

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE SANTA CATARINA
CAMPUS SÃO JOSÉ

RHENZO HIDEKI SILVA KAJIKAWA

**ESTUDO E IMPLEMENTAÇÃO DOS ALGORITMOS DE
COMPRESSÃO LZ77 E CODIFICAÇÃO ARITMÉTICA NA
BIBLIOTECA KOMM**

SÃO JOSÉ

2025

Rhenzo Hideki Silva Kajikawa

Estudo e Implementação dos Algoritmos de Compressão LZ77 e
Codificação Aritmética na Biblioteca Komm

Monografia apresentada ao Curso de Engenharia de Telecomunicações do Instituto Federal de Santa Catarina, para a obtenção do título de bacharel em Engenharia de Telecomunicações.

Área de concentração: Telecomunicações

Orientador: Prof. Roberto Wanderley da Nóbrega, Dr.

São José

2025

Rhenzo Hideki Silva Kajikawa

Estudo e Implementação dos Algoritmos de Compressão LZ77 e
Codificação Aritmética na Biblioteca Komm

Monografia apresentada ao Curso de Engenharia de Telecomunicações do Instituto Federal de Santa Catarina, para a obtenção do título de bacharel em Engenharia de Telecomunicações.

São José, 16 de abril de 2025.

Prof. Roberto Wanderley da Nóbrega, Dr.
Instituto Federal de Santa Catarina

Professor Fulano, Dr.
Instituto Federal de Santa Catarina

Professora Fulana, Dra.
Instituto Federal de Santa Catarina

Aos meus pais, que sempre acreditaram em mim, e me ensinaram a sonhar.

AGRADECIMENTOS

Agradeço à minha família, que sempre esteve ao meu lado, me apoiando e incentivando a seguir em frente. Vocês são a minha base e a razão pela qual busco sempre o melhor.

Agradeço especialmente ao meu orientador, Professor Fulano, pela orientação e apoio durante todo o processo de elaboração deste trabalho. Sua experiência e conhecimento foram fundamentais para o meu aprendizado e crescimento acadêmico.

Agradeço ao Instituto Federal de Santa Catarina, que me proporcionou uma formação sólida e de qualidade, e a todos os professores que contribuíram para o meu aprendizado.

Agradeço também aos meus colegas de curso, que compartilharam comigo momentos de aprendizado e crescimento. Juntos, enfrentamos os desafios e celebramos as conquistas.

“Sempre que te perguntarem se podes fazer um trabalho, respondas que sim e te ponhas em seguida a aprender como se faz.” (NOME, ANO)

RESUMO

A biblioteca de compressão de dados Komm será estendida com a integração de dois algoritmos sem perdas: LZ77 e codificação aritmética. O cenário de redes de telecomunicações, marcado pelo aumento exponencial de dados, exige técnicas eficientes de compressão para otimização de largura de banda e armazenamento. Os objetivos englobam a estudo teórica dos algoritmos, projeto e implementação de módulos na arquitetura existente em Python, documentação e validação por meio de testes automatizados de correto funcionamento e desempenho. A metodologia inclui revisão bibliográfica, desenvolvimento de software e análise comparativa, visando quantificar ganhos em taxa de compressão, tempo de execução e uso de memória. Espera-se comprovar a viabilidade prática dos algoritmos e oferecer subsídios para futuras ampliações da biblioteca.

Palavras-chave: Compressão sem perdas; LZ77; codificação aritmética; Python.

ABSTRACT

The Komm data compression library will be extended by integrating two lossless algorithms: LZ77 and arithmetic coding. In the telecommunications network context, characterized by exponential data growth, efficient compression techniques are required to optimize bandwidth and storage. The objectives include a theoretical study of these algorithms, the design and implementation of modules in the existing Python architecture, documentation, and validation through automated testing to ensure proper functioning and performance. The methodology encompasses a literature review, software development, and comparative analysis, aiming to quantify improvements in compression ratio, execution time, and memory usage. It is expected that the practical feasibility of the algorithms will be demonstrated, providing a foundation for future expansions of the library.

Keywords: Lossless compression; LZ77; arithmetic coding; Python.

LISTA DE ILUSTRAÇÕES

| | |
|---|----|
| Figura 1 – Divisão da janela deslizando no algoritmo LZ77. | 16 |
| Figura 2 – Estado inicial da janela deslizando no algoritmo LZ77 | 17 |
| Figura 3 – Estado da janela após primeira codificação no algoritmo LZ77 | 18 |
| Figura 4 – Decodificação do exemplo $\langle 7, 4, \mathbf{r} \rangle$ | 19 |
| Figura 5 – Imagem bitmap (BMP) <i>smiley</i> | 26 |
| Figura 6 – Imagem bitmap (BMP) <i>snail</i> | 27 |
| Figura 7 – Texto <i>Alice</i> : taxa de compressão (menor é melhor). | 28 |
| Figura 8 – Texto <i>Alice</i> : memória pico. | 28 |
| Figura 9 – Texto <i>Alice</i> : tempo de compressão. | 28 |
| Figura 10 – Imagem <i>smiley</i> : taxa de compressão (menor é melhor). | 29 |
| Figura 11 – Imagem <i>smiley</i> : memória pico. | 29 |
| Figura 12 – Imagem <i>smiley</i> : tempo de compressão. | 30 |
| Figura 13 – Imagem <i>snail</i> : taxa de compressão (menor é melhor). | 30 |
| Figura 14 – Imagem <i>snail</i> : memória pico. | 30 |
| Figura 15 – Imagem <i>snail</i> : tempo de compressão. | 31 |
| Figura 16 – Texto <i>Alice</i> : taxa de compressão (menor é melhor). | 32 |
| Figura 17 – Texto <i>Alice</i> : memória pico. | 32 |
| Figura 18 – Texto <i>Alice</i> : tempo de compressão. | 33 |
| Figura 19 – Imagem <i>smiley</i> : taxa de compressão (menor é melhor). | 33 |
| Figura 20 – Imagem <i>smiley</i> : memória pico. | 34 |
| Figura 21 – Imagem <i>smiley</i> : tempo de compressão. | 34 |

LISTA DE QUADROS

| | |
|--|----|
| Quadro 1 – Resumo de tempo e memória na compressão da imagem <i>smiley</i> (implementações na <i>Komm</i>). | 31 |
|--|----|

LISTA DE TABELAS

LISTA DE CÓDIGOS

| | |
|---|----|
| Código 3.1 – Exemplo de utilização do <i>LZ77</i> na biblioteca <i>Komm</i> | 23 |
| Código 3.2 – Conversão intermediária entre sequência e tokens. | 23 |

SUMÁRIO

| | | |
|----------|---|-----------|
| 1 | INTRODUÇÃO | 13 |
| 1.1 | OBJETIVO GERAL | 13 |
| 1.2 | OBJETIVOS ESPECIFICOS | 14 |
| 1.3 | ORGANIZAÇÃO DO TEXTO | 14 |
| 2 | FUNDAMENTAÇÃO TEÓRICA | 15 |
| 2.1 | FUNCIONAMENTO DO LZ77 | 15 |
| 2.1.1 | Estrutura da janela deslizante | 15 |
| 2.1.2 | Formação dos tokens | 15 |
| 2.1.3 | Codificação e exemplo | 16 |
| 2.1.4 | Exemplo de Codificação com LZ77 | 17 |
| 2.2 | BIBLIOTECA KOMM | 21 |
| 3 | DESENVOLVIMENTO | 22 |
| 3.1 | MÓDULO LZ77 | 22 |
| 3.1.1 | Objetivos e escopo | 22 |
| 3.1.2 | Como utilizar | 22 |
| 3.1.3 | Como foi feito | 23 |
| 3.1.4 | Processo de busca na codificação (source_to_tokens) | 24 |
| 3.1.5 | Testes e validação | 24 |
| 4 | RESULTADOS | 26 |
| 4.1 | CONJUNTOS DE DADOS E PROTOCOLO EXPERIMENTAL | 26 |
| 4.2 | MÉTRICAS DE AVALIAÇÃO | 27 |
| 4.3 | RESULTADOS DO LZ77 NA <i>KOMM</i> | 27 |
| 4.4 | COMPARAÇÃO COM IMPLEMENTAÇÕES EXTERNAS DE LZ77 | 31 |
| 4.5 | DISCUSSÃO | 35 |
| | Referências | 36 |
| | APÊNDICE A – MEU PRIMEIRO APÊNDICE | 37 |
| | ANEXO A – MEU PRIMEIRO ASSUNTO DE ANEXO | 38 |
| | ANEXO B – SEGUNDO ASSUNTO QUE PESQUISEI | 39 |

1 INTRODUÇÃO

A compressão de dados é essencial para minimizar custos de armazenamento e transmissão, reduzindo o volume de dados sem comprometer o entendimento da informação. No caso da compressão sem perdas, segundo a teoria de Shannon, a entropia da fonte estabelece o limite inferior para a taxa média de bits de qualquer esquema de compressão (MACKAY, 2003). Para aproximar-se desse limite teórico, algoritmos de compressão combinam técnicas de dicionário e codificação estatísticas.

Na prática, formatos de compressão amplamente utilizados empregam essa abordagem híbrida. Por exemplo, o algoritmo DEFLATE, utilizado no formato ZIP, aplica LZ77 em conjunto com a codificação de Huffman para obter compressões melhores. Essa combinação explora tanto os padrões repetitivos de longo alcance quanto as probabilidades de ocorrência dos símbolos, elevando a eficiência global do método (DEUTSCH, 1996).

De modo geral, os algoritmos de compressão sem perdas dividem-se em métodos de codificação estatística e métodos de dicionário (SAYOOD, 2012). Por exemplo, a codificação de Huffman e a codificação aritmética são técnicas estatísticas, a última capaz de representar toda a mensagem como um único número fracionário no intervalo $[0, 1[$, alcançando compressões muito próximas do limite de entropia. Por sua vez, os métodos de dicionário exploram redundâncias substituindo sequências repetitivas por referências a ocorrências anteriores já vistas: um dos mais conhecidos é o algoritmo LZ77, proposto por Ziv e Lempel (ZIV; LEMPEL, 1977), que implementa esse princípio construindo dinamicamente um “dicionário” de padrões enquanto lê os dados.

Apesar do uso disseminado dessas técnicas em aplicações, a biblioteca de código aberto Komm não dispunha até o momento de uma implementação do LZ77. Essa biblioteca voltada ao ensino e à simulação em comunicação digital, já inclui diversos algoritmos clássicos de compressão, como os códigos de Huffman, Shannon–Fano e LZ78, evidenciando a relevância de incorporar as técnicas LZ77 para torná-la mais completa. Portanto, este trabalho tem como objetivo desenvolver e integrar um versão do LZ77 na biblioteca Komm, avaliando estes algoritmos em termos de taxa de compressão, tempo de processamento e uso de memória.

1.1 OBJETIVO GERAL

Expandir as capacidades da biblioteca Komm com a implementação dos algoritmos LZ77.

1.2 OBJETIVOS ESPECIFICOS

- Projetar e desenvolver módulos da codificação LZ77 na arquitetura da Komm, observando padrões de codificação e cobertura de testes e escrita da documentação.
- Validar a compressão e descompressão em arquivos de textos e imagens, assegurando corretude e integridade.
- Realizar análise comparativa de desempenho (taxa de compressão, tempo de execução e uso de memória) em relação a Huffman, LZ78 e LZW.

1.3 ORGANIZAÇÃO DO TEXTO

A ser decidido conforme escrito o TCC

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção será discutido o funcionamento e as principais características do LZ77, com o propósito de permitir ao leitor uma melhor compreensão dos algoritmos utilizados neste trabalho.

2.1 FUNCIONAMENTO DO LZ77

O algoritmo de compressão LZ77, proposto por Ziv e Lempel em 1977 (ZIV; LEMPEL, 1977), pertence à classe dos métodos baseados em dicionário. Sua principal característica é o uso de uma **janela deslizando** de comprimento fixo W , que percorre sequencialmente a entrada e permite substituir repetições por referências a ocorrências anteriores.

2.1.1 Estrutura da janela deslizando

A janela deslizando possui comprimento total W e é dividida em duas partes:

- ***Search buffer*** — de comprimento S : contém a porção da sequência já processada, isto é, o passado disponível para referência;
- ***Lookahead buffer*** — de comprimento L : contém os próximos símbolos a serem codificados, representando o futuro imediato;

de modo que:

$$W = S + L.$$

Esses três parâmetros (W , S , L) são comprimentos relevantes para o funcionamento do algoritmo. W determina o tamanho da janela total, S limita o histórico de busca, e L define o tamanho máximo de sequência a ser codificada em um único passo.

2.1.2 Formação dos tokens

Durante a codificação, o algoritmo busca no *search buffer* o maior prefixo, sendo a maior sequência de caracteres iguais, que ocorra dentro do *lookahead buffer*. O resultado dessa busca é codificado como um **token** $\langle p, \ell, x \rangle$, onde:

- p (*pointer*) é a posição, contada a partir do **início** do *search buffer*, em que ocorre a correspondência mais longa;
- ℓ (*length*) é o comprimento da sequência coincidente, com $0 \leq \ell \leq L$;

- $x \in \mathcal{X}$ é o próximo símbolo literal, que segue a sequência copiada e garante a continuidade da decodificação.

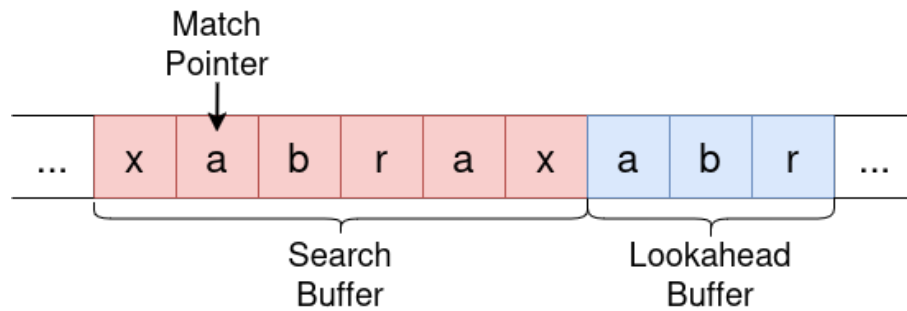
A saída do codificador é, portanto, uma sequência de tokens $\langle p_i, \ell_i, x_i \rangle$, que representam os fatores sucessivos do texto de entrada.

Origem do ponteiro.

Seguindo a convenção do artigo original de Ziv e Lempel (ZIV; LEMPEL, 1977), o ponteiro p é medido a partir do **início** do *search buffer*. Essa escolha difere de algumas implementações posteriores, nas quais p é medido a partir do **fim** do buffer, mas ambas produzem resultados equivalentes.

A Figura 1 ilustra a estrutura da janela deslizante e o processo de busca pelo maior prefixo coincidente.

Figura 1 – Divisão da janela deslizante no algoritmo LZ77.



Fonte: Adaptada de Sayood (2012).

2.1.3 Codificação e exemplo

O processo de codificação pode ser descrito como:

1. Identificar no *search buffer* o maior trecho que coincide com o início do *lookahead buffer*;
2. Gerar o token $\langle p, \ell, x \rangle$, em que x é o próximo símbolo literal após a correspondência;
3. Avançar a janela em $\ell + 1$ posições e repetir o processo.

Como exemplo, considere a sequência "cabracadabrarrarrad", com parâmetros $W = 13$ e $L = 6$. O *search buffer* inicial contém "cabraca". O primeiro caractere a ser codificado é "d", para o qual não há correspondência anterior, resultando no token $\langle 6, 0, d \rangle$. Após o avanço da janela, o algoritmo encontra no histórico a sequência "abra", gerando o token $\langle 0, 4, r \rangle$, e assim sucessivamente.

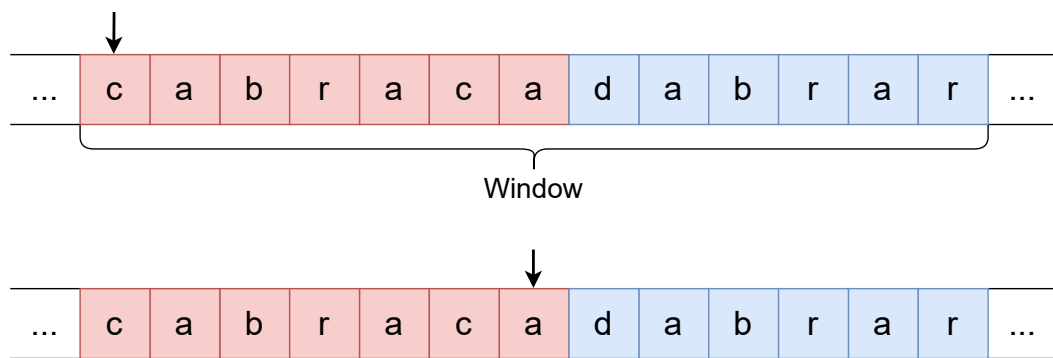
Durante a decodificação, cada token é expandido de acordo com p e ℓ , copiando o trecho correspondente do histórico decodificado e adicionando o símbolo literal x . Dessa forma, o processo é perfeitamente reversível e sem perdas.

2.1.4 Exemplo de Codificação com LZ77

Nesta seção será demonstrado um exemplo detalhado do funcionamento da janela deslizante e da representação do token $\langle p, \ell, x \rangle$ utilizados pelo algoritmo LZ77.

A sequência a ser codificada é "cabracadabrarrarrad". Para este exemplo, a janela deslizante possui um tamanho total fixo de 13 caracteres, com o *lookahead buffer* definido em 6 caracteres e search buffer de 7 caracteres. O estado inicial da janela pode ser visto na Figura 2.

Figura 2 – Estado inicial da janela deslizante no algoritmo LZ77



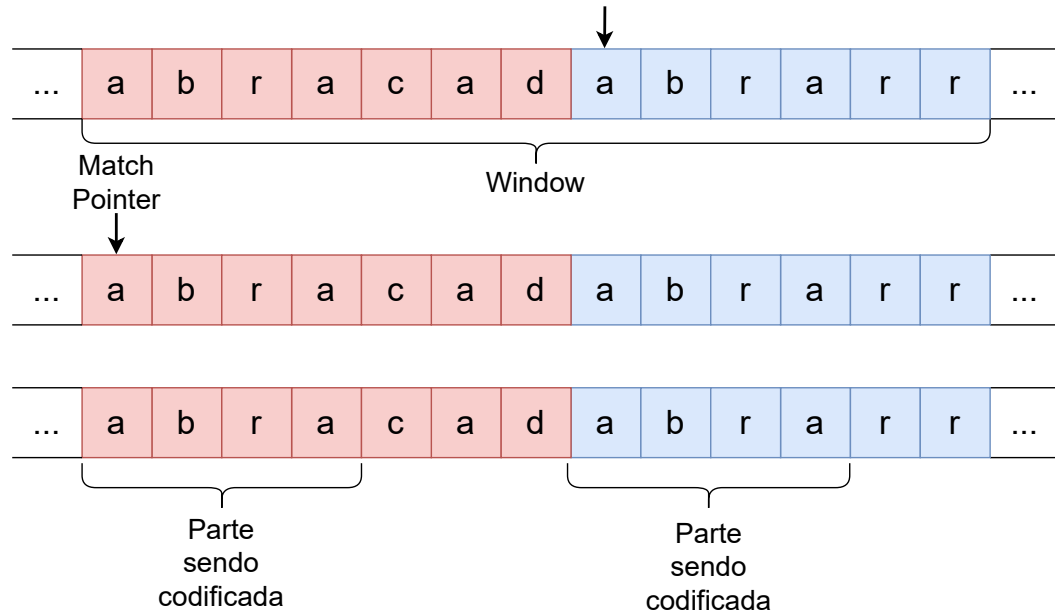
Fonte: Adaptada de Sayood (2012)

Inicialmente, busca-se no *search buffer*, da esquerda para direita, algum caractere ou sequência que coincida com o início do *lookahead buffer*. Neste momento, deseja-se codificar o primeiro caractere do *lookahead buffer*, que é "d". Ao observar o *search buffer*, verifica-se que não há nenhuma correspondência prévia para este caractere. Portanto, o algoritmo gera o token $\langle 6, 0, d \rangle$, indicando que não houve correspondência, apesar de o pointer ser 6, o comprimento da sequência é 0, assim não fazendo a diferença na codificação, pois não existe cópia sendo feita.

Após esta codificação inicial, a janela deslizante avança uma posição, o que altera o conteúdo tanto do *search buffer* quanto do *lookahead buffer*, conforme mostrado na Figura 3.

Nesse novo estado, procura-se novamente uma correspondência no *search buffer* para a sequência do *lookahead buffer*, que agora começa com "a". Observando o *search buffer*, é possível encontrar múltiplas ocorrências isoladas do caractere "a", próximo a esquerda. Notavelmente, existe uma sequência completa "abra" previamente codificada,

Figura 3 – Estado da janela após primeira codificação no algoritmo LZ77



Fonte: Adaptada de Sayood (2012)

iniciando a 0 caracteres de distância da posição atual da janela. Essa correspondência possui comprimento 4 caracteres.

Dessa forma, o algoritmo codifica a sequência encontrada como o token $\langle 0, 4, r \rangle$, onde 0 indica a distância até o início da correspondência no *search buffer*, 4 indica o comprimento da correspondência encontrada ("abra"), e "r" é o caractere seguinte imediatamente após essa sequência, ainda não codificado. Após isso, a janela avança em 5 posições (4 caracteres da sequência codificada mais 1 caractere literal).

Para realizar o processo inverso, ou seja, decodificar o token recebido $\langle 0, 4, r \rangle$, o decodificador utiliza o mesmo princípio do algoritmo LZ77, porém no sentido inverso.

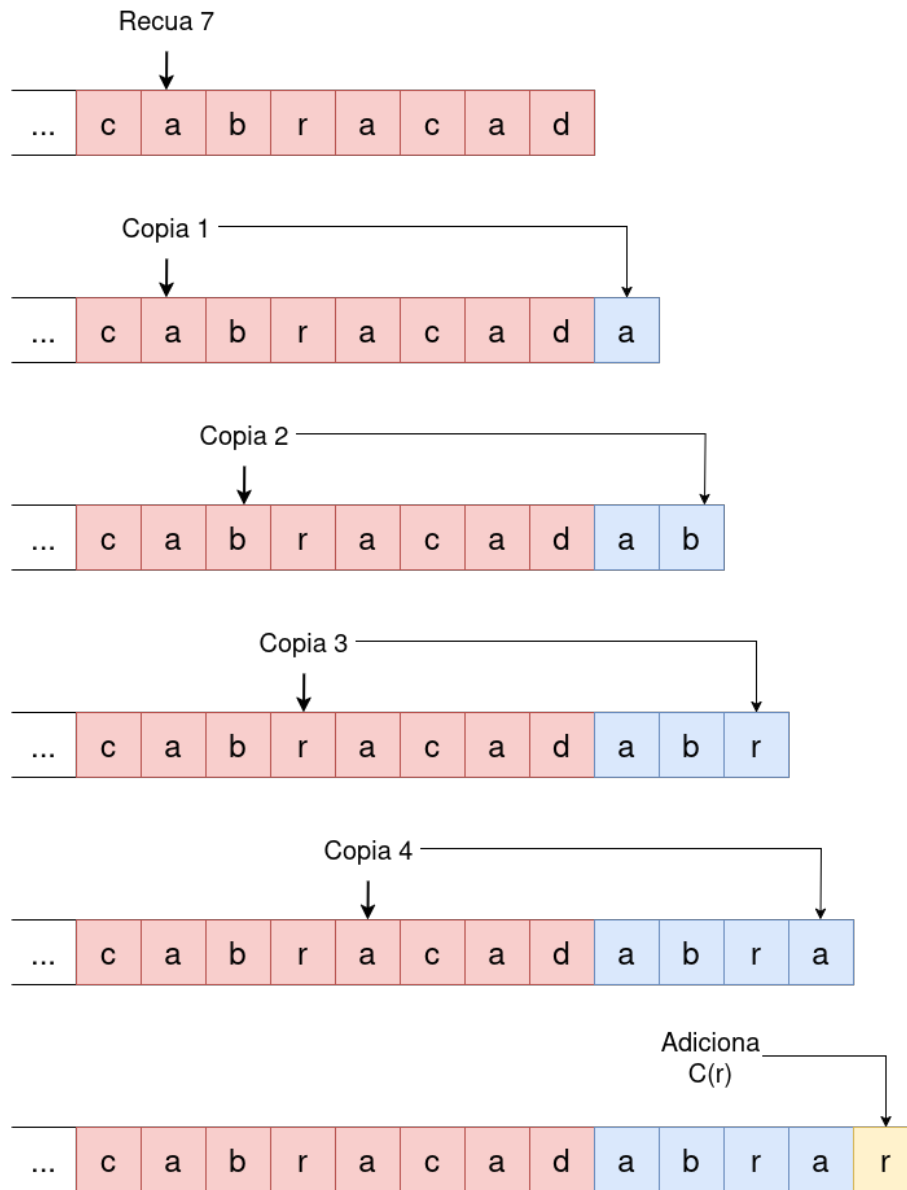
Inicialmente, ele utiliza o pointer (p) para retornar exatamente 0 posições na sequência já decodificada até o momento. A partir dessa posição inicial encontrada, copia-se uma sequência de comprimento 4 (valor l), obtendo o trecho "abra". Em seguida, adiciona-se ao final desta sequência copiada o caractere literal adicional (c), que neste exemplo é "r".

Esse processo é ilustrado passo a passo na Figura 4. Inicialmente, há o estado parcial da decodificação com o *buffer* já reconstruído. Em seguida, avança-se caractere a caractere, copiando-se do *buffer* reconstruído e adicionando o caractere literal no final. O resultado final após a decodificação deste token será "abrar".

Vale destacar que o decodificador reconstrói o buffer de busca dinamicamente, conforme recebe e processa novos tokens, permitindo a reconstrução exata dos dados

originais sem perda alguma.

Figura 4 – Decodificação do exemplo $\langle 7, 4, r \rangle$



Fonte: Adaptada de Sayood (2012)

Utilizando o exemplo anterior do token $\langle 7, 4, r \rangle$, onde a janela total possui 13 caracteres, temos a seguinte definição dos campos em bits:

- O deslocamento (p) pode variar de 0 a 6, portanto requer 3 bits;
- O comprimento (ℓ) pode variar de 0 a 7, logo exige também 3 bits;
- O caractere literal (c) é codificado em ASCII, utilizando 8 bits (1 byte).

Dado que na implementação adotada os tamanhos em bits para cada campo do token são

$$\underbrace{3}_{\substack{\text{bits para} \\ \text{pointer } (p)}} + \underbrace{3}_{\substack{\text{bits para} \\ \text{length } (\ell)}} + \underbrace{8}_{\substack{\text{bits para} \\ \text{character } (x)}} = 14 \text{ bits},$$

temos um formato de código de comprimento fixo, no qual cada token $\langle p, \ell, x \rangle$ ocupará exatamente 14 bits. Em outras palavras, após definido o tamanho da janela (*pointer*) e do *lookahead buffer* (*length*), bem como o padrão de 8 bits para o caractere literal, toda e qualquer codificação produzida por esse esquema terá comprimento contínuo de 14 bits por símbolo codificado (NELSON, 2008).

Realizando a codificação especificamente para o exemplo dado ($\langle 0, 4, r \rangle$), obtêm-se:

- O valor do *pointer* $o = 0$, em 3 bits é $000_{(2)}$.
- O comprimento $l = 4$, em 3 bits é $100_{(2)}$.
- O caractere r , em ASCII binário (8 bits), é $01110010_{(2)}$.

Portanto, a representação completa do token em bits será:

$$000 \mid 100 \mid 01110010$$

Resultando na sequência binária final: $00010001110010_{(2)}$.

Esse processo ilustra o funcionamento fundamental do algoritmo LZ77, mostrando como ele explora redundâncias por meio de correspondências encontradas em trechos já codificados, reduzindo o volume de dados transmitidos ou armazenados.

2.2 BIBLIOTECA KOMM

A *Komm* é uma biblioteca em *Python* para sistemas de comunicação desenvolvida pelo professor Roberto Nobrega. Trata-se de um projeto *open-source* para *Python 3* que fornece ferramentas para análise e simulação de sistemas de comunicação analógicos e digitais.

Este projeto foi inspirado em diversas outras bibliotecas para sistemas de comunicação, como *MATLAB® Communications System Toolbox™*, *GNU Radio*, *CommPy* e *SageMath*.

Além disso, a biblioteca já disponibiliza diversas codificações implementadas, abrangendo diferentes técnicas de *source coding* e *lossless coding*, incluindo:

- Shannon Code
- Fano Code
- Huffman Code
- Tunstall lCode
- LZ78
- LZW

3 DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento do módulo `LempelZiv77Code`, integrado à biblioteca *Komm*. A implementação foi realizada em Python, linguagem-base do projeto, buscando equilíbrio entre clareza didática e conformidade com a arquitetura existente da biblioteca.

3.1 MÓDULO LZ77

Esta seção detalha a implementação do algoritmo *Lempel–Ziv 77* (LZ77), conforme descrito originalmente por Ziv e Lempel (ZIV; LEMPEL, 1977), e sua integração na biblioteca *Komm*¹. A *Komm* já disponibiliza diversos esquemas de compressão sem perdas, como Huffman, Tunstall, LZ78 e LZW, mas ainda não possuía uma implementação do LZ77. Assim, a principal contribuição deste trabalho foi o desenvolvimento de uma versão completa e modular do algoritmo, com foco tanto prático quanto educacional.

3.1.1 Objetivos e escopo

Os objetivos deste módulo foram:

- implementar as rotinas de codificação (`encode`) e decodificação (`decode`), compatíveis com o padrão interno da *Komm*;
- expor funções intermediárias para inspeção didática, permitindo a visualização dos tokens gerados e dos fluxos de dados nas diferentes etapas;
- validar a corretude do módulo por meio de testes unitários e de *round trip* (`decode(encode(x)) == x`);
- documentar e integrar o código ao padrão da biblioteca.

3.1.2 Como utilizar

A classe `LempelZiv77Code` implementa a codificação e decodificação completa do algoritmo. Sua utilização segue o mesmo formato das demais classes de codificação da *Komm*. O construtor requer quatro parâmetros principais: a cardinalidade do alfabeto de entrada (`source_cardinality`), a cardinalidade do alfabeto de saída (`target_cardinality`), o tamanho da janela deslizante total (`window_size`) e o tamanho do *lookahead buffer* (`lookahead_size`). Opcionalmente, pode-se fornecer um conteúdo inicial para o *search buffer*, útil em experimentos ou testes.

¹ <https://komm.dev/ref/LempelZiv77Code>

Código 3.1 – Exemplo de utilização do *LZ77* na biblioteca *Komm*.

```

1 import komm
2
3 # Instanciação do codificador
4 lz77 = komm.LempelZiv77Code(
5     window_size=13,
6     lookahead_size=6,
7     source_cardinality=256,
8     target_cardinality=2
9 )
10
11 # Exemplo de sequência de entrada
12 source = [ord(x) for x in "cabracadabrrarrarrad"]
13
14 # Codificação e decodificação
15 encoded = lz77.encode(source)
16 decoded = lz77.decode(encoded)
17
18 assert decoded == source

```

O método `encode()` retorna o fluxo comprimido no alfabeto de saída \mathcal{Y} , enquanto `decode()` reconstrói a sequência original no alfabeto de entrada \mathcal{X} . Para análise intermediária, a classe expõe os métodos `source_to_tokens()` e `tokens_to_source()`, que convertem a sequência de entrada em uma lista de tokens $\langle p, \ell, x \rangle$ e vice-versa.

Código 3.2 – Conversão intermediária entre sequência e tokens.

```

1 tokens = lz77.source_to_tokens(source)
2 print(tokens)
3 # [(6, 0, 100), (0, 4, 114), (4, 5, 100)]
4
5 reconstructed = lz77.tokens_to_source(tokens)
6 assert reconstructed == source

```

Esses métodos são especialmente úteis em contextos acadêmicos, pois permitem acompanhar o comportamento do codificador em cada estágio do processo.

3.1.3 Como foi feito

A arquitetura do módulo segue o padrão das demais classes de codificação da biblioteca. O código foi estruturado em quatro submódulos internos: dois relacionados à etapa de codificação e dois à etapa de decodificação.

1. Codificação:

- `source_to_tokens`: converte a sequência de entrada em tokens $\langle p, \ell, x \rangle$;

- `tokens_to_target`: transforma os tokens no fluxo final no alfabeto \mathcal{Y} .

2. Decodificação:

- `target_to_tokens`: reverte o fluxo comprimido de volta aos tokens;
- `tokens_to_source`: reconstrói a sequência original em \mathcal{X} .

Essa separação permite maior legibilidade, facilita testes unitários e torna a implementação mais transparente para uso didático e comparações com versões descritas na literatura.

3.1.4 Processo de busca na codificação (`source_to_tokens`)

O núcleo da codificação LZ77 está no método `source_to_tokens()`, que busca, dentro do *search buffer*, o maior trecho que coincida com o prefixo do *lookahead buffer*. Nesta implementação, a busca é feita de forma simples e eficiente, utilizando o método `rfind()` da classe `bytes` do Python — o que resulta em código mais limpo e desempenho superior sem o uso explícito de tabelas de hash.

A cada iteração, o algoritmo:

1. define o *search buffer* como o trecho de tamanho $S = W - L$;
2. tenta localizar a sequência mais longa do *lookahead* dentro do *search buffer*;
3. emite o token $\langle p, \ell, x \rangle$, onde p é o ponteiro (posição a partir do início da janela), ℓ é o comprimento da correspondência e x é o próximo símbolo literal.

Essa abordagem segue a convenção do artigo original de Ziv e Lempel, em que o ponteiro p é medido a partir do início da janela, diferindo da convenção moderna (offset medido a partir do fim do *search buffer*). A utilização do `rfind()` simplifica a busca, preservando a corretude e a clareza do código.

3.1.5 Testes e validação

Os testes unitários² abrangem tanto casos básicos quanto exemplos reproduzidos da literatura, verificando:

- a corretude da codificação e decodificação (*round trip*);
- o comportamento com sobreposição de trechos (*overlap*);
- a consistência dos tokens gerados em diferentes tamanhos de janela e *lookahead*.

² https://github.com/rwnobrega/komm/blob/main/tests/lossless_coding/test_lz77.py

Os resultados confirmam a consistência e compatibilidade do módulo com o padrão de codificação da *Komm*, assegurando sua integração com os demais algoritmos já presentes na biblioteca.

4 RESULTADOS

Este capítulo apresenta os experimentos realizados com a implementação do algoritmo *LZ77* desenvolvida na biblioteca *Komm*, bem como comparações com outras implementações externas do mesmo método. Os demais algoritmos disponíveis na *Komm* (Huffman, Shannon–Fano, LZ78 e LZW) foram utilizados apenas como referência para fins de contextualização dos resultados.

4.1 CONJUNTOS DE DADOS E PROTOCOLO EXPERIMENTAL

Foram utilizados dois tipos de arquivos: (i) **textos** e (ii) **imagens**. O texto escolhido foi o livro *Alice’s Adventures in Wonderland*, do Projeto Gutenberg¹, por se tratar de um corpus literário clássico amplamente utilizado em experimentos de compressão sem perdas. O arquivo possui tamanho de 154.573 bytes e apresenta ampla variedade de caracteres e repetições, sendo adequado para testar diferentes tamanhos de janela no *LZ77*.

As imagens utilizadas foram bitmaps (**BMP**) simples, escolhidas por possuírem regiões de baixa entropia e padrões espaciais redundantes, que permitem observar o comportamento do algoritmo em fontes bidimensionais. A Figura 5 mostra a imagem *smiley* (246 bytes) e a Figura 6 a imagem *snail* (196.666 bytes).

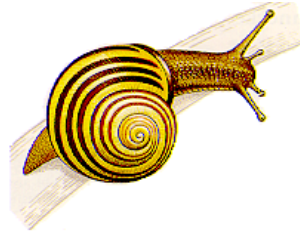
Figura 5 – Imagem bitmap (BMP) *smiley*.



Fonte: Adaptada de <https://cse1.net/recaps/graphics>.

¹ <https://www.gutenberg.org/ebooks/11>

Figura 6 – Imagem bitmap (BMP) *snail*.



Fonte: Adaptada de <https://people.math.sc.edu/Burkardt/data/bmp/bmp.html>.

4.2 MÉTRICAS DE AVALIAÇÃO

As seguintes métricas foram utilizadas para avaliar o desempenho dos algoritmos:

- **Taxa de compressão:** relação entre o tamanho comprimido e o tamanho original (quanto menor, melhor).
- **Tempo de compressão e descompressão:** medido em segundos.
- **Memória pico:** quantidade máxima de memória alocada durante a execução (em MB).
- **Integridade:** verificação de *round trip*, isto é, `decode(encode(x)) == x`.

4.3 RESULTADOS DO LZ77 NA KOMM

Foram consideradas duas implementações populares: **FastLZ²** (C) e **LZ77-Compressor³** (Python). As configurações efetivas foram: $W = 8$ KiB e $L = 264$ bytes no *FastLZ*; no *LZ77-Compressor*, W variou entre 100 e 400 bytes e $L = 15$ bytes. Na *Komm*, W e L são parametrizáveis; para comparação, utilizaram-se configurações próximas às disponíveis nas bibliotecas externas quando pertinente, além de uma configuração de referência com $W = 64$ kB.

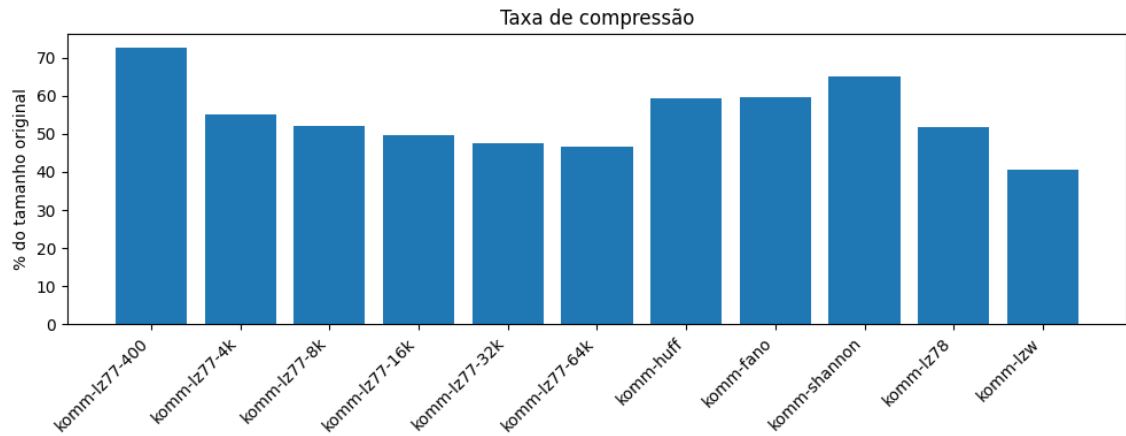
Análise (texto):

- O *LZ77* apresentou o *trade-off* esperado: janelas maiores proporcionam melhor compressão, mas aumentam o tempo e o consumo de memória.
- Para o arquivo *Alice*, o ganho de compressão é significativo ao aumentar W de 400 B para 32 kB, estabilizando a partir de 64 kB.

² <https://github.com/ariya/FastLZ>

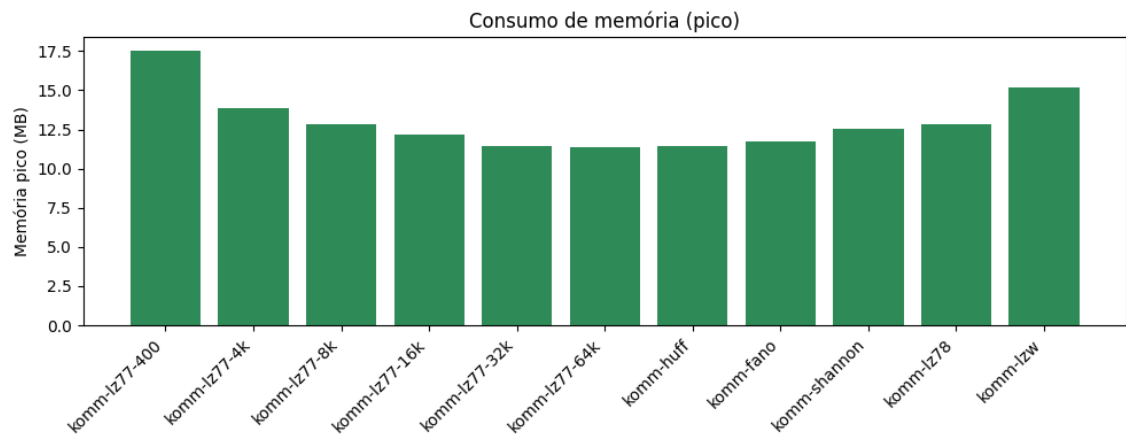
³ <https://github.com/ariya/FastLZ>

Figura 7 – Texto *Alice*: taxa de compressão (menor é melhor).



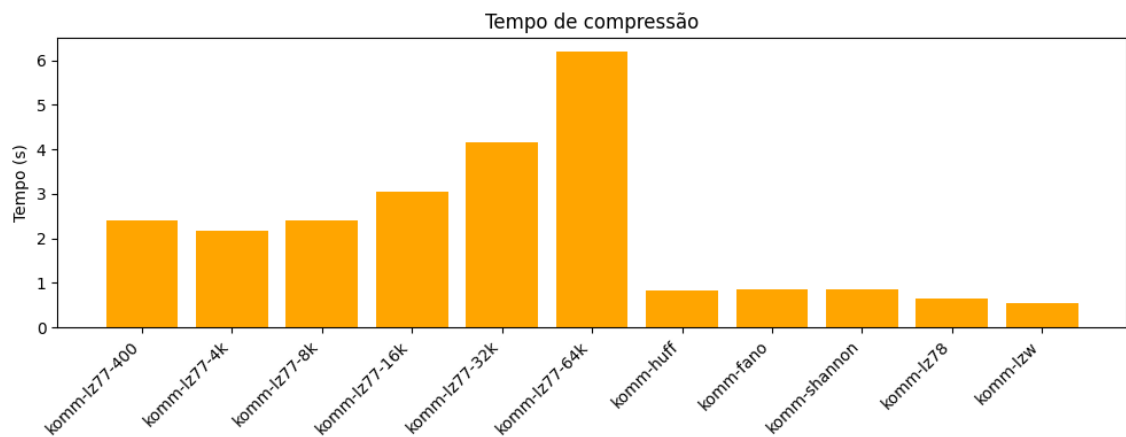
Fonte: Elaborada pelo autor.

Figura 8 – Texto *Alice*: memória pico.



Fonte: Elaborada pelo autor.

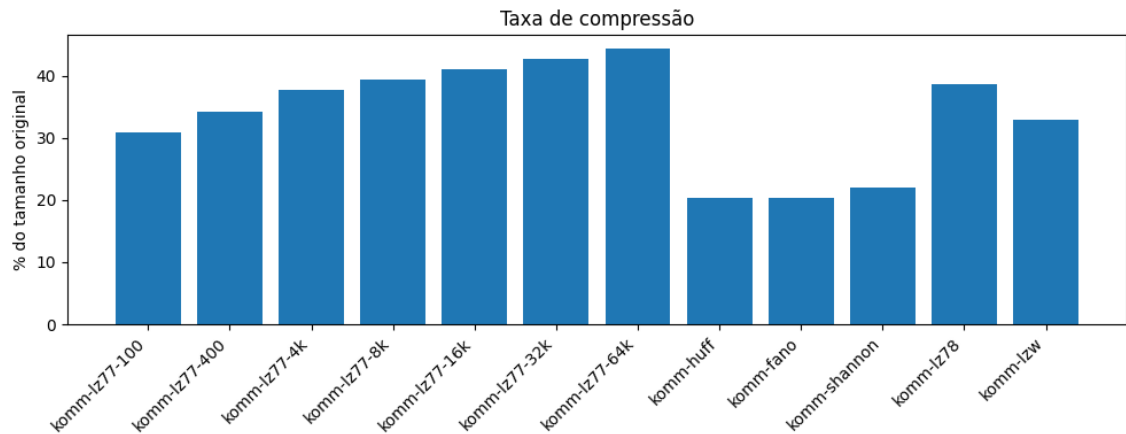
Figura 9 – Texto *Alice*: tempo de compressão.



Fonte: Elaborada pelo autor.

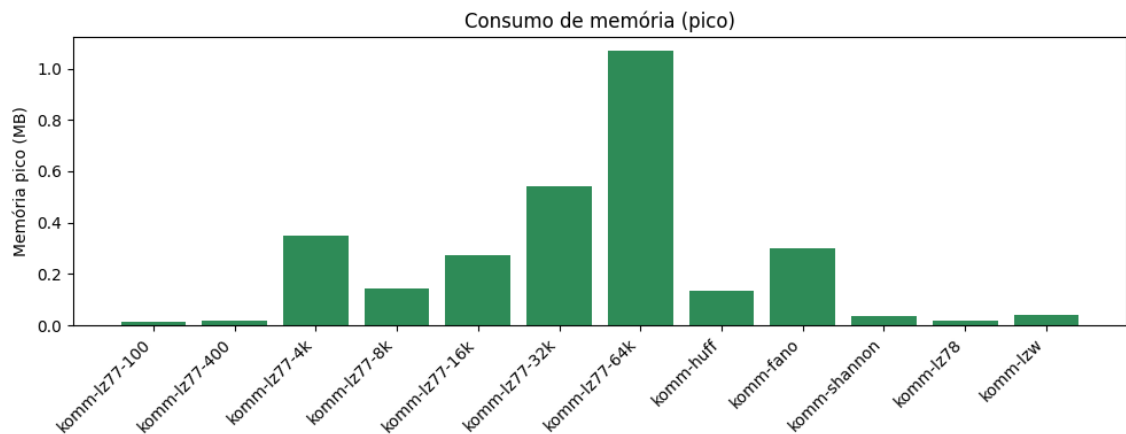
- Em comparação, Huffman e Shannon–Fano mantêm desempenho rápido e baixo uso de memória, porém com taxas de compressão inferiores.

Figura 10 – Imagem *smiley*: taxa de compressão (menor é melhor).



Fonte: Elaborada pelo autor.

Figura 11 – Imagem *smiley*: memória pico.

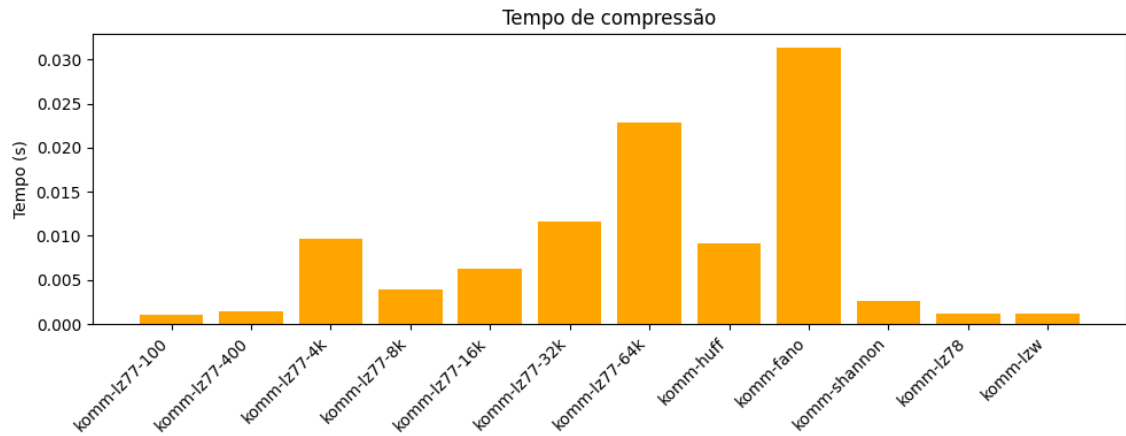


Fonte: Elaborada pelo autor.

Análise (imagens):

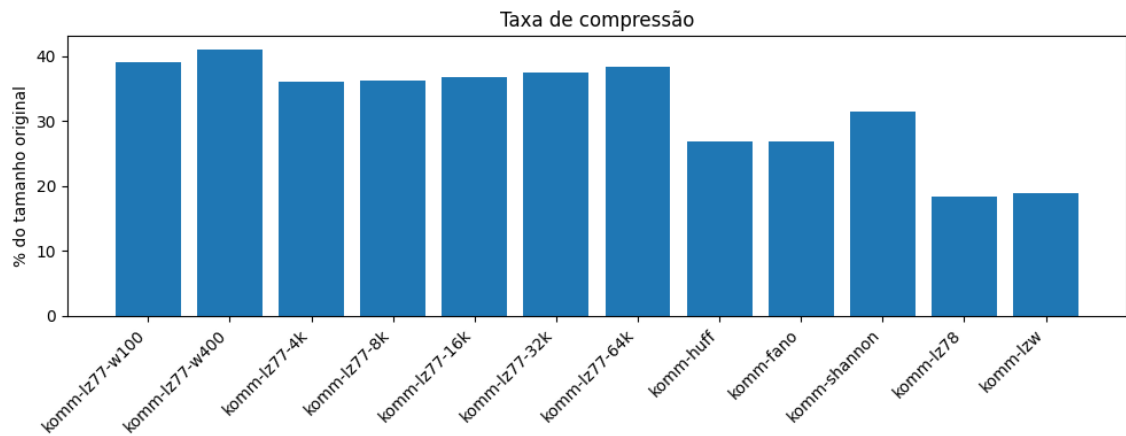
- Nas imagens *smiley* e *snail*, o *LZ77* apresentou menor ganho de compressão em comparação ao texto, o que é esperado dada a menor redundância sequencial.
- Janelas maiores continuam a reduzir ligeiramente a taxa de compressão, porém com crescimento do custo computacional.
- A relação entre tempo e memória manteve comportamento aproximadamente linear com o tamanho da janela.

Figura 12 – Imagem *smiley*: tempo de compressão.



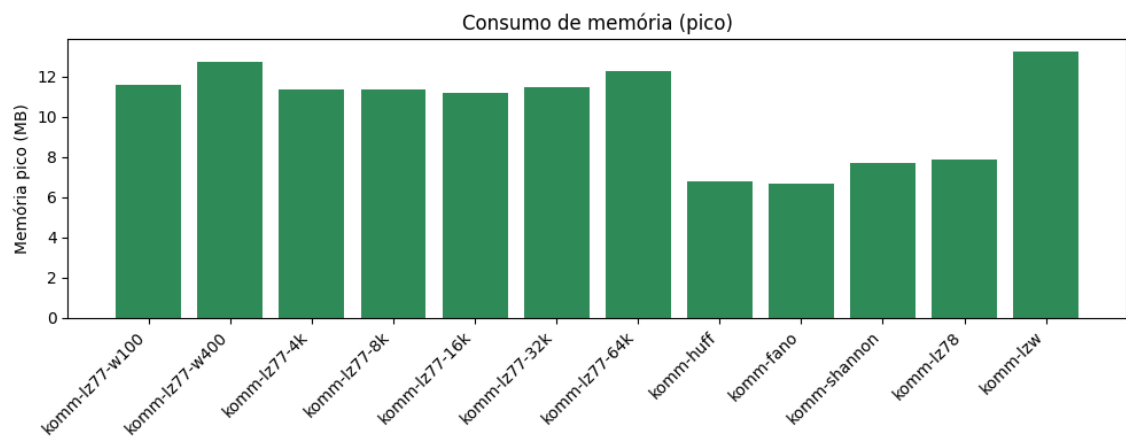
Fonte: Elaborada pelo autor.

Figura 13 – Imagem *snail*: taxa de compressão (menor é melhor).

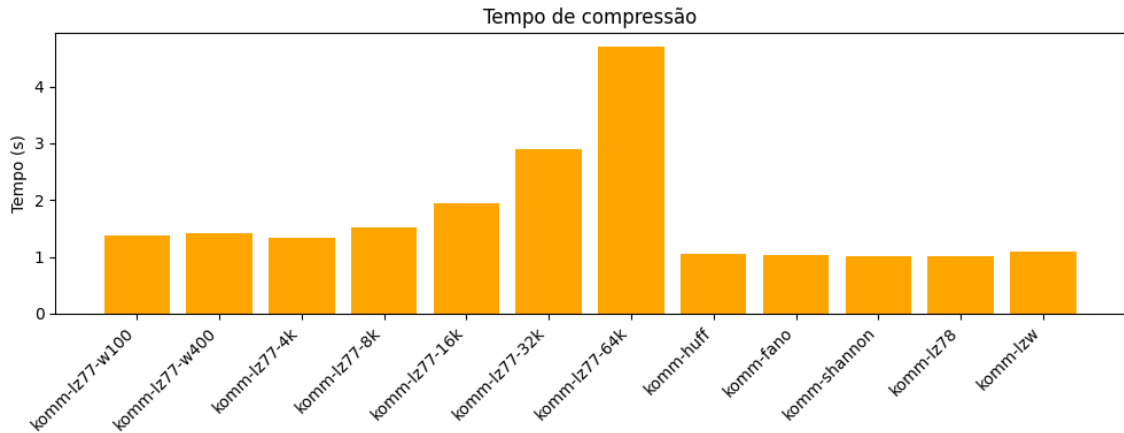


Fonte: Elaborada pelo autor.

Figura 14 – Imagem *snail*: memória pico.



Fonte: Elaborada pelo autor.

Figura 15 – Imagem *snail*: tempo de compressão.

Fonte: Elaborada pelo autor.

Quadro 1 – Resumo de tempo e memória na compressão da imagem *smiley* (implementações na *Komm*).

| Algoritmo | Tempo (s) | Memória pico (MB) |
|---------------|-----------|-------------------|
| komm-lz77-100 | 1,382 | 12,128 |
| komm-lz77-400 | 1,416 | 13,330 |
| komm-lz77-8k | 0,0039 | 148,218 |
| komm-lz77-16k | 0,0062 | 287,370 |
| komm-lz77-32k | 0,0115 | 566,106 |
| komm-lz77-64k | 0,0228 | 112,295 |
| komm-huff | 0,0092 | 140,961 |
| komm-fano | 0,0313 | 314,210 |
| komm-shannon | 0,0026 | 38,195 |
| komm-lz78 | 0,0011 | 19,928 |
| komm-lzw | 0,0011 | 43,084 |

Fonte: Elaborada pelo autor.

4.4 COMPARAÇÃO COM IMPLEMENTAÇÕES EXTERNAS DE LZ77

Foram consideradas duas implementações populares do algoritmo *LZ77*: **FastLZ**⁴ (em C) e **LZ77-Compressor**⁵ (em Python). As configurações padrão dessas bibliotecas são:

- **FastLZ**: janela de 8 kB e *lookahead* de 264 bytes;
- **LZ77-Compressor**: janela variável (100–400 bytes) e *lookahead* fixo de 15 bytes.

Na *Komm*, foi possível parametrizar W e L livremente, permitindo a replicação aproximada das condições dessas bibliotecas, além de uma configuração de referência com

⁴ <https://github.com/ariya/FastLZ>

⁵ <https://github.com/manassra/LZ77-Compressor>

$W = 64\text{ kB}$.

Figura 16 – Texto *Alice*: taxa de compressão (menor é melhor).

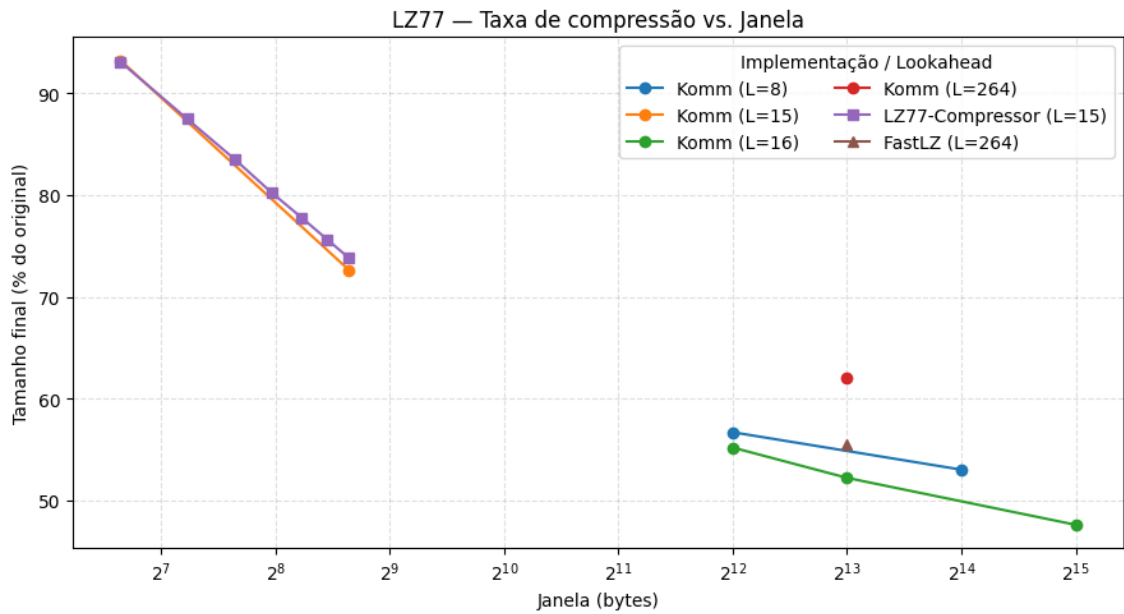
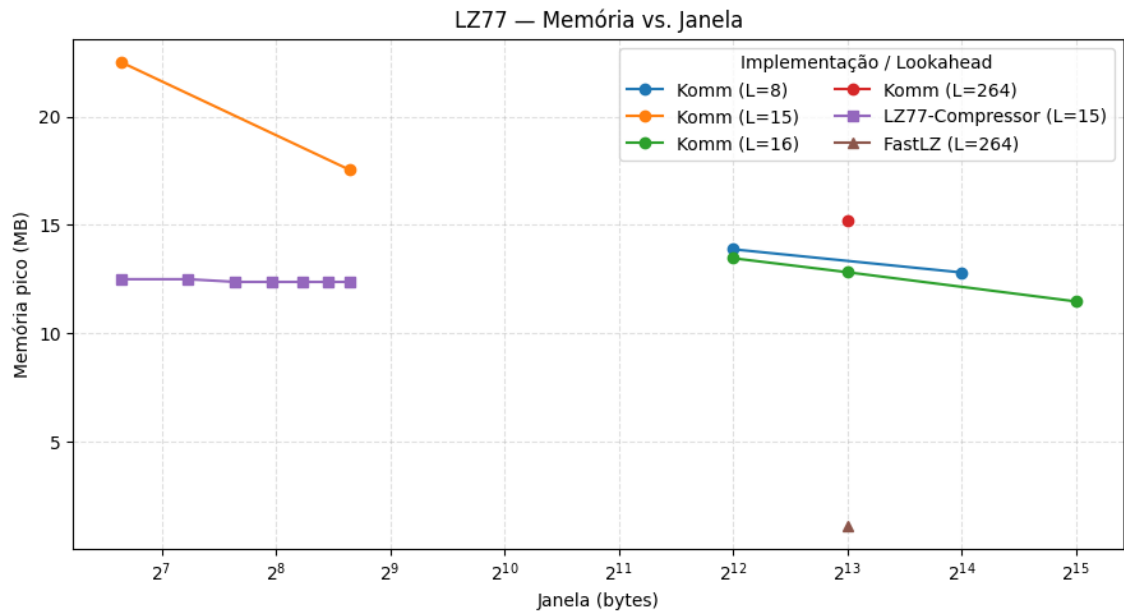
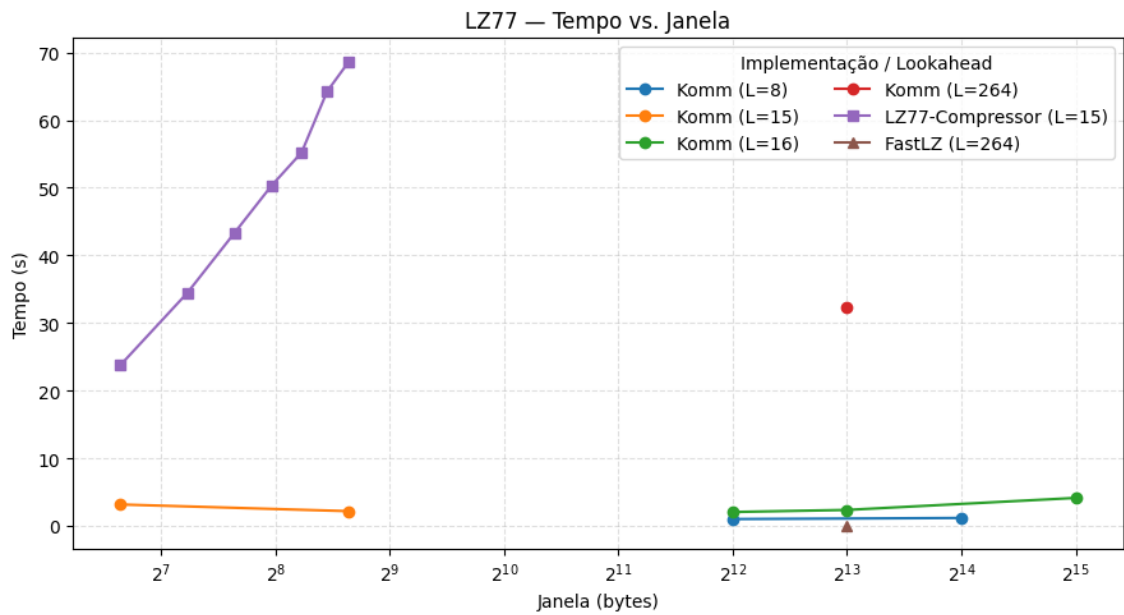


Figura 17 – Texto *Alice*: memória pico.



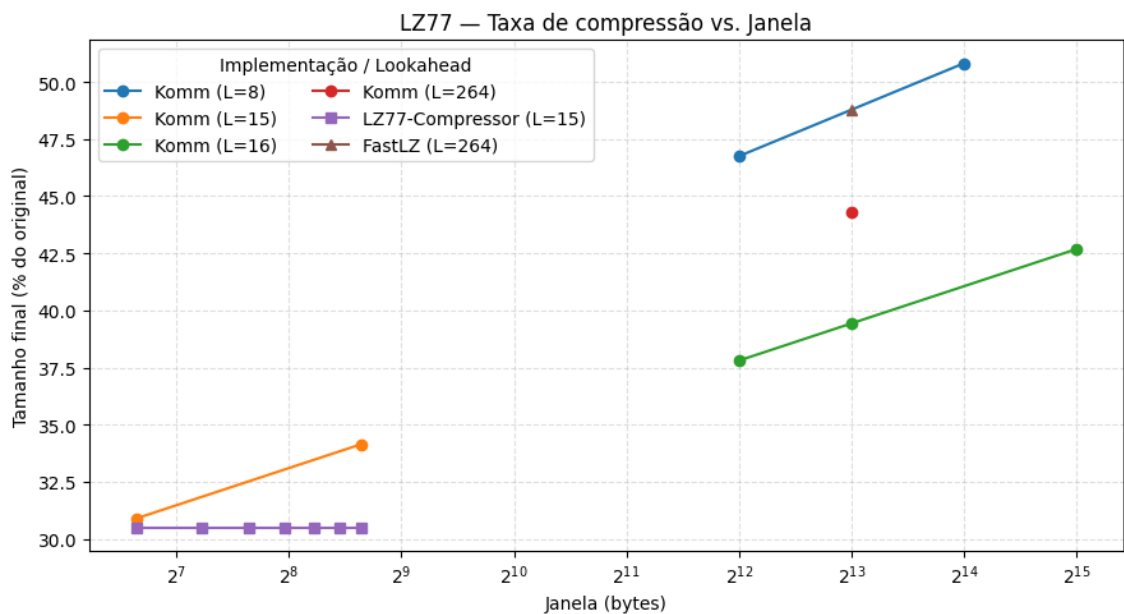
Análise (texto):

- A implementação *FastLZ* apresentou tempos de compressão inferiores, resultado coerente com sua implementação em C voltada à velocidade.

Figura 18 – Texto *Alice*: tempo de compressão.

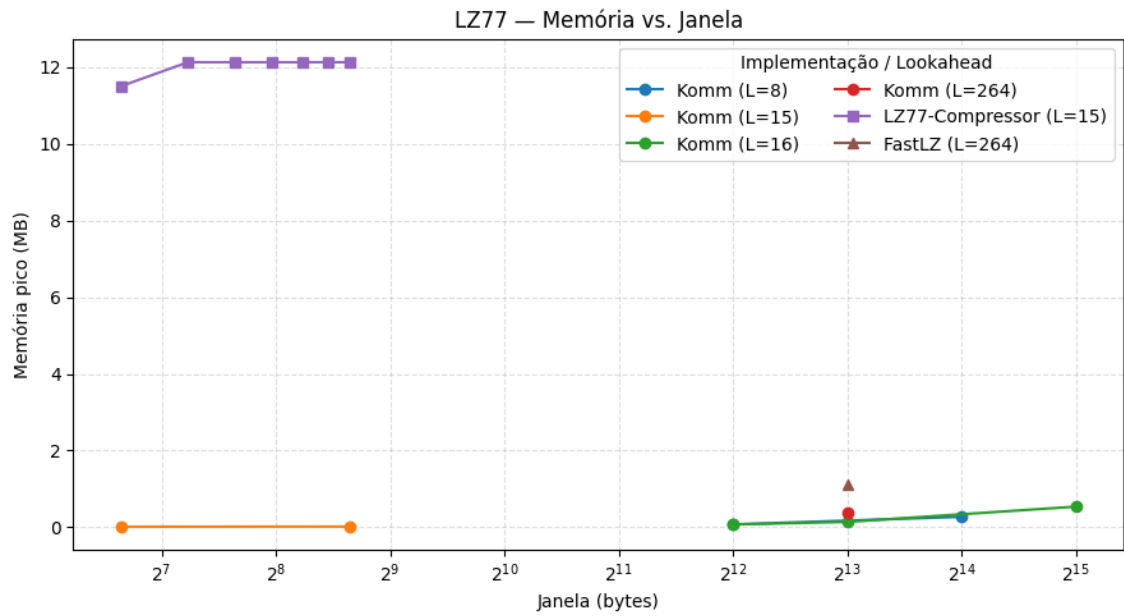
Fonte: Elaborada pelo autor.

- A versão da *Komm*, com $W = 64\text{ kB}$, obteve taxas de compressão próximas, mostrando boa eficiência mesmo em Python.
- A biblioteca *LZ77-Compressor* apresentou maior tempo de execução, mas consumo de memória reduzido.

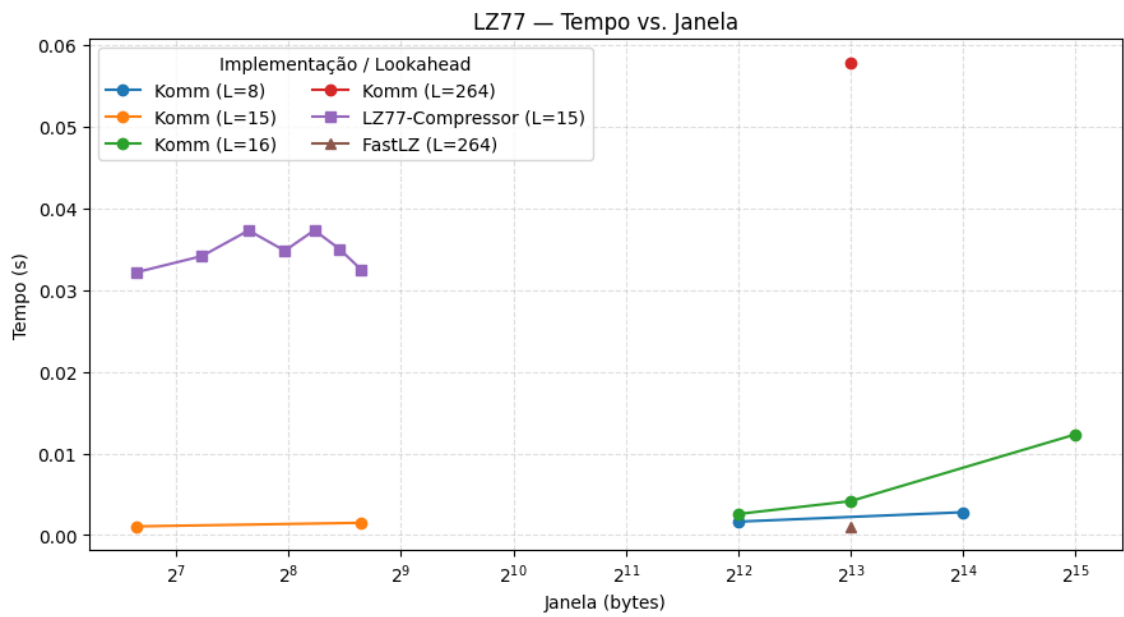
Figura 19 – Imagem *smiley*: taxa de compressão (menor é melhor).

Fonte: Elaborada pelo autor.

Análise (imagens):

Figura 20 – Imagem *smiley*: memória pico.

Fonte: Elaborada pelo autor.

Figura 21 – Imagem *smiley*: tempo de compressão.

Fonte: Elaborada pelo autor.

- A implementação da *Komm* superou as versões externas em taxa de compressão, principalmente para janelas maiores.
- O *FastLZ* manteve vantagem em tempo de execução, mas com menor eficiência de compressão.
- O *LZ77-Compressor* exibiu desempenho inferior em tempo e memória, refletindo limitações inerentes à implementação em Python puro.

4.5 DISCUSSÃO

Os resultados obtidos confirmam o comportamento clássico do *LZ77*: (i) janelas maiores aumentam a capacidade de reutilização de padrões, melhorando a taxa de compressão; (ii) esse ganho é acompanhado de aumento linear de tempo e memória; (iii) implementações em linguagens compiladas, como C, tendem a dominar em desempenho absoluto.

Apesar disso, a versão implementada na *Komm* mostrou-se competitiva, validando a adequação da arquitetura modular proposta e seu potencial como ferramenta educacional e de pesquisa. Todos os testes preservaram a integridade dos dados (*round trip* verdadeiro).

REFERÊNCIAS

- DEUTSCH, L. Peter. **DEFLATE Compressed Data Format Specification version 1.3**. RFC Editor, mai. 1996. 17 p. RFC 1951. (Request for Comments, 1951). DOI: 10.17487/RFC1951. Disponível em: <<https://www.rfc-editor.org/info/rfc1951>>.
- MACKAY, D.J.C. **Information Theory, Inference and Learning Algorithms**. Cambridge University Press, 2003. ISBN 9780521642989. Disponível em: <https://books.google.com.br/books?id=AKuMj4PN_EM>.
- NELSON, M. **The Data Compression Book**. BPB Publications, 2008. ISBN 9788170297291. Disponível em: <<https://books.google.com.br/books?id=pndwnAEACAAJ>>.
- SAYOOD, K. **Introduction to Data Compression**. Elsevier Science, 2012. (The Morgan Kaufmann Series in Multimedia Information and Systems). ISBN 9780124157965. Disponível em: <<https://books.google.com.br/books?id=mkCMxnHm6hsC>>.
- ZIV, J.; LEMPEL, A. A universal algorithm for sequential data compression. **IEEE Transactions on Information Theory**, v. 23, n. 3, p. 337–343, 1977. DOI: 10.1109/TIT.1977.1055714.

APÊNDICE A – MEU PRIMEIRO APÊNDICE

Texto ou documento, elaborado pelo autor, a fim de complementar sua argumentação, sem prejuízo da unidade nuclear do trabalho. Os apêndices são identificados por letras maiúsculas ordenadas alfabeticamente, travessão e pelo respectivo título.

ANEXO A – MEU PRIMEIRO ASSUNTO DE ANEXO

Texto ou documento não elaborado pelo autor, que serve de fundamentação, comprovação e ilustração. Os anexos são identificados por letras maiúsculas ordenadas alfabeticamente, travessões e pelos respectivos títulos.

ANEXO B – SEGUNDO ASSUNTO QUE PESQUISEI

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.