Universidade Federal da Paraíba
Centro de Informática

Introdução à Teoria da Informação

Gabriel Teixeira Patrício - Matrícula: 20170170889

Sarah Andrade Toscano de Carvalho - Matrícula: 20170022975

João Pessoa, 2021.

Universidade Federal da Paraíba
Centro de Informática

Projeto Final: Compressor e Descompressor de dados - LZW (parte I)

Relatório parcial sobre o software de compressão e descompressão de dados, cujo funcionamento baseia-se no algoritmo LZW. Tal projeto está sendo desenvolvido durante a realização da disciplina de Introdução à informação, ministrada pelo professor Derzu Omaia, do Centro de Informática da Universidade Federal da Paraíba.

João Pessoa, 2021.

Sumário

Objetivo Geral

Neste trabalho almejou-se a implementação de um software que realizasse a função de um compressor e descompressor de dados. Os algoritmos de compressão e descompressão que poderiam ser utilizados nesse programa foram o PPM e o LZW. Dessa forma, optamos por utilizar o LZW pela facilidade de implementação quando comparado a lógica do PPM.

Objetivo Específico

Implementação de um software baseado no algoritmo LZW para compressão e descompressão de dados. Não foram utilizadas bibliotecas como referência para simular o funcionamento de tal algoritmo. O código foi desenvolvido com a linguagem de programação Python. O software foi testado com um arquivo .txt de 16MB e outro em .mp4, além disso os dicionários foram testados com tamanhos variados, com K iniciando em 9 e indo até 16.

Introdução

O Algoritmo **LZW (Lempel–Ziv–Welch)**, resulta do melhoramento efetuado por Terry Welch dos algoritmos LZ77 e LZ78, os quais foram desenvolvidos por Jacob ZIV e Abraham Lempel, respetivamente. Inicialmente, foi desenhado a pensar no número de bits que se poderia reduzir aquando do envio de ficheiros armazenados nos discos, mas tem vindo a ser usado em muitos outros contextos. [1]

Trata-se de um algoritmo de compressão de dados baseado num dicionário que produz códigos de comprimento fixo associados a sequências de símbolos de comprimento variável.

Os códigos gerados durante o processo, tanto de codificação como de descodificação, correspondem a sequências de símbolos cada vez mais longas, que vão sendo adicionadas ao dicionário. O LZW é um modo de compressão sem perdas (CSP), o que significa que o fluxo de dados descomprimidos é exatamente igual ao fluxo de dados originais, não se perdendo qualquer informação. [2]

Em resumo, o algoritmo LZW é um algoritmo de codificação de palavras cuja idéia principal é ir construindo um dicionário de símbolos ou palavras conforme o texto ou a informação vai sendo processado pelo algoritmo.

Desse modo utilizamos a linguagem de programação Python para construirmos o algoritmo do LZW e executá-lo com diferentes arquivos e com dicionários de tamanhos variados, com K iniciando em 9 e indo até 16.

Desenvolvimento

Para realizar a compressão iniciamos o dicionário LZW com um tamanho de 256 caracteres da tabela ASCII. Durante o processo de análise do texto, vamos adicionando no dicionário novas palavras a partir de combinações de pelo menos 2 caracteres encontrados no texto. A cada nova combinação encontrada, é criada uma nova entrada no dicionário que será

utilizada posteriormente para novas codificações. Ao final do processo de codificação, teremos um dicionário de combinações de palavras que poderá ser utilizado para codificar o texto.

Algoritmo do compressor

A lógica do algoritmo de compressão pode ser descrita em duas etapas principais, uma quando o caracter analisado encontra-se indexado no dicionário e outra quando ele não está. Considerando a construção inicial do dicionário como etapa primordial, o funcionamento dessas situações é detalhado com o diagrama em blocos a seguir:

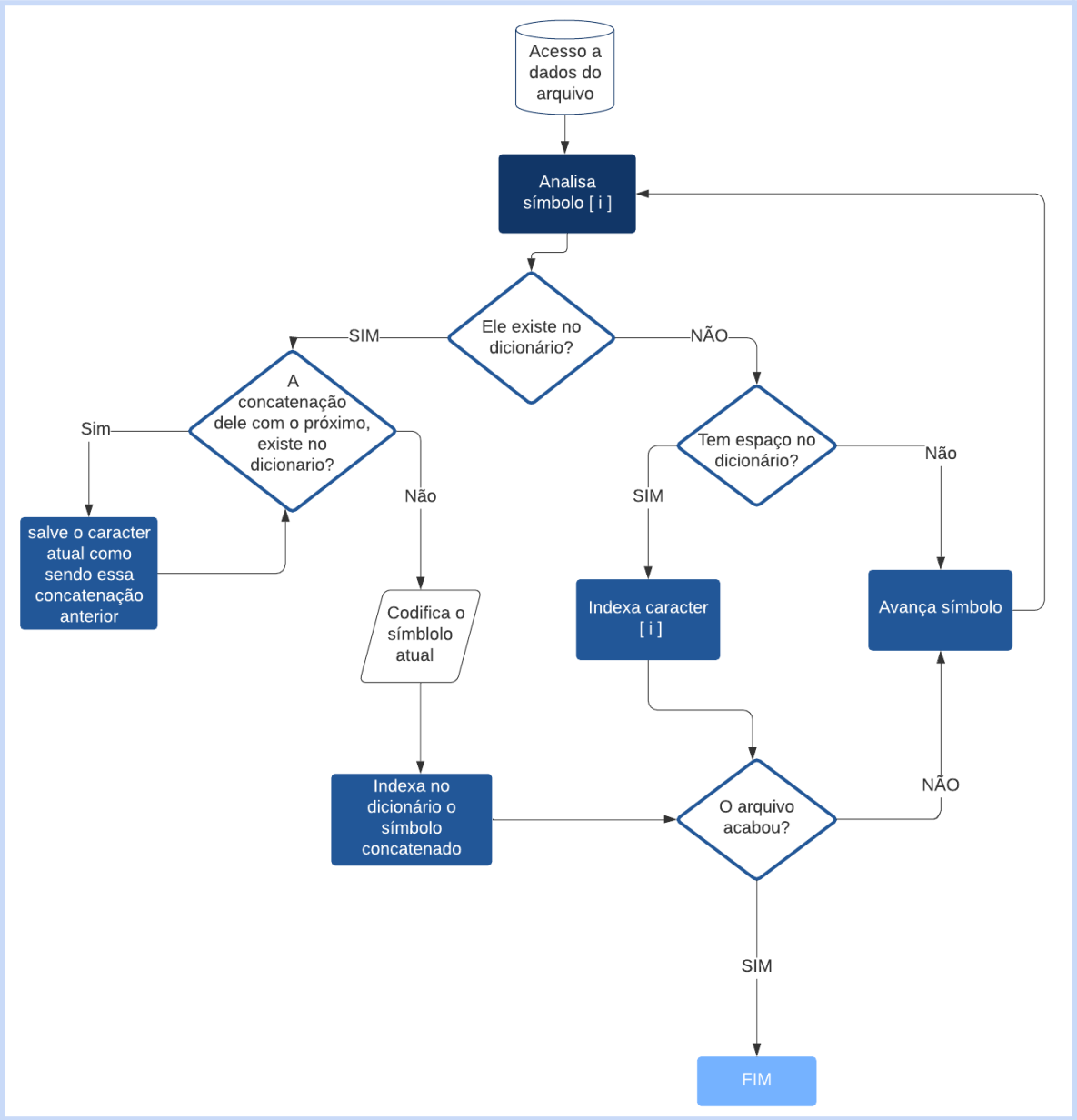


Figura 1. Diagrama em blocos do funcionamento do compressor pelo algoritmo do LZW.

Algoritmo do descompressor

O funcionamento do descompressor é similar ao do compressor, com algumas peculiaridades. Na leitura inicial do arquivo já é feita a decodificação do número binário para o seu inteiro equivalente e em seguida para o seu char equivalente na tabela ascii. Desse modo, vale ressaltar que no compressor, a modificação dos dados é feita apenas no final para escrevê-los no arquivo binário, enquanto no descompressor a primeira etapa já envolve tratamento nos dados recebidos.

O diagrama de blocos, ilustrado abaixo, destaca em verde, as principais diferenças de funcionamento em relação ao compressor.

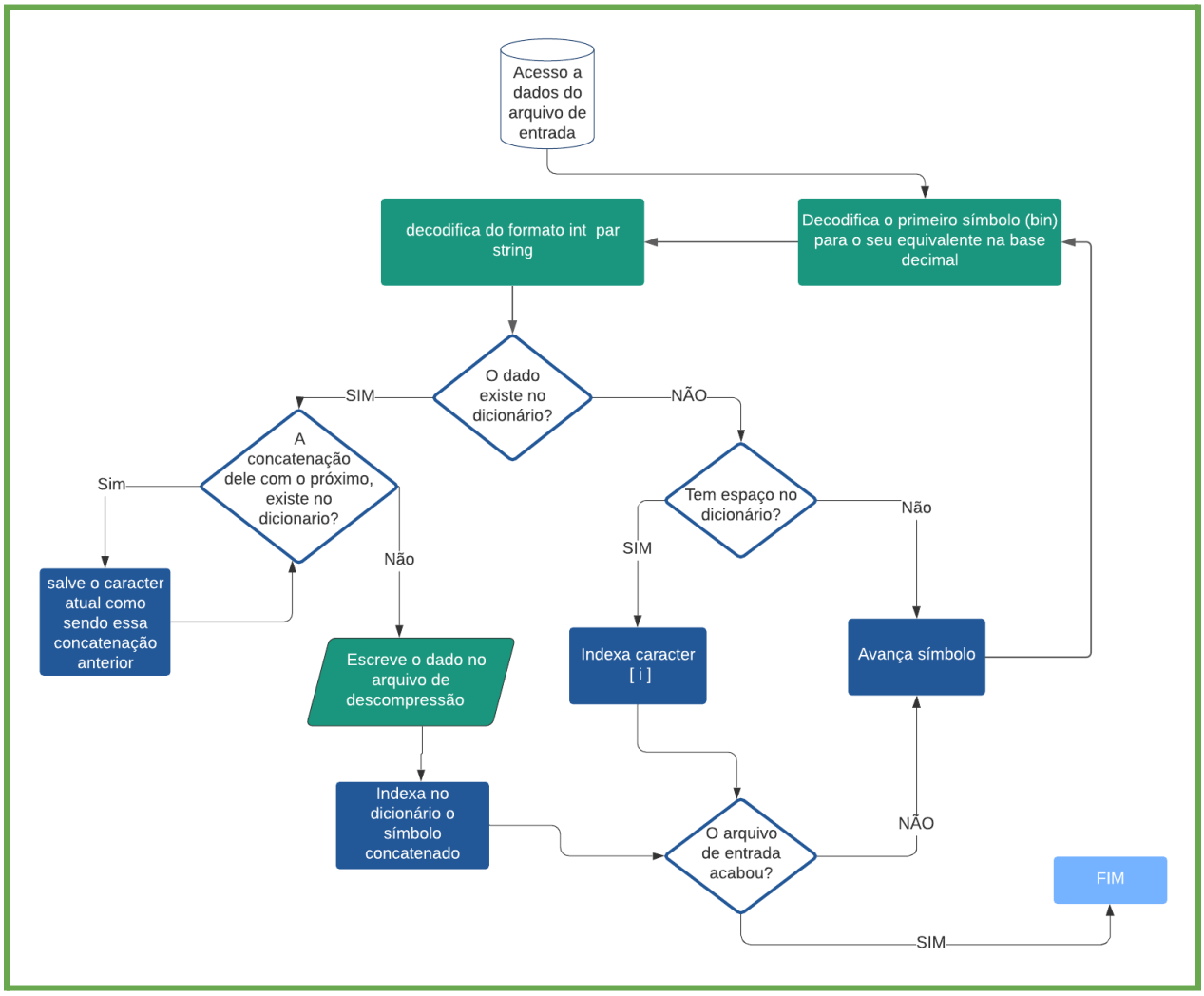


Figura 3. Diagrama em blocos do funcionamento do descompressor pelo algoritmo do LZW.

Problemas e soluções

Durante a implementação do compressor alguns problemas foram encontrados, principalmente em relação a codificação e no modo de criação do dicionário.

Indexação dos símbolos no dicionário

Inicialmente, estávamos inicializado o dicionário (posições 0-255) com através da função *chr( )*. A qual convertia o valor em inteiro, para o seu símbolo equivalente na tabela ASCII. O problema dava-se nas demais posições, quando tentávamos adicionar ao dicionário um símbolo especial. Esta limitação nos impedia, por exemplo, de realizar a codificação de um texto com quebra de linhas, pois o *\n\r* não era interpretado pelo chr.

A solução para este problema foi utilizar as funções encode( )/decode( ) do python, definindo como linguagem do texto o 'latin-1'. Dessa forma, quando tínhamos a string e gostaríamos de ter acesso ao seu index, bastava realizar string.encode('latin-1'). De modo análogo, para saber qual o símbolo representava determinado index, o string.decode('latin-1) foi a solução. Desse modo, conseguimos gerar uma lista com todos os index, em inteiro, que deveriam ser convertidos para binário, codificação, e em seguida escritos no arquivo final do compressor.

Tamanho da codificação dos dados no arquivo

O segundo problema surge neste cenário, como codificação do index deveria ser feita com a quantidade de bits igual ao K da execução, o primeiro método de tentativa da codificação foi falho. Inicialmente, tentamos a seguinte solução *'{0:08b}'.format(var_p_codificar)*. De fato, no arquivo, os dados eram codificados com K bits, porém cada bit codificado neste formato era alocado em 1 byte. Ou seja, para codificar somente o caracter 'a', com K igual a 9, eram gastos 9 bytes.

Para solucionar este problema, utilizamos o struct.pack( ) que interpreta os bytes como dados binários compactados. Desse modo, durante a codificação definimos o tamanho H, equivalente a um *unsigned short*, ou seja, 2 bytes. Desse modo, para codificar apenas o caracter 'a' com K sendo 9, o arquivo binário teria um tamanho de 2 bytes, dos quais apenas 9 bits seriam uma informação útil.

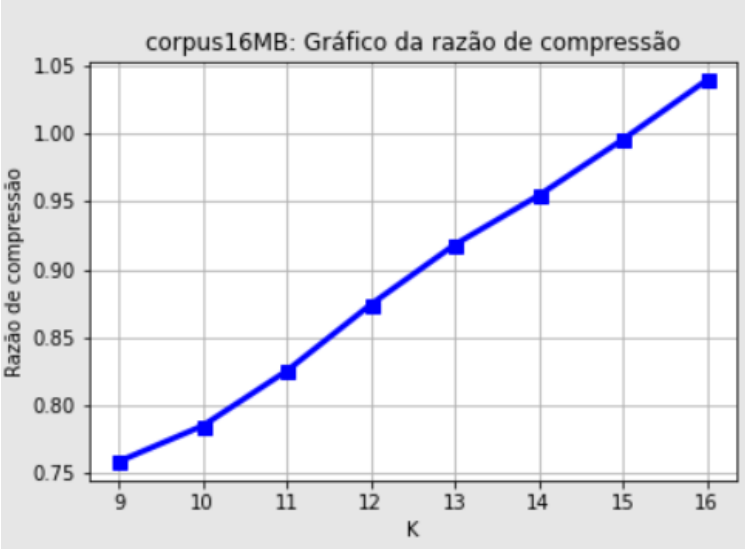
Análise e Resultados

Após a implementação das soluções citadas anteriormente, o compressor foi testado com K variando de 9 até 16, para o arquivo do corpus e do vídeo, gerando assim o seu equivalente na versão comprimida em binário. Em seguida, após tal arquivo ser interpretado pelo descompressor, todos os arquivos foram restaurados perfeitamente, sem haver perda de informações no texto e na reprodução do vídeo.

Corpus 16 MB e Vídeo

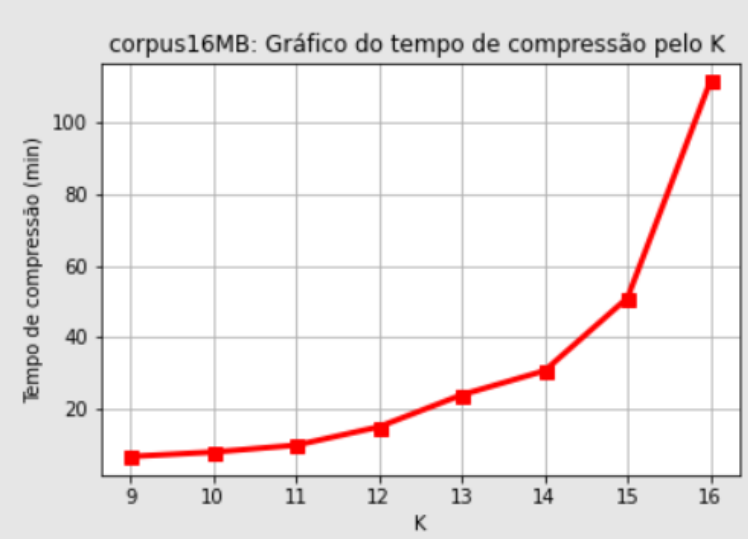
- Razão de Compressão por K

No gráfico a seguir podemos perceber que com o aumento do K a razão de compressão (que representa a quantidade de bytes do original referente ao comprimido) diminui para o caso do vídeo. Mas para o caso do texto essa razão é o contrário, quanto maior o K, maior a razão de compressão. A equação utilizada para o cálculo da razão de compressão para cada variação do K foi, RC = tamArqOriginal / ((totalIndices*K)/8).



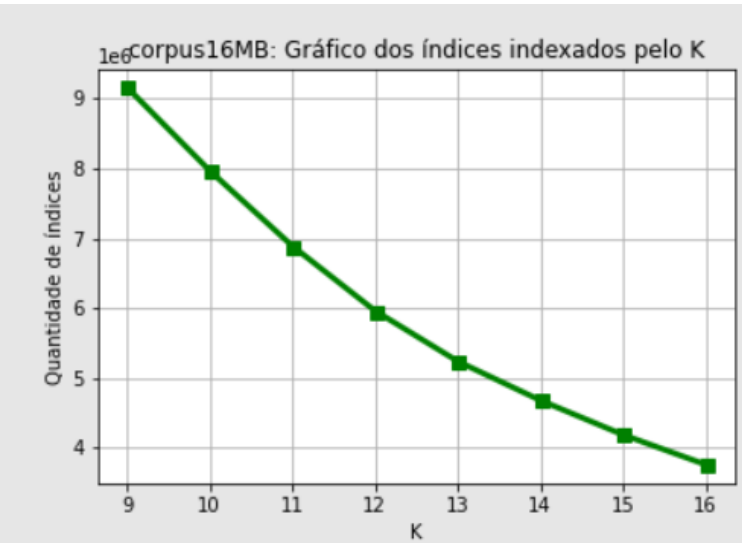
- Tempo de Processamento por K

Analisando os gráficos abaixo pode-se inferir que o compressor possui um tempo variavelmente linear para realizar a compressão dos dados. Nesse caso, as curvas são semelhantes, a diferença se evidencia na proporção do tempo. Como o vídeo é um arquivo menor, ele é comprimido mais rápido do que o arquivo do corpus.



- Índices codificados por K

Nos gráficos a seguir percebe-se que quanto maior o K, menor será a quantidade de índices indexados, pois o número disponível no dicionário é maior e não se tem a necessidade de uma quantidade maior de índices. Quando se tem um número pequeno de K, como por exemplo no caso da esquerda entre K9 - K13, o número do dicionário não foi suficiente para fazer a codificação e necessitando de mais índices.



Conclusão

Em suma, o compressor implementado nesta disciplina apresenta um tempo de compressão variável linearmente em relação ao número de Ks. Além disso, a quantidade de índices codificados diminui quando o K aumenta.

Referências

1. http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lzw/
2. https://www.ime.usp.br/~fli/gzip.html
3. http://www.ic.uff.br/~aconci/LZW.pdf
4. https://www.decom.fee.unicamp.br/dspcom/EE088/Algoritmo_LZW.pdf
5. https://lucid.app/lucidchart/11576a09-73a4-4ada-b72d-8269be1d02c1/edit?viewport_loc=-205%2C746%2C2074%2C855%2C0_0&invitationId=inv_97c215ce-9790-46f8-9ba8-46b82b8dadf3
6. https://lucid.app/lucidchart/17c6c72d-ab2b-4143-b94a-a44540a4c726/edit?viewport_loc=-244%2C99%2C2414%2C996%2C0_0&invitationId=inv_22c30412-0d2e-4d30-93be-491d21e585c2