Java Lempel–Ziv–Welch Compression, AES Encryption with GUI Interface

“LZW Guard”

Alex Breger 205580087 & Danny Kogel 318503257

Data Compression Course – Sapir College 2021

Lecturer: Dr. Amit Benbassat

Implementation:

Written in Java, a program that compresses and decompresses files using LZW compression, allowing the user to choose multiple files by using “Drag and Drop”, setting the bit size of the LZW algorithm, as well as encrypting and decrypting the files using AES implemented by the Java Cipher class. The GUI is implemented with Java Swing.

Data Compression Course Final Project by Alex Breger 205580087 & Danny Kogel 318503257

Java Lempel–Ziv–Welch Compression, AES Encryption with GUI Interface

"LZW Guard"

Table of Content

References

Owen Astrachan, 2004 BitInputStream.java / BitOutputStream.java

https://courses.cs.duke.edu/spring11/cps100e/assign/huff/code/BitInputStream.html

https://courses.cs.duke.edu/spring11/cps100e/assign/huff/code/BitOutputStream.html

Nam Ha Minh, 2019 Java File Encryption and Decryption Simple Example

https://www.codejava.net/coding/file-encryption-and-decryption-simple-example

“Raja” ,2019 How to restrict the number of digits inside JPasswordField in Java

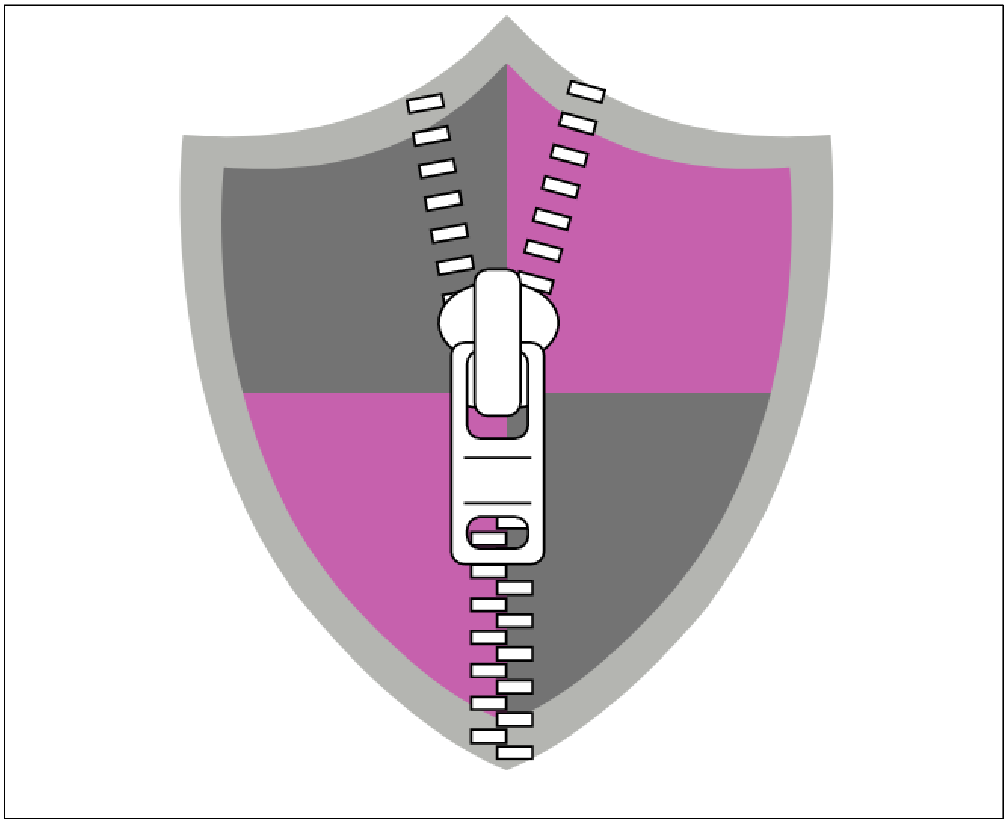
https://www.tutorialspoint.com/how-to-restrict-the-number-of-digits-inside-jpasswordfield-in-java

“Art of the Probelm” ,2018 The Beauty of Lempel-Ziv Compression

https://www.youtube.com/watch?v=RV5aUr8sZD0

Data Compression Course Final Project by Alex Breger 205580087 & Danny Kogel 318503257

"LZW Guard"



Lempel–Ziv–Welch Compression

Presented in 1984 by Abraham Lempel, Jacob Ziv and Terry Welch is a lossless data compression algorithm. The algorithm states that we create a set of rules by which we create a compression dictionary. We create a base dictionary and, while we compress the file, we add more 'keys: values' based on previous dictionary items. After compression, we don't send the whole dictionary, which saves us space and makes the file more compressed.

The only thing we must send is the base dictionary and a defined bit size to be read or written.

Compression Pseudocode:

```
Loop while file not empty
        Initialize the dictionary to contain all strings of length one.
        Find the longest string W in the dictionary that matches the current input.
        Emit the dictionary index for W to output and remove W from the input.
        Add W followed by the next symbol in the input to the dictionary.
```

By chaining characters, we create new words that are easy to represent with the 'key: value' system, where the key is the string, and the value is a number. We output the numbers into the compressed file. Why should we write the same long sentence if we can just reference it with a shorter number?

During decompressing the same logic applies, we will read the output numerical representation, each time checking if we have the representation in the dictionary. If yes, we write the corresponding string to the decompressed file, if not, we will create the string using the known characters, this way we could create the original dictionary once again and rebuild the file using only the numerical references of strings.

Java Implementation

Using the above logic, we started programing something that can be used both on text files and various other files, we've set the original ascii 256 characters as the default dictionary. Because of our first 256 dictionary entries have already taken half the size in case of 9-bit reading/writing, a small bit size as 9 would not make sense and limit us to very small files, for example a 9-bit program will be able to read at most 2^9-256bits considering the worst case, although it's unlikely a small file will capture all available dictionary entries, as the more entries are added the less likely we will encounter a certain bit string for the first time. So, we've decided to set the minimum size to 12, maximum size to 32 and the default size to 22. The bigger the bit size the larger files we can read but it also means compression might be worse.

<u>Compress()</u> - First we will read a certain amount of bits, between 12-32, chosen by the user, generate a default dictionary into a HashMap, read next char. if we encountered the first char with next, combine them to be the next "next", if not, write their index in the output file and remember it in the HashMap . Do so until file is empty.

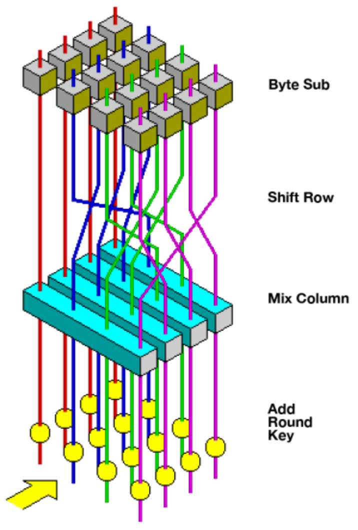
<u>Decompress()</u> - Decompressing is similar, but instead of using the strings as keys for the HashMap we use the indexes. Each time reading a certain bit size, and look for that value in the HashMap, finding the appropriate char for that key. If the dictionary contains the key, it will print it and register the key, if it does not exist yet it will register the added string without the added character. And does so till the end of the file.

<u>BitOutputStream / BitInputStream</u> - are classes that allow us to write and read bits and not bytes, it allows us more control over the size of the bit strings we are reading and allows us to use non byte sizes like 12 bits, 20 bits etc. Changing these values will allow us to compress certain files better as the bigger the bit size the larger the strings we can read but also a bigger dictionary which means a worse compression.

Java Cipher Class

Cipher class is an inbuilt java resource that allows implementing a certain cryptographic functionality. Here we used it to implement the AES 128bit encryption algorithm.

AES – This encryption standard takes a symmetric key, meaning the same key is used both for encryption and decryption and uses it to encrypt blocks of data. The same key is usually copied and expanded into "instructions" like sequences for each step of the encryption and decryption. The AES reshuffles the way the data is stored based on the given key and only with it it's possible to shuffle the data back to a readable state.

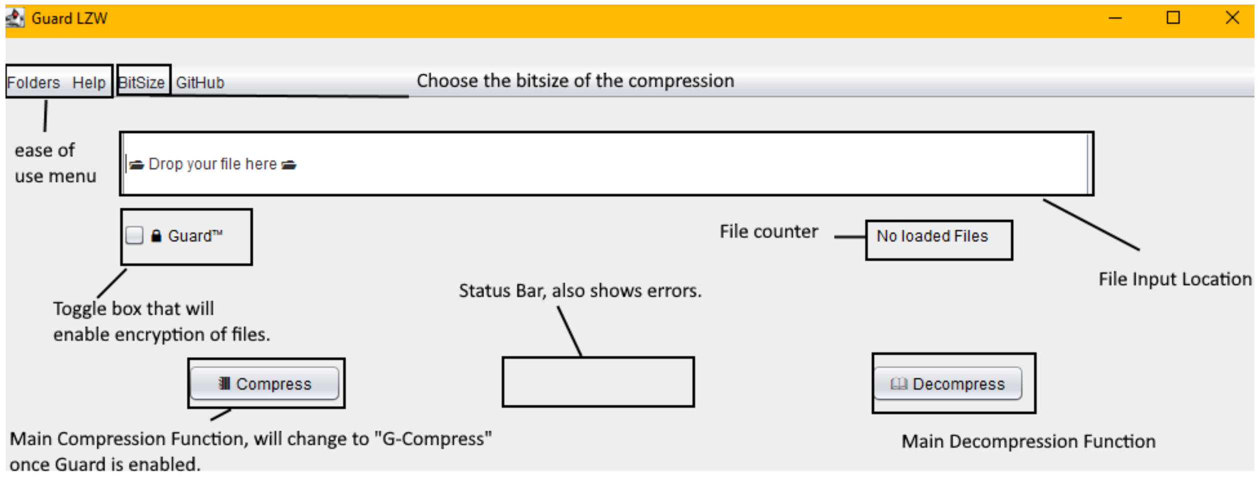


The implementation we used for it in Java was taken from the internet. We've found it to fit our needs and be simple enough to not have anything improved or changed for encrypting already compressed or decompressed files.

Although at the process of LZW we create both an encrypted and a non-encrypted version of the file which could cause a security problem as there is an option to use the non-encrypted file, but we've decided to focus on the idea of encryption rather than a proper high security implementation for the project purposes and leave the option to check the non-encrypted files as well without the need to compress and decompress again.

It was a good opportunity to learn about this algorithm and find a way to implement it in our work.

Java Swing GUI


Guard LZW
Folders
Help
BitSize
GitHub
Choose the bitsize of the compression
ease of use menu
Drop your file here
Guard™
File counter
No loaded Files
File Input Location
Toggle box that will enable encryption of files.
Status Bar, also shows errors.
Compress
Decompress
Main Compression Function, will change to "G-Compress" once Guard is enabled.
Main Decompression Function

The above GUI is a simple implementation for our program. It allows for fairy comfortable file choosing (Drag and Drop), even dropping multiple files, although the program isn't multi-threaded so compression times will still be as running the files one by one. The Guard toggle button will show a password field limited to 16 characters as requires by AES 128, any shorter passwords will be padded to the required length and empty field will use a default password "1234567891011121", Please notice the function to limit the character length was taken from the internet as we wanted to limit it in an aesthetic way, It's not necessary as we could've just taken the 16 first character of the password. Status bar will warn the user if he tries to compress already compressed files or decompressed non compressed files and other common user errors.

Compressed files will end with a ".lzw" at the end of the file and will be stored in a special folder named "CompressedFiles" that will be created upon first use. Encrypted compressed files will have ".lzw.Guard" at the end and will be stored in the same folder.

Decompressed files will be stored in a special folder named "DecompressedFiles" that will be created upon first use.

Final Thoughts

Although this may not be the most perfect implantation of our program, we encountered a lot of things we never had tried before and is sure was pleasant to work by our own specifications and our choice of implementation. There might be some errors present in the code that we didn't manage to find during debugging.

We found the compression to be surprisingly efficient with certain files even achieving compression rates close to the default Windows zip program.

We found out that although compression seems to lose some of its usefulness with the prices of hardware storages being lower every year, there are still plenty of fields where compression could be more essential like networking and cloud.