

Relatório do projeto da disciplina de Introdução à Teoria da Informação

Implementação do algoritmo de compressão e descompressão Lempel-Ziv-Welch (LZW) utilizando a linguagem de programação Python

Jackson Leandro do Nascimento
Universidade Federal da Paraíba
Depto. de Sistemas para Computação
Engenharia da Computação
João Pessoa, Paraíba

Email: jacksonleandro077@hotmail.com@gmail.com

Luiz Eduardo Rodrigues Correa
Universidade Federal da Paraíba
Depto. de Sistemas para Computação
Engenharia da Computação
João Pessoa, Paraíba

Email: lerc.luizrodrigues@gmail.com

Gabriel Assis Guedes Rosas
Universidade Federal da Paraíba
Depto. de Sistemas para Computação
Engenharia da Computação
João Pessoa, Paraíba

Email: gabrielrosas@eng.ci.ufpb.br

Abstract—This report will present and evaluate the results obtained in the development of the project, which is the implementation of the Lempel-Ziv-Welch (LZW) algorithm, used to compress and decompress data. This algorithm was presented in this discipline (Introduction to Information Theory). For its implementation, the Python language was used.

08 de maio, 2023

Resumo—Neste relatório serão apresentados e avaliados os resultados obtidos no desenvolvimento do projeto, que é a implementação do algoritmo de Lempel-Ziv-Welch (LZW), usado para fazer compressão e descompressão de dados. Esse algoritmo foi apresentado nesta disciplina (Introdução à Teoria da Informação). Para sua implementação foi usada a linguagem Python.

08 de maio, 2023

1. Introdução

Para a implementação do algoritmo optou-se por usar a linguagem Python, pois ela é bastante versátil, poderosa e os membros do grupo já possuíam certa familiaridade com ela. As funções para compressão e descompressão foram implementadas em um único arquivo de maneira simples e concisa, com o auxílio de bibliotecas da própria linguagem.

2. Funções de compressão e descompressão

Para o caso desse projeto, o professor forneceu dois arquivos para serem utilizados nos testes, um arquivo de texto (.txt) e um arquivo de vídeo (.mp4) e ambos deveriam ser tratados pelo algoritmo para compressão e posteriormente descompressão, para que possamos validar a implementação.

2.1. Compressão

Em uma breve explicação temos que, o algoritmo de compressão recebe dois parâmetros, o dados do arquivo que será comprimido e o tamanho máximo do dicionário que será usado durante o processo de compressão. A função inicializa com os primeiros 256 caracteres padrão da tabela ASCII como chaves e seus respectivos valores em bytes, em seguida itera pelos dados de entrada, construindo um *buffer* até encontrar uma sequência de bytes que não esteja no dicionário. Quando isso ocorre, a função gera o código correspondente ao *buffer*, adiciona a nova

sequência ao dicionário e define o *buffer* para o símbolo de byte atual. A função também usa o método "struct.pack()" para empacotar os códigos de saída no formato *little-endian* de 2 bytes e os anexa a um *bytearray* chamado "resultado", que conterá os dados compactados. Por fim, ele gera os dados compactados e o tamanho do dicionário, que será útil para a descompressão.

2.2. Descompressão

Já para a função de descompressão temos o seguinte funcionamento, inicia-se criando um dicionário com as chaves em bytes e os valores inteiros que vão de 0 a 255 (totalizando os 256 valores da tabela ASCII). Em seguida ele inicializa variáveis para armazenar valores de modo temporário. No *loop* principal tem-se o processo de desempacotamento do próximo par de bytes do arquivo de entrada e converte para um número inteiro usando o método "struct.unpack()", é então verificado se o código descompactado está presente no dicionário, caso esteja o valor é adicionado a uma variável. Caso contrário cria-se um novo valor adicionando o primeiro caractere do *buffer* no final e o adiciona ao dicionário. Então esse valor é adicionado à variável "resultado" que armazena a descompactação no final do arquivo, logo em seguida verifica-se se o *buffer* já possui um valor, caso tenha ele adiciona a sequência atual do dicionário, desde que o mesmo não esteja cheio. Caso contrário apenas adiciona-se o valor atual ao *buffer*. E por fim o código retorna a variável "resultado" que contém a descompressão completa do arquivo e seu tamanho final do dicionário.

3. Valores de 'K' utilizados

Todos os valores de K (neste caso do 9 ao 16, pois foi implementado o LZW) funcionaram bem, tanto para o arquivo de texto quanto para o de vídeo.

4. Obtenção do valor exato de 'K' bits

Para essa finalidade conseguimos salvar o valor de bits ao salvá-los em índices de 2 bytes.

5. Razão de diferentes 'RCs' x K

Para esse projeto tivemos que utilizar duas expressões diferentes para a obtenção da razão de compressão (RC), e para

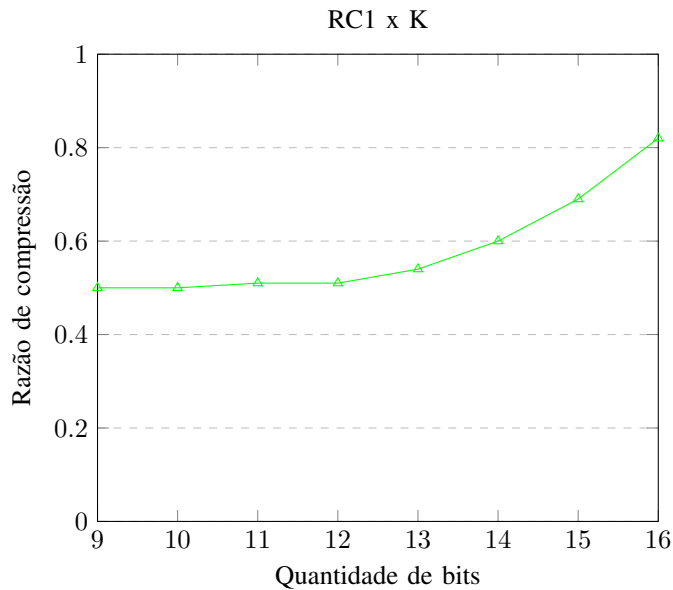
fins de auxiliar na elaboração do relatório elas serão nomeadas como RC1 e RC2, da seguinte forma:

$$RC1 = \frac{tamArgOriginal}{tamArgComprimido}$$

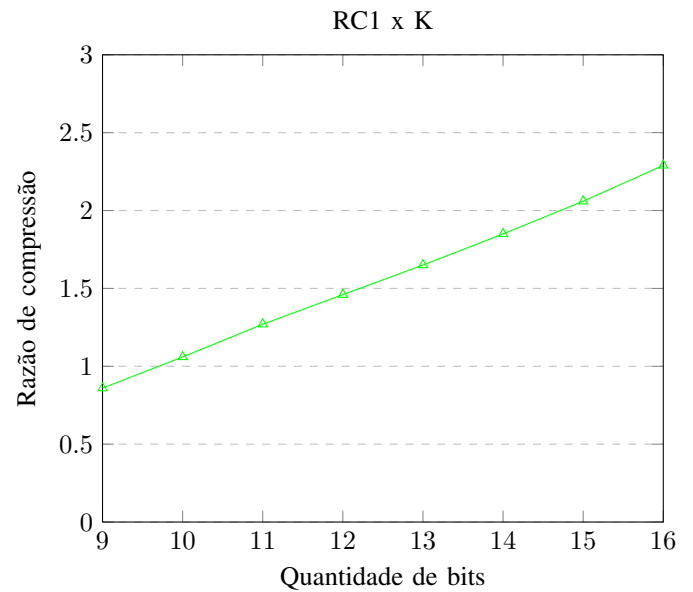
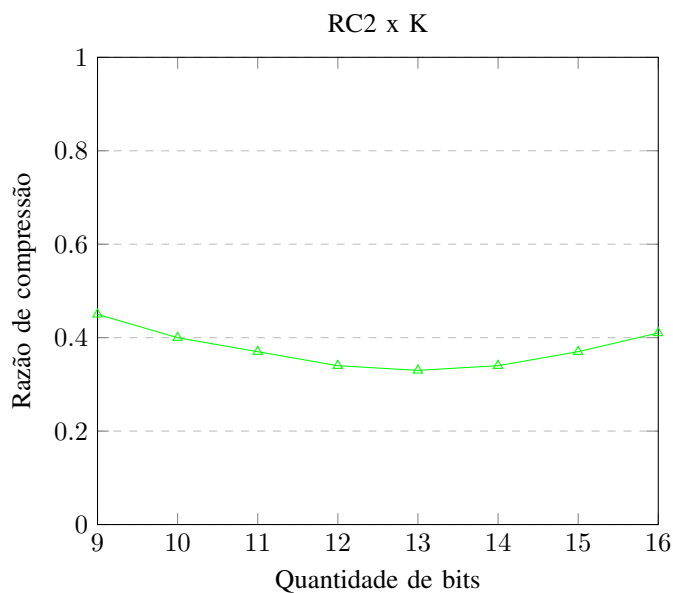
$$RC2 = \frac{tamArgOriginal}{\frac{totalIndices * K}{8}}$$

5.1. Curva dos 'RCs' x K para o arquivo de vídeo

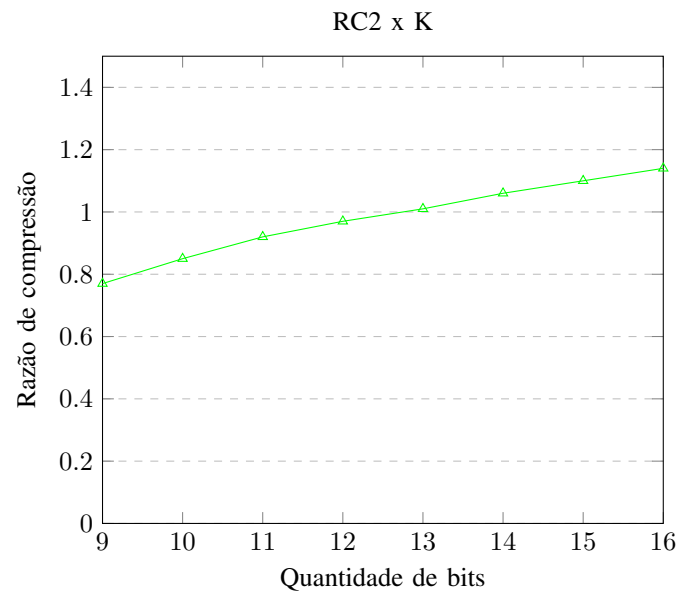
Para o RC1 obtivemos o seguinte gráfico:



Para o RC2 obtivemos o seguinte gráfico:



Para o caso do RC2 obtivemos o gráfico a seguir:



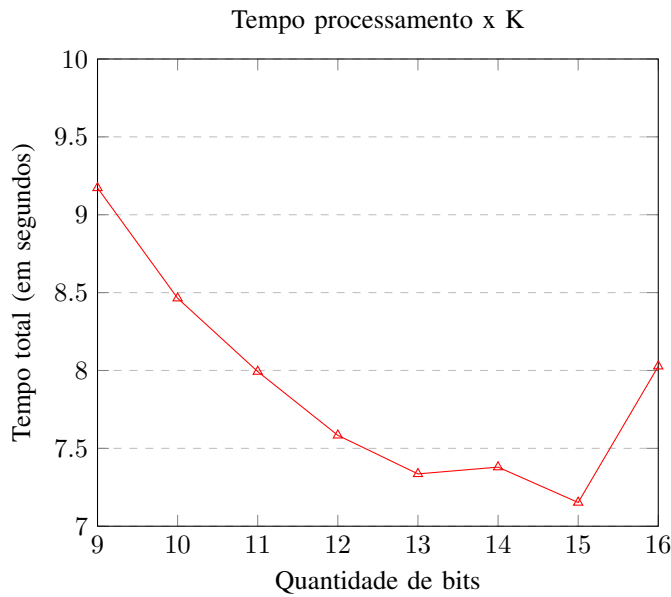
5.2. Curva dos 'RCs' x K para o arquivo de texto

Para o caso do RC1 obtivemos o gráfico a seguir:

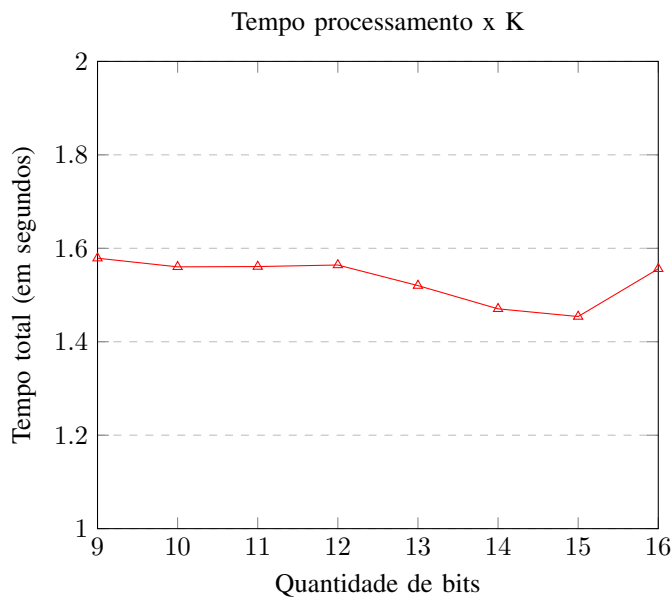
6. Curvas de Tempo x K

Para esta análise utilizamos a soma de todo o tempo de processamento do algoritmo.

6.1. Curva de Tempo x K para o arquivo de texto



6.2. Curva de Tempo x K para o arquivo de vídeo



7. Estaticidade do dicionário

Depois que o dicionário atinge o seu tamanho máximo, ele permanece estático. Salientando que seu valor máximo depende do 2^k , temos como valor máximo $2^{16} = 65536$