phrase and the character that follows. In case there is no match in the dictionary, using 0 as the index.

- to find a match during parsing, we look up future substring in the input and then find the longest match in the dictionary keys. The value of this key is the index of that substring in the output list. The index and one immediate following character forms a tuple which is then put into the output list. The longest matching substring is appended with the immediate subsequent character to form a new substring. This new substring will be the key to a new dictionary entry, whose value will be the index of it in the output list, which is the length of the output list.
- Entropy encodes the output list into byte arrays.

Pseudo code

```
def lz78_compress(data):
    dictionary = {}
    result = []
    pos = 0
    while pos < len(data):
        prefix = data[pos]
        advance = 1
        while prefix + data[pos + advance] in dictionary:
            prefix += data[pos + advance]
            advance++
        result += (dictionary[prefix], data[pos + advance])
        dictionary[prefix + data[pos + advance]] = len(dictionary)
        pos = pos + advance
    return result
```

Detailed implementation can be found here.

Entropy encoding

After the input text has been encoded into a list of tuples of index, character, we then entropy encode the list into byte arrays using the same technique in LZ77. Indexes are first binned in log based 2 scale, then encoding the bin numbers with empirical Huffman coder. The residuals are stored as plain old bits. Characters are first converted into ASCII integers, then encoding using empirical Huffman coder again.

Decode

The decompression mechanism of LZ78 relies on the iterative reconstruction of the dictionary used during the encoding process. The original content is recovered by replacing indexes in the received list with reconstructed substrings.

Algorithm

Overview

- Decodes received byte arrays back to a list of (index, character) tuples.
- The tuples are read sequentially. Each tuple consists of a pair: a pointer to a previous entry in the dictionary (if any) and the next character of input. In the case of the first entry, there is no prior reference, and only the single character is used.
- Reconstructing entries: As each tuple is read, the decompression algorithm performs a look-up in the dictionary using the dictionary index provided in the tuple. If a non zero index is given, the corresponding phrase is retrieved and output. The character that accompanies the index is then added to this phrase to form a new phrase, which is added to the dictionary with the size of the dictionary as its key.

Pseudo code

```
def lz78_decode_from_tuples(tuples):
        dictionary = []
        output = []

        for index, symbol in tuples:
            new_string = symbol
            if index != 0:
                new_string = dictionary[index] + symbol

            dictionary[dict_index] = new_string
            dict_index += 1
            output.append(new_string)
        return output
```

Detailed implementation can be found here.

Performance

As we mentioned earlier in this report, one intention in designing LZ78 was to make a faster version of LZ77. Therefore, it is expected to have a faster compression speed. However, because the string comparison at each step in LZ78 works on a set of fixed substrings, unlike the entire past data used in LZ77, it is expected to have a lower compression ratio. In this section, LZ78 is used to compress and then decompress a variety of files; speed and memory usage are measured for each file. This same operation will then be performed with LZ77, and the two sets of measurements will be compared side-by-side.

The files used in this performance test include three novels, a web page styling file (CSS), an extra-large novel, and a text file consisting of randomly generated characters. These files are sourced from https://github.com/nemequ/squash-corpus, https://corpus.canterbury.ac.nz/descriptions/#cantrbry, and handmade.

File name	raw size	LZ77 size	LZ78 size	LZ77 encode	LZ78 encode	LZ77 decode	LZ78 decode	LZ77 encode mem usage (peak)	LZ78 encode mem usage (peak)	LZ77 decode mem usage (peak)	LZ78 decode mem usage (peak)
alice29.txt	152KB	54KB	68KB	0.645 s	0.264 s	0.551 s	1.180 s	1.3 MB	1.1 MB	4.5 MB	9.1 MB
sherlock.txt	387KB	127KB	158KB	1.574 s	0.389 s	0.147 s	4.615 s	48.0 MB	18.8 MB	6.9 MB	18.5 MB
asyoulike.txt	125KB	49KB	61KB	0.525 s	0.282 s	0.565 s	0.998 s	22.1 MB	8.4 MB	4.5 MB	8.3 MB
bootstrap-3.3.6.min.css	121KB	20KB	45KB	0.574 s	0.239 s	0.336 s	0.664 s	13.5 MB	6.4 MB	3.5 MB	6.4 MB
sherlock_large.txt	3623 KB	124KB	1149 KB	5.970 s	1.974 s	1.483 s	27.282s	273.8 MB	65.8 MB	31 MB	25.5 MB
randomly generated char	1048 KB	757KB	790KB	2.051 s	1.021 s	21.874s	34.803s	258.9 MB	45.6 MB	11.4 MB	35.7 MB

The experimental results corroborate the initial hypothesis that LZ78 encodes faster than LZ77. It also consumes less memory during the compression phase because it does not require the retention of all previously observed content in a search tree. However, LZ78 exhibits a lower decompression speed due to the necessity of rebuilding and maintaining the dictionary employed in encoding, whereas LZ77 does not need an additional data structure for reconstructing the original data—it operates directly on the data it has partially decompressed.

Regarding the compression ratio, LZ78 underperforms compared to LZ77 across all test files. This outcome is anticipated since LZ78 can only match recurrences partially, whereas LZ77 is capable of matching them in full. This phenomenon is particularly evident with the 'sherlock_large.txt' test file, which was created by repeating the content of 'sherlock.txt' ten times. LZ77 can compress 'sherlock_large.txt' to a size comparable to that of the compressed 'sherlock.txt' by treating the first occurrence of content as an anchor and substituting the next nine recurrences with references to this anchor point. Conversely, LZ78 extends the match of the repetition incrementally with each occurrence, resulting in a larger compressed file size.

Conclusion:

This report delved into the intricacies of the LZ78 lossless compression algorithm, demystifying its dictionary-based approach to identifying redundancies and replacing them with efficient indexes. Furthermore, it conducted a comprehensive performance analysis, comparing LZ78 with the established LZ77 algorithm across essential metrics like compression ratio, processing speed, and memory usage. Based on these performance metrics, LZ78 shines when prioritizing compression speed. Conversely, LZ77 wins when aiming for the highest compression ratios, but it comes at the cost of slower processing and increased memory demands.