Document and re-implement LZMA algorithm

Group Member: Tiaohao Zhao, Shaoqin Lin

Abstract

This project aims to understand the compression algorithm LZMA, by designing a compression software. The software is a java program that uses the LZMA algorithm, which takes an input file and outputs a compressed zip file. In order to find advantages and disadvantages of the LZMA algorithm, this project studies the difference between LZMA and other compression algorithms.

LZMA uses context-based encoder and stores the longest context match, therefore it usually has a large dictionary size. For compression efficiency, LZMA and other combinational compression algorithms, such as Bzip2, have a higher compression ratio, but use longer compression time.

1. Introduction

Data compression has a great demand for people nowadays. On the road of exploring proper methods to reach the target, several data compression algorithms were developed in the past decades. Among those, the LZ77 was a pretty creative one which was published in 1977 by Abraham Lempel and Jacob Ziv. The Lemple-Ziv algorithm spread widely and derived many powerful algorithms based on LZ77 more or less. The Lempel-Ziv Markov Chain Algorithm (LZMA) was one of the products at that time, it became a widely used approach in document compression which was published in 1998. And this algorithm was the foundation of the 7-Zip archiver.

Our group will focus on the mechanism of LZMA which includes delta coding, sliding dictionary window and ranged coding. And a demo based on LZMA is also appended. Finally we will also present a comparison with other compression algorithms.

2. LZMA Algorithm

LZMA, or Lempel-Ziv-Markov Chain Algorithm, was established in 1998 which is based on LZ77. The aim of this algorithm is to optimize and provide higher compression rate and fast decompression. It is open source in varieties of different programming languages like C and Java. This algorithm can basically divide into three steps for encoding or decoding stage: delta encoding, sliding window and range coding. The Fig.1 shows the brief concept of

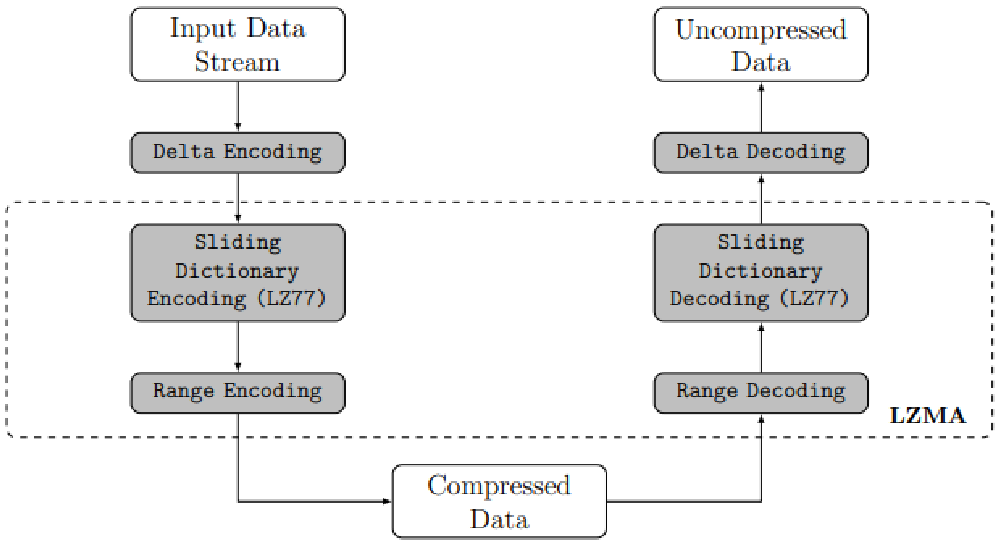
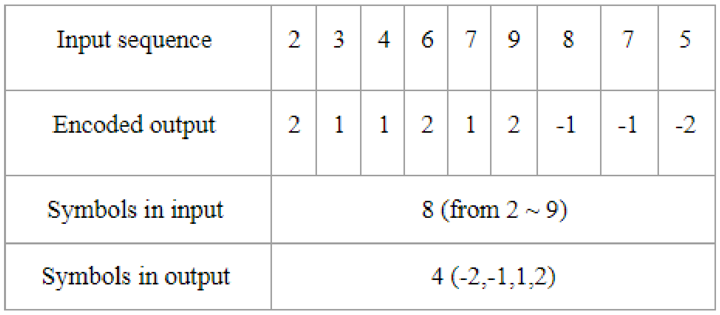


Figure 1: LZMA compression and decompression diagram (*Horita, Augusto Y., et al.*, 2019)

how LZMA works step by step.

2.1. Delta coding

Delta coding is the first step when the input data is sent into the LZMA process. A filter called "Delta Filter" is set up to shape the input data for a better compression. So that it can enhance the efficiency of the next step which is the sliding window. This step basically compresses the data via extracting the difference between sequential data instead of the whole complete files. Table 1 illustrates an example of delta coding.



Input sequence	2	3	4	6	7	9	8	7	5
Encoded output	2	1	1	2	1	2	-1	-1	-2
Symbols in input	8 (from 2 ~ 9)								
Symbols in output	4 (-2,-1,1,2)								

Table 1: Example of delta coding(adapted from E.Jebamalar, 2013)

The encoded output of the first byte is the data stream itself (for example, the first is 2 and the first output is also 2), then the following bytes are stored as the difference between the current data and its previous byte (for example, the second one is 3, the output of second is the difference between 3 itself and previous 2 that results to 1). According to this approach, the symbols of input can be compressed from 8 to 4. The delta coding can make the sliding window step more efficient.

2.2. Sliding dictionary window

This step is the key feature in the Lemple-Ziv part which is based on LZ77. Basically, the technology is matching in a generated large dictionary and compresses data by the distance from the current data in the look-ahead buffer to its match in the dictionary (or search buffer). Here's an example of a sliding dictionary window with detailed processes for better understanding.

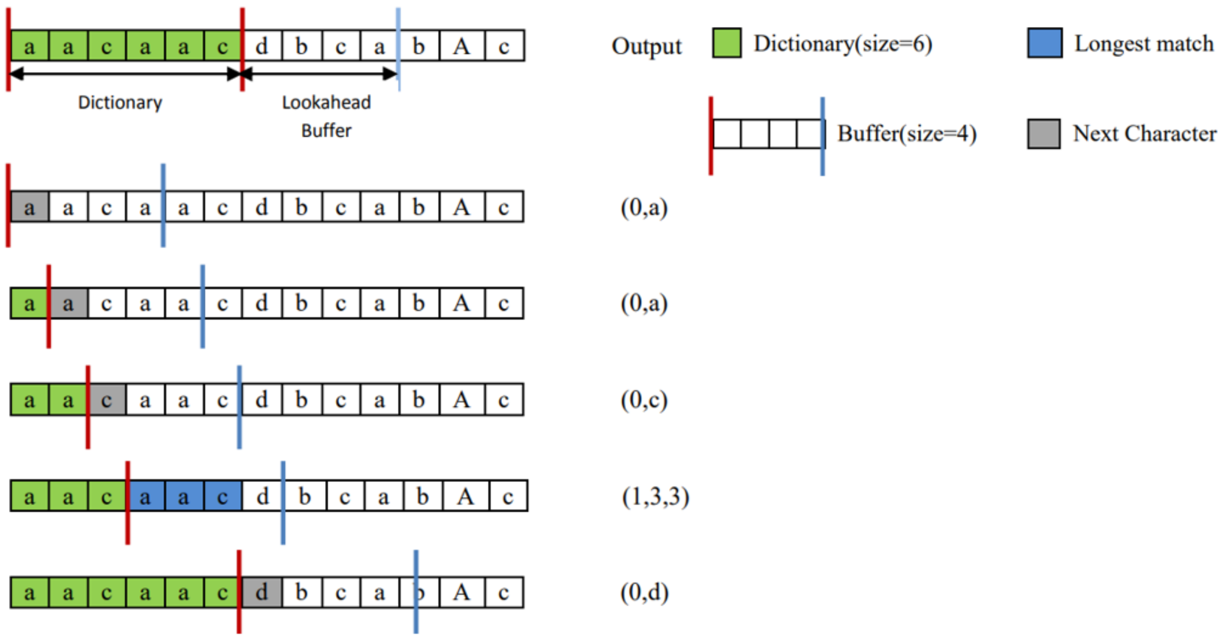


Figure 2: Example of Sliding dictionary window(Jing Zhang, 2015)

Figure 2 presents the diagram of how the sliding window works. And we make some assumptions in advance: the dictionary size is 6 bytes, look-ahead buffer size is 4 bytes, the minimum match length is 2 bytes and the maximum match length is 4 bytes. First the look-ahead buffer is filled with “acca” with the loading of the input stream. Because no match symbol is found in the dictionary, the output is a literal (0,a). The processed symbol “a” is moved into the dictionary with index 0. And the next input symbol “a” is moved into the buffer.

The process restarts by finding the match symbol in the dictionary. This time a match “a” is found in the dictionary, however, the match length is less than the minimum match length that we assumed before which is 2 bytes. Therefore, the output is also a literal(0,a). The next symbol “a” is moved in the dictionary with index 0 as the former step. The next input symbol “c” is moved into the buffer.

Process repeat again. Here no matching found in the dictionary, the output is a literal (0,c). The following symbol “c” is moved in the dictionary with index 0. Same as before, the new input symbol “d” is moved into the look-ahead buffer.

Repeat the step. A new matching string found in the dictionary this time. The longest match length is 3 bytes which is longer than we assumed before. The distance is 3 from the current symbol to the beginning matched symbol. Therefore, output is a literal (1,3,3). The sliding window process will continue until the end of the input data.

To enhance the whole efficiency of LZMA, the search algorithm should be optimized. And the proper approach is hashed chain and binary tree which can improve the searching speed. Here's an example of a binary search tree in LZMA of the searching process.

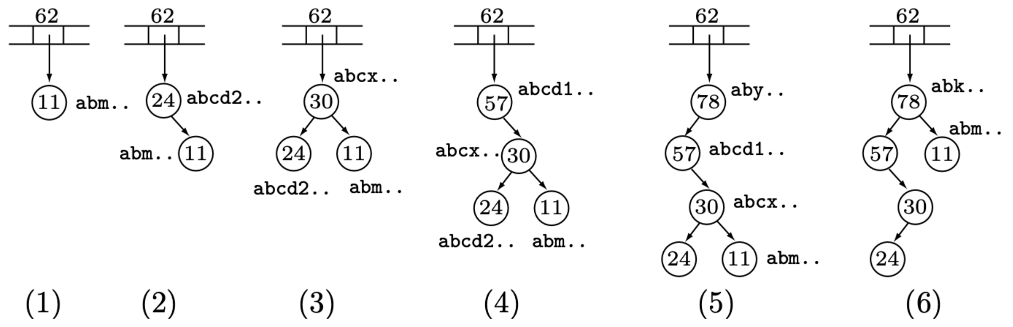


Figure 3: Binary search tree in LZMA(David, 2010)

Figure 3 presents the details of how binary search trees work in the searching process in LZMA. Similar to the former illustration of diagrams, we make some assumptions in advance. First of all, we list the details of the node sample in format of "nodes-string": 11-abm; 24-abcd2; 30-abcx; 57-abcd1; 78-aby. Additionally, we also assume all of the pair "ab" of bytes is hashed to 62 location. Then the encoder starts at location 11 and finds "a", it hashes this byte and the "b" that follows. Encoder examines location 62 of the hash-array and finds it empty. Then it generates a new binary tree for the pair "ab" with one node which contains the pointer 11. Also, it will set location 62 of the hash-array to point to this tree.

The encoder continues to the next pair "ab" which is found at location 24. Hashing produces the same 62 and location 62 of the hash-array is found to point to a binary tree whose root contains 11. Then the encoder sets 24 as the new root and 11 as its right subtree. In the match process, with a match length of 2 bytes, the encoder finds "abcd2" and "abm". After that, the encoder continues with "cd2".

Next pair “ab” is found at location 30. Same as the former, hashing produces the same 62, the encoder places 30 as the new root with 24 as its left subtree and 11 as its right subtree. The better match is “abcd2” with a match length of 3 bytes. The next two pairs “bc” and “cx” are appended to their respective trees.

The encoder continues with the pair “x”. And the next occurrence of “ab” is located at 57 which is also hashed to 62. It becomes the root of the tree with 30 as its right subtree. The best match is “abcd2” with a match length of 4 bytes.

Then the encoder continues with the string “l”, and it finds the next pair “ab” at location 78. This string “aby” becomes the new root with 57 as its left subtree. The match length is 2 bytes.
Now the encoder continues the step and reaches “abk” at location 78. Since “abm” is greater than “abk”, it should be moved to the right subtree of “abk”. Finally, a completely different tree is finished and this stage terminates.

2.3. Range coding

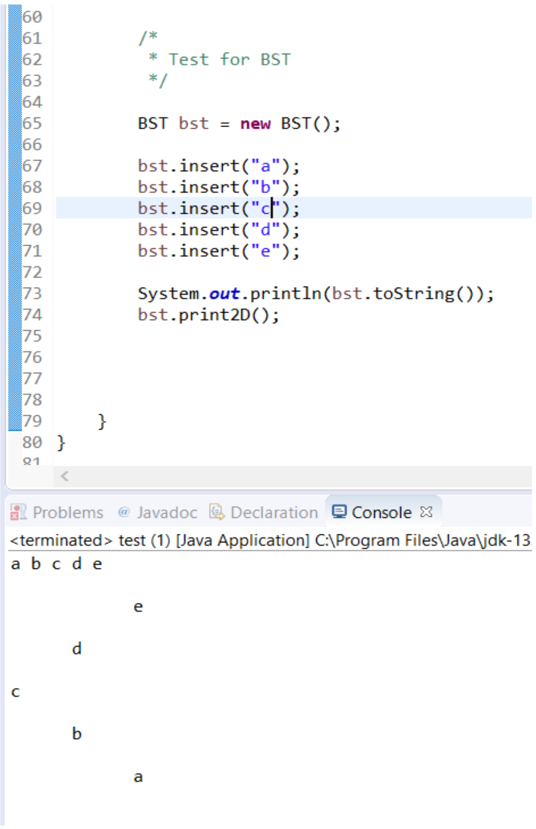
The range coder in LZMA can manipulate each bit in the process of compression or decompression (This is also the reason why LZMA has an excellent compression ratio compared to other algorithms). It converts texts or symbols of the message into binary numbers. The range encoder can be readily constructed based on the decoder description. It’s initialization and termination are not fully determined. There are several different encoder methods. The main goal of the encoder is trying to find the longest match of context and most repeated match in the dictionary.

The LZMA compression algorithm will usually create a large dictionary size. Therefore, the encoder needs to quickly locate matches of context in the dictionary. In order to perform fast searching on its library, there are few sophisticated data structures used, such as hash chains, and binary trees. They also combined the probability concept from the stochastic model Markov-chain. For example, a xz encoder includes 5 variables, *low*, *range*, *cache*, *cache_size*, *nice_len* and a *prob* variable for predicting occurrence of a previous value.

3. Test

First, we use the existing library to build a simple demo, which is based on the XZ library for java. This demo aims to test LZMACompressor.java to compress files.

Then, we create a BST class with Node and Hashmap. It implements a binary search tree with key-value pairs. At the same time, we store the key-values in a hashmap. The BST class includes insert(), getKey(), getVal(), contains(), size(), print2D() and toString() function. On the right, here is a brief demo of the BST class.

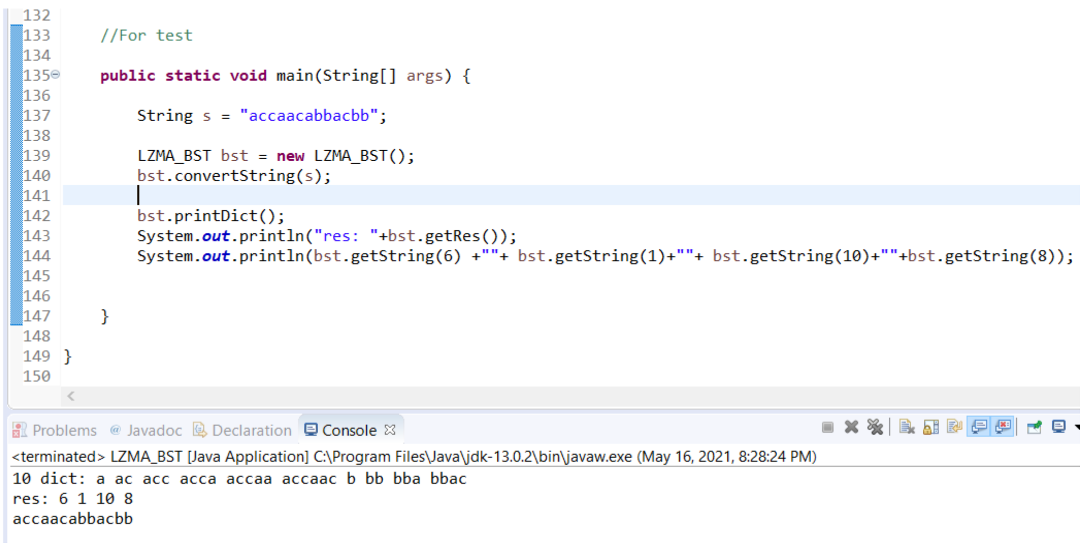


```
        /*
         * Test for BST
         */

        BST bst = new BST();

        bst.insert("a");
        bst.insert("b");
        bst.insert("c");
        bst.insert("d");
        bst.insert("e");

        System.out.println(bst.toString());
        bst.print2D();



    }
}
```

```
<terminated> test (1) [Java Application] C:\Program Files\Java\jdk-13
a b c d e

            e

      d

c

      b

            a
```

Figure 4: BST demo

In order to perform a conversion between text and numbers. We create a LZMA_BST class for taking input string and converting it to numbers. This class builds up a dictionary for conversion when parsing the input string. The dictionary is in BST type. This class includes convertString(), getRes(), printDict(), getString(), getLongestMatch().



```
    //For test

    public static void main(String[] args) {

        String s = "accaacabbacbb";

        LZMA_BST bst = new LZMA_BST();
        bst.convertString(s);

        bst.printDict();
        System.out.println("res: "+bst.getRes());
        System.out.println(bst.getString(6) +""+ bst.getString(1)+""+ bst.getString(10)+""+bst.getString(8));


    }

}
```

```
<terminated> LZMA_BST [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (May 16, 2021, 8:28:24 PM)
10 dict: a ac acc acca accaa accaac b bb bba bbac
res: 6 1 10 8
accaacabbacbb
```

Figure 5: LZMA_BST demo

4. Comparison and Conclusion

Prior to LZMA, most encoder models were purely byte-based, which uses a cascade of context to represent dependencies on previous value. LZMA uses a context-based model, which uses contexts corresponding to its bitfields to represent literal or phrase. This model avoids mixing unrelated bits together, but it lets the LZMA encoder create a much larger dictionary than other compression algorithms. LZMA incorporates the Markov chain model, therefore it is considered as a combinational compression algorithm that uses arithmetic coding instead of traditional Huffman coding. In general, combinational compression algorithms perform better than other compression algorithms. They have a higher compression ratio, but they usually take more compression time.

Reference

Alakuijala, Jyrki, and Evgenii Kliuchnikov. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and ... Google, Inc., cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf.

Bin Tariq, Zaid, et al. "Enhanced LZMA and BZIP2 for Improved Energy Data Compression." Proceedings of the 4th International Conference on Smart Cities and Green ICT Systems, 2015, doi:10.5220/0005454202560263.

Handbook of Data Compression, by David Salomon and Giovanni Motta, Springer London, 2010, pp. 414–423.

Horita, Augusto Y., et al. "Lempel-Ziv-Markov Chain Algorithm Modeling Using Models of Computation and ForSyDe." Proceedings of the 10th Aerospace Technology Congress, October 8-9, 2019, Stockholm, Sweden, 2019, doi:10.3384/ecp19162017.

Lan, Cuiling, et al. "Compound Image Compression Using Lossless and Lossy LZMA in HEVC." 2015 IEEE International Conference on Multimedia and Expo (ICME), 2015, doi:10.1109/icme.2015.7177430.

Leavline, E.Jebamalar, and D.Asir Antony Gnana Singh. Hardware Implementation of LZMA Data Compression Algorithm. 4 Nov. 2013, research.ijais.org/volume5/number4/ijais12-450900.pdf.

"Lempel–Ziv–Markov Chain Algorithm." Wikipedia, Wikimedia Foundation, 27 Apr. 2021, en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm#Compression_algorithm_details.

Zhang, Jing. Real-Time Lossless Compression of SoC Trace Data. Dec. 2015, www.eit.lth.se/sprapport.php?uid=900.