Indian Institute of Technology

Indore

Data Compression

Project Report

Department of Computer Science and Engineering

Submitted by –

Dhruv Chadha (150001009)

Apoorva Joshi (150001002)

Under the Guidance of **Dr. Kapil Ahuja**

Index

Introduction

Data compression involves encoding the given information and representing them using fewer bits than the original representation. A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (of bits, for example) which contains the same information but whose length is as small as possible. It is useful because it reduces the resources required to store and transmit data. As a trade-off for cheaper and faster transmission of data, computational resources are consumed in the compression and the decompression process.

Data compression paradigm involves trade-off among many factors, like Compression speed, Degree of compression, Decompression speed, and the computational resources required for compressing and decompressing the data.

Compression can be of 2 kinds, lossy or lossless. Lossless compression reduces bits by identifying and eliminating redundancy in the given file. No information is lost in lossless compression. Lossy compression, on the other hand, reduces bits by removing unnecessary or less important information.

Objectives –

1. To study and analyse the LZW and LZSS data compression algorithms.
2. Compression and decompression of a text file using LZW and LZSS algorithm.
3. Attempt to improve the compression speed in the LZW algorithm, and discussing future aspects.

Motivation –

Data compression finds its applications everywhere. It is widely used in backup utilities, spreadsheet applications, and database management systems. It also eases data transfer and storage.

Algorithm Analysis

Algorithm name: Lempel – Ziv – Welch (LZW)

Pseudocode:

Compression:

```
Initialize the dictionary (with the first 256 entries).
[prefix_string] ← [empty]
While (bytes left in input) {
    B ← next byte in the input.
    Is the string [prefix_string]B in the dictionary?
            Yes:
                    [prefix_string] ← [prefix_string]B
            No:
                    Add the string [prefix_string]B to the dictionary.
                    Output the index of [prefix_string] to the compressed file.
                    [prefix_string] ← B
}
Output the index of [prefix_string] to the compressed file.
```

Decompression:

```
Initialize the dictionary ‘string[]’ (with the first 256 entries).
<index> ← first index value in the input.
Write string[index] to the result.
[old] ← string[index]
While (bytes left in input) {
    <index> ← next index value in the input.
    Does <index> exist in the dictionary?
            Yes:
                    Write string[index] to the result.
                    B ← string[index][0]
                    Add [old]B to the dictionary.
            No:
                    B ← old[0]
                    Add [old]B to the dictionary.
                    Write the string for [old]B to the decompressed file.
    [old] ← string[index]
}
```

Complexity Analysis:
A simple implementation on the above algorithm takes $O(n^2)$ time for compression and $O(n)$ time for decompression.

Justification –

1) In the compression algorithm, the outer while loop –

"While (bytes left in input) "

takes O(length of file) time to run. Inside this while loop, searching a string in the dictionary –

"Is the string [prefix_string]B in the dictionary?"

Takes at max O(length of file) time to run as in the worst case, the dictionary can fill like this –

We know that 256 characters are already stored in the dictionary.

For the first 256 x 256 characters, the dictionary may fill at every new character, as each sequence of 2 characters may not be present.

For the next 256 x 256 x 256 characters, the dictionary will fill after every 2 characters, as every possible sequence of 2 characters is present in the dictionary.

So, the size of dictionary (in the worst case), varies with 'n = length of file' as –

$$\boldsymbol{dict\_size(n)} = \begin{cases} \boldsymbol{n}, & \boldsymbol{n \le 256} \\ \boldsymbol{256 + \frac{n-256}{2}}, & \boldsymbol{256 < n \le 256 \times 256} \\ \boldsymbol{256 + \frac{256 \times 256}{2} + \frac{n - 256 \times 256}{3}}, & \boldsymbol{256 \times 256 < n \le 256 \times 256 \times 256} \\ \boldsymbol{and\ so\ on} \ldots & \end{cases}$$

This simply turns out to be O(n), as for any n, the formula simply computes to a form $\frac{n}{a} + x$ where $x \leq$ n. So, searching for a string in dictionary turns out to be of linear time complexity.

Hence, the net result is $O(n^2)$ running time.

2) In the decompression algorithm, there is only an outer while loop –

"While (bytes left in input)"

that takes O(length of compressed file) time to run. Inside the while loop, searching an index in the dictionary –

"Does <index> exist in the dictionary?"

is a simple O(1) operation as it just checks whether the array size is atleast '**<index>**' length long.

Algorithm name: Lempel – Ziv – Storer - Szymanski (LZSS)

Pseudocode:

Compression:

```
while (lookAheadBuffer not empty) {
        get a reference (position, length) to longest match;
        if (length > 0) {
                output (position, length, next symbol);
                shift the window length+1 positions along;
        }
        else {
                output (0, 0, first symbol in the lookahead buffer);
                shift the window 1 character along;
        }
}
```

Decompression:

```
while (file not read) {
        if(flag_coded) {
                output (final_data(offset,length));
        }
        else {
                output(character);
        }
}
```

Complexity Analysis:

A simple implementation on the above algorithm takes $O(n^2)$ time for compression and $O(n)$ time for decompression. The time taken for compression can be improved to $O(n\log n)$ by using data structures like hash tables, binary search trees etc.

Justification-

1) In the compression algorithm, the outer while loop –

"While (LookAheadBuffer not empty) "

takes O(length of file) time to run as initially the LookAheadBuffer constitutes the entire unread data. In the standard implementation, even the LookAheadBuffer size is fixed to 4096 or 12 bits but in a naive implementation the LookAheadBuffer is fixed to be the entire unread file. To get the length of the longest match takes a naive implementation of -
O(length of Sliding Window × LookAheadBufferSize).
Since the length of the sliding window is fixed to be at the most 12 bits, it takes O(LookAheadBufferSize). This matching complexity can further be reduced by choosing data structure like hash table (directly apply KMP string matching algorithm). Then this tuple is written to the output file depending on whether the length of the match is greater than 0. It is not wise to write this tuple for uncoded characters, rather, a flag is used to denote whether the given character is coded or not. This takes constant time.
So, the total compression algorithm runs in quadratic time.

2) In the decompression algorithm, there is only an outer while loop –

“While (file not read)”

that takes O(length of compressed file) time to run. Inside the while loop, searching for the flag_coded in the dictionary –

“if(flag_coded)”

is a simple O(1) operation as it just checks whether the most significant bit of the read byte is 1 or not.
Checking it is a simple operation

“i >>15 == 1”

and then extracting the length and the offset values

“i & (0<<15)”

The call to the function final_data(offset, length) could be finally implemented in constant time by using splicing.
So, the overall decompression algorithm can be implemented in linear time.

Factors affecting speed and compression ratio

LZW algorithm

The speed of compression as well as the size of compressed file depends a lot on input file and the dictionary built up, stored and the way it is accessed. As the number of keys in the dictionary increases, a good idea would be to use hash tables to speed up access and storage. At the same time, when the number of keys increase decompression takes more time to access the values.

LZSS algorithm

The speed of compression as well as the size of compressed file depends largely on the input file and the size of LookAheadBuffer, the size of the Sliding Window and the maximum number of bits that could be alloted to the length of match (usually kept 4 bits). The compression ratio depends on the size of Sliding Window and the maximum length of the match while the time taken to compress depends on the size of LookAheadBuffer.

Note: One of the issues faced in compression and decompression when the file size is very large is that of the memory constraints (RAM) of the machine so many a times the file is loaded in chunks in the main memory and hence compressing and decompressing takes more time.

Optimisations

The naïve implementation of the LZW algorithm takes $O(n^2)$ time to compress a given text file, as discussed above. However, it can be optimized and can be done in linear time, using trie data structure.

Every node of the trie will be a structure containing 3 fields –

char symbol
int dict_index
node *children[256]

Pseudocode –

```
Initialize the dictionary 'root'
        for (i = 0 to 255) {
                root[i] -> symbol = character with ascii value 'i'.
                root[i] -> dict_index = i;
                initialise all children pointers to null;
        }
B = first character in file;
node *currentnode = root[ascii(B)];
int nextindex = 256;
While (bytes left in input) {
    B ← next byte in the input.
    Is a node with symbol = B present as a child of currentnode?
            Yes:
                    currentnode = child node of currentnode with symbol = B;
            No:
                    Create a new node 'temp' with symbol = B, dict_index =
                    nextindex, and all children pointers NULL;
                    nextindex++;
                    currentnode->children[ascii(B)] = temp;
                    Output the dict_index of currentnode to the compressed file;
                    currentnode = root[ascii(B)];
}
Output the dict_index of currentnode to the compressed file;
```

Complexity Analysis –

We see that now, with every new character read from the uncompressed file, we only check the next character, whether it is present in the trie, as a child of currentnode, which keeps updating itself with every passing letter. So, we do not need to check the whole string again and again.

The above implementation takes O(1) time to search and create the dictionary, along with an outer while loop that iterates through the length of the file.

Hence the optimised overall time complexity for the compression algorithm now becomes O(n).

Implementation and Results

The naïve LZW algorithm was implemented in python, whereas the efficient LZW algorithm was implemented in C++. Text files with varying sizes were compressed using the algorithm.

Data compression ratio = $\frac{Uncompressed\ file\ size}{Compressed\ file\ size}$

The results of the naïve algorithm were as follows-

Name of file	Original Size	Time taken to compress	Compressed file size	Compression Ratio	Time taken to decompress
small.txt	14.8 KB	1.07 seconds	8.42 KB	1.75	0.015 seconds
medium.txt	93.1 KB	16.25 seconds	33.5 KB	2.78	0.046 seconds
large.txt	162 KB	41.29 seconds	59.2 KB	2.73	0.062 seconds

The results of the efficient algorithm were as follows-

Name of file	Original Size	Time taken to compress	Compressed file size	Compression Ratio	Time taken to decompress
small.txt	14.8 KB	0.056 seconds	8.42 KB	1.75	0.015 seconds
medium.txt	93.1 KB	0.074 seconds	33.5 KB	2.78	0.046 seconds
large.txt	162 KB	0.107 seconds	59.2 KB	2.73	0.062 seconds

As a result, we can see a significant improvement in compression times, by using an efficient implementation of the LZW algorithm.

Future Work

LZSS and LZW can be further improved by using various special techniques.

DEFLATE is a modern data compression algorithm, used in the .zip file format. It improves LZSS by introducing Huffman codes. It is also based on the standard bit manipulation to denote the encoded and not coded streams.

Compression is achieved through two steps:

1) The matching and replacement of duplicate strings with pointers.
2) Replacing symbols with new, weighted symbols based on frequency of use.

Machine learning can also help in compressing data. A system that predicts the posterior probabilities of a sequence given its entire history can be used for optimal data compression (by using arithmetic coding on the output distribution) while an optimal compressor can be used for prediction (by finding the symbol that compresses best, given the previous history).

In arithmetic coding, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total.

Data compression is an open area of research.
After the deflate algorithm there have been many new discoveries - the most recent one is GOOGLE GUETZLI. The GOOGLE GUETZLI is a JPEG encoder that aims for excellent compression density at high visual quality.

References –

1. http://warp.povusers.org/EfficientLZW/index.html
2. http://marknelson.us/2011/11/08/lzw-revisited/
3. https://en.wikipedia.org/
4. http://michael.dipperstein.com/lzss/
5. https://sites.google.com/site/datacompressionguide/lz77
6. https://pdfs.semanticscholar.org/ddf5/4749ad7d83dbe6ec17e88e5f4af6ac69c1c5.pdf
7. https://github.com/google/guetzli