INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE SANTA CATARINA

CAMPUS SÃO JOSÉ

RHENZO HIDEKI SILVA KAJIKAWA

ESTUDO E IMPLEMENTAÇÃO DO ALGORITMO DE COMPRESSÃO LZ77 NA BIBLIOTECA KOMM

SÃO JOSÉ

2025

Rhenzo Hideki Silva Kajikawa

Estudo e Implementação do Algoritmo de Compressão LZ77 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 do Algoritmo de Compressão LZ77 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.

Documento assinado digitalmente
ROBERTO WANDERLEY DA NOBREGA
Data: 17/12/2025 15:01:54-0300
CPF: ***.486.259-**
Verifique as assinaturas em https://v.ufsc.br

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

Documento assinado digitalmente
DIEGO DA SILVA DE MEDEIROS
Data: 19/12/2025 09:37:38-0300
CPF: ***.242.729-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Diego da Silva de Medeiros, Dr.
Instituto Federal de Santa Catarina

Documento assinado digitalmente
ELEN MACEDO LOBATO
Data: 18/12/2025 14:47:35-0300
CPF: ***.804.772-**
Verifique as assinaturas em https://v.ufsc.br

Professora Elen Macedo Lobato, Dra.
Instituto Federal de Santa Catarina

Aos meus pais, que sempre me apoiaram na minha jornada.

AGRADECIMENTOS

Agradeço aos meus pais por me proporcionarem esta oportunidade e por sempre me apoiarem ao longo deste caminho. Mesmo nos momentos difíceis, estiveram ao meu lado, motivando-me a seguir em frente e lembrando-me de que nunca estarei sozinho.

Sou grato ao meu orientador, professor Roberto W. Nobrega, que me acompanhou durante o desenvolvimento deste trabalho. Seu conhecimento foi essencial para o meu aprendizado e crescimento acadêmico.

Agradeço também aos meus amigos, que sempre estiveram comigo e que considero uma segunda família. Eles fazem parte de quem eu sou e estiveram presentes tanto nos momentos tranquilos quanto nos desafiadores.

Por fim, sou grato a todos os professores, amigos e colegas que, de alguma forma, contribuíram para esta jornada.

"Não há fatos eternos, como não há verdades absolutas." (Friedrich Nietzsche)

RESUMO

A biblioteca de compressão de dados Komm foi estendida com a integração de um algoritmo sem perdas adicional: LZ77. 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 o estudo teórica dos algoritmos, projeto e implementação de módulos na arquitetura existente em Python, documentação e validação por meio de testes automatizados do 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. Esperou-se comprovar a viabilidade prática dos algoritmos e oferecer subsídios para futuras ampliações da biblioteca.

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

ABSTRACT

The Komm data compression library will be extended by integrating an additional lossless algorithm: LZ77. 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; Python.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

SUMÁRIO

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 estatística.

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 código 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 do algoritmo LZ77.

1.2 OBJETIVOS ESPECIFICOS

- Projetar e desenvolver módulos da codificação LZ77 na arquitetura da *Komm*, observando padrões de codificação, cobertura de testes e boa 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

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta um estudo dos conceitos teóricos da compressão sem perdas LZ77. O Capítulo 3 detalha a implementação do algoritmo na biblioteca *Komm*. O Capítulo 4 discute os resultados experimentais obtidos. E por fim, o Capítulo 5 conclui o trabalho e sugere direções para trabalhos futuros.

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 deslizante** de comprimento fixo W, que percorre sequencialmente a entrada e permite substituir repetições por referências a ocorrências anteriores.

2.1.1 Estrutura da janela deslizante

A janela deslizante 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 e L) são comprimentos fundamentais para o funcionamento do algoritmo. Na prática, é suficiente especificar apenas dois deles, pois o terceiro pode ser obtido diretamente pela relação acima. O parâmetro W determina o tamanho da janela total, S limita o histórico de busca e L define o comprimento máximo da sequência a ser codificada em um único passo.

Quando o *search buffer* não possui dados pré-carregados (isto é, no início da codificação), o algoritmo considera as posições anteriores como preenchidas com símbolos especiais, por exemplo zeros, representando a ausência de histórico real.

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, com $0 \leq p < S$;
- ℓ (*length*) é o comprimento da sequência coincidente, com $0 \leq \ell < L$;
- $x \in \mathcal{X}$ é o próximo símbolo literal, que segue a sequência copiada e garante a continuidade da decodificação.

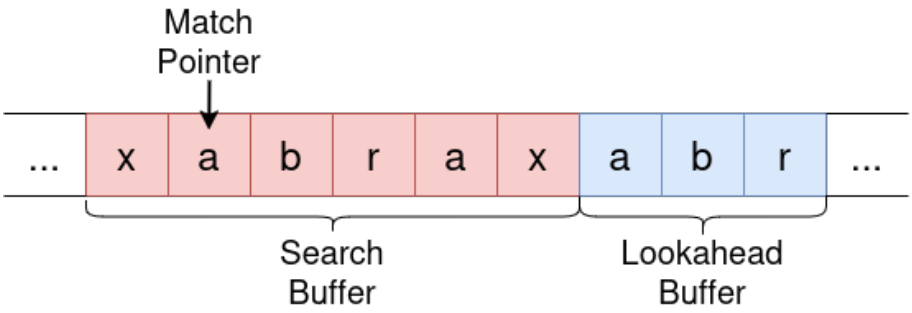
A saída do codificador é, portanto, uma sequência de tokens $\langle p_i, \ell_i, x_i \rangle$, que representam os fatores sucessivos do texto de entrada.

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 Exemplo de Codificação e Decodificação

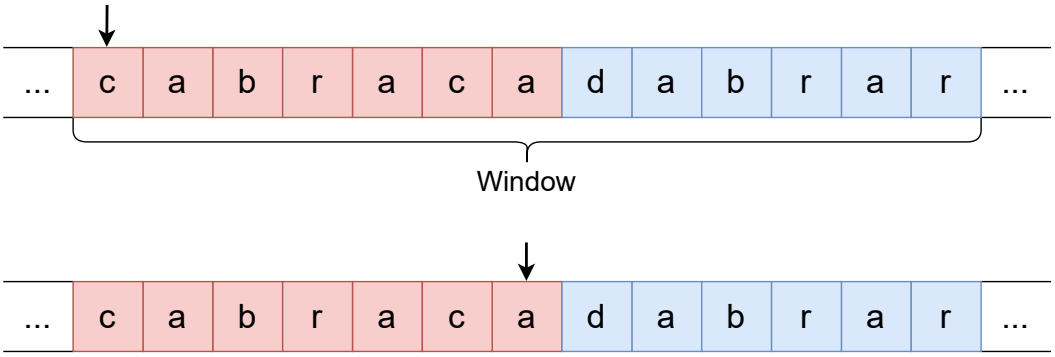
Nesta seção será demonstrado um exemplo detalhado do funcionamento da janela deslizante e da representação do token $\langle p, \ell, x \rangle$ utilizados pelo algoritmo LZ77.

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.

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.

A sequência a ser codificada é `"cabracadabrarrarrad"` (SAYOOD, 2012). 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

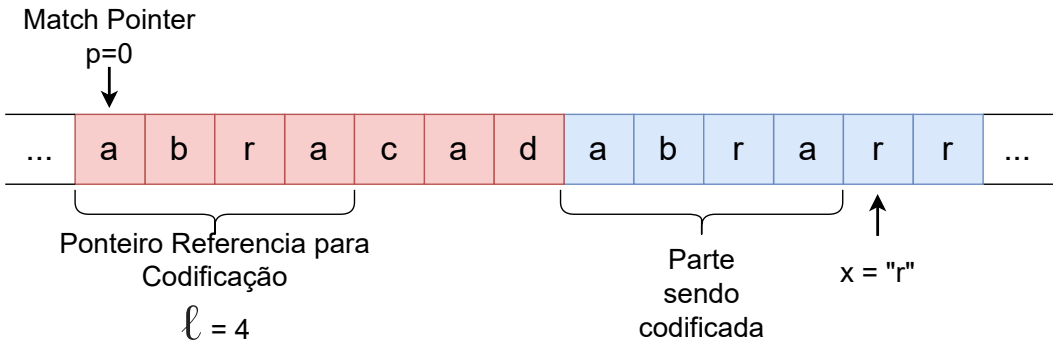


Fonte: Adaptada de Sayood (2012)

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 $\langle 6, 0, \texttt{d} \rangle$, 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.

Figura 3 – Estado da janela na segunda codificação no algoritmo LZ77

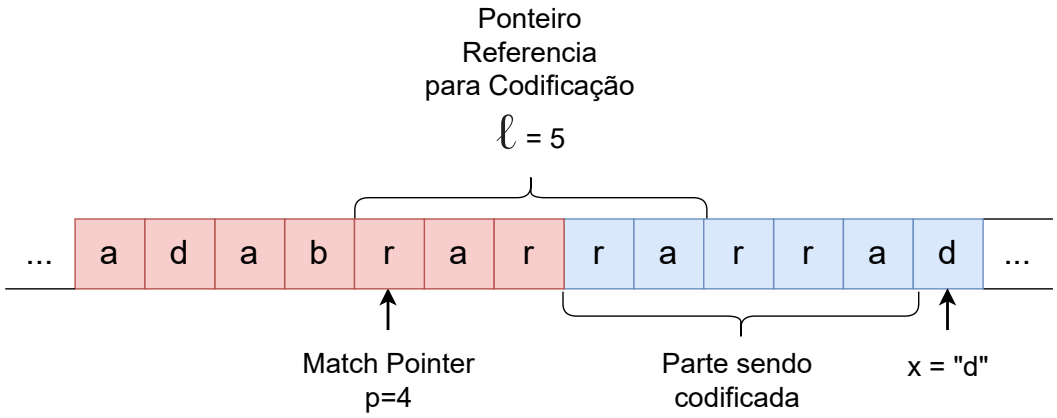


Fonte: Adaptada de Sayood (2012)

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 na posição inicial da janela. Essa correspondência possui comprimento 4 caracteres.

Dessa forma, o algoritmo codifica a sequência encontrada como o token $\langle 0, 4, \texttt{r} \rangle$, onde 0 indica a distância 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).

Figura 4 – Estado da janela na terceira codificação no algoritmo LZ77



Fonte: Adaptada de Sayood (2012)

Após a segunda codificação, a janela deslizante avança novamente, e o processo se repete. Procura-se no *search buffer* a maior correspondência para o início do *lookahead buffer*. Neste caso, a o deslocamento até a correspondência mais longa é 4, com comprimento 5 (`"rarra"`), sendo possível Figura 4 observar uma sobreposição que ocorre apartir

da codificação da sequência encontrada, seguido do caractere literal `"d"`. Assim, o token gerado é $\langle 4, 5, \texttt{d} \rangle$, este processo pode ser observado na Figura 4.

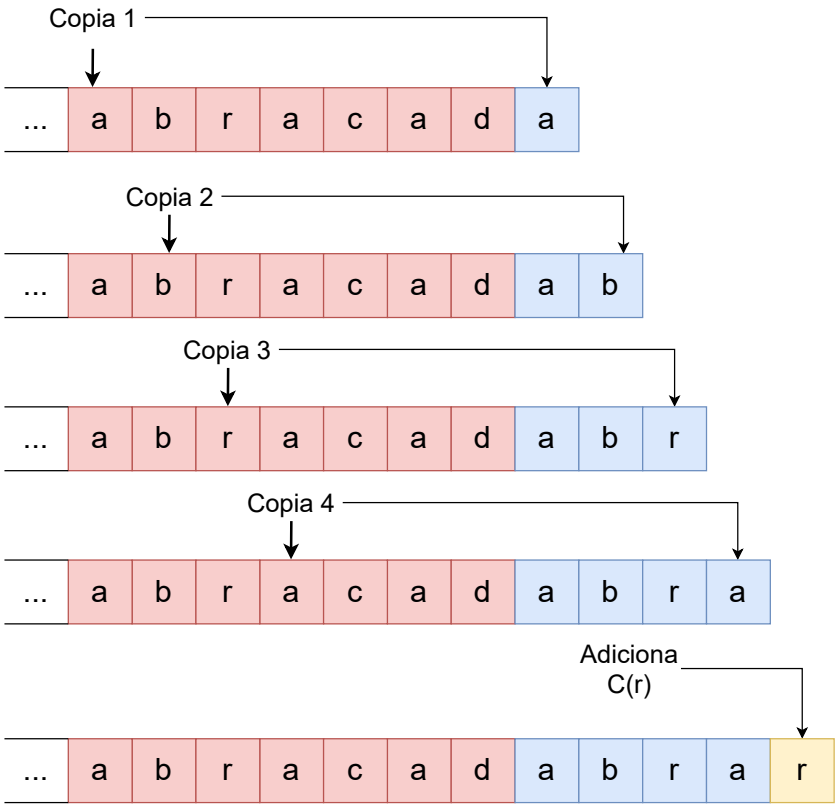
Para realizar o processo inverso, ou seja, decodificar o token recebido $\langle 0, 4, \texttt{r} \rangle$, o decodificador utiliza o mesmo princípio do algoritmo LZ77, porém no sentido inverso.

Inicialmente, ele utiliza o pointer (p) para reconstruir, neste caso na posição 0 do *search buffer*, a 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 (x), que neste exemplo é `"r"`.

Esse processo é ilustrado passo a passo na Figura 5. 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 5 – Decodificação do exemplo $\langle 0, 4, \texttt{r} \rangle$



Fonte: Adaptada de Sayood (2012)

Utilizando o exemplo anterior do token $\langle 0, 4, \mathtt{r} \rangle$, 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 5, logo exige também 3 bits;
- O caractere literal (x) 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 ($\langle 0, 4, \mathtt{r} \rangle$), obtêm-se:

- O valor do *pointer* $p = 0$, 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* voltada para o estudo e a simulação de sistemas de comunicação, desenvolvida pelo professor Roberto Nóbrega. Trata-se de um projeto *open-source* compatível com *Python 3*, que fornece um conjunto de ferramentas voltadas tanto para a análise teórica quanto para a implementação prática de técnicas clássicas e modernas de comunicação digital.

A biblioteca adota uma filosofia fortemente inspirada em ferramentas acadêmicas e industriais, como o *MATLAB® Communications System Toolbox™*, o *GNU Radio*, o *CommPy* e o *SageMath*. Contudo, diferencia-se dessas alternativas por priorizar uma abordagem didática e modular, em que cada componente do sistema é modelado de forma independente e parametrizável. Dessa maneira, a *Komm* se consolida como um ambiente unificado para ensino e pesquisa em comunicação, codificação de fontes e teoria da informação.

2.2.1 Estrutura e organização

A arquitetura da *Komm* é organizada em módulos temáticos que cobrem diferentes aspectos dos sistemas de comunicação, incluindo: modulação, codificação de canal, correção de erros, codificação de fonte e geração de sinais. Cada módulo é projetado para ser independente, porém interoperável com os demais, de modo que o usuário possa compor sistemas completos de comunicação digital a partir de blocos básicos.

No contexto deste trabalho, o foco está no módulo de **codificação sem perdas** (*lossless coding*), responsável pela implementação de algoritmos de compressão de dados baseados em dicionário e em códigos de comprimento variável. Esse módulo disponibiliza atualmente os seguintes esquemas de codificação:

- **Shannon Code**: implementação baseada no princípio de entropia mínima para codificação por símbolos individuais;
- **Fano Code**: variante do método de Shannon–Fano com particionamento recursivo das probabilidades;
- **Tunstall Code**: esquema de comprimento fixo sobre alfabetos de saída maiores, utilizado em compressão de texto;
- **Huffman Code**: código ótimo de comprimento variável, usado como referência clássica em compressão sem perdas;
- **Lempel–Ziv 78 (LZ78)**: algoritmo incremental baseado em dicionário dinâmico;
- **Lempel–Ziv–Welch (LZW)**: extensão do LZ78 que remove o envio explícito de caracteres literais.

2.2.2 Motivação educacional e integração

A *Komm* foi concebida não apenas como uma biblioteca de software, mas também como uma ferramenta de apoio ao ensino e à experimentação. Sua estrutura modular permite que estudantes explorem separadamente as etapas de codificação, decodificação e análise de desempenho, favorecendo a compreensão dos princípios de cada algoritmo.

Cada classe segue um padrão uniforme de interface, documentação e testes, permitindo o uso consistente de diferentes algoritmos com a mesma estrutura de chamada. Isso viabiliza comparações diretas entre métodos de compressão, tanto em termos de eficiência quanto de complexidade computacional, o que é particularmente útil em contextos acadêmicos.

2.2.3 Relação com este trabalho

A implementação do algoritmo *Lempel–Ziv 77* (LZ77) desenvolvida neste trabalho foi incorporada ao módulo de *lossless coding* da biblioteca *Komm*. Essa contribuição amplia o conjunto de técnicas de compressão disponíveis, ampliando a família de algoritmos *Lempel–Ziv*.

Dessa forma, o presente trabalho se insere diretamente no propósito pedagógico da *Komm*: disponibilizar implementações abertas, documentadas e consistente em relação aos algoritmos clássicos de codificação de fonte, permitindo ao mesmo tempo estudo teórico, análise experimental e integração com outros componentes de sistemas de comunicação digital.

3 DESENVOLVIMENTO

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 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.

3.2 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.

Código 3.1 – Exemplo de conversão intermediária entre sequência e tokens.

```
1 import komm
2
3 # Instanciação do codificador
4 lz77 = komm.LempelZiv77Code(
5     window_size=13,
```

[1] https://komm.dev/ref/LempelZiv77Code

```
    lookahead_size=6,
    source_cardinality=256,
    target_cardinality=2,
    search_buffer=[ord(x) for x in "cabraca"]
)

# Exemplo de sequência de entrada
source = [ord(x) for x in "dabrarrarrad"]


tokens = lz77.source_to_tokens(source)
print(tokens)
# [(6, 0, 100), (0, 4, 114), (4, 5, 100)]

reconstructed = lz77.tokens_to_source(tokens)
assert reconstructed == source
```

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

Código 3.2 – Codificação e Decodificação direta.

```
# Codificação e decodificação
encoded = lz77.encode(source)
print(encoded.reshape(-1,14)[1:2])
#[[0 0 0 1 0 0 0 1 1 1 0 0 1 0]]


decoded = lz77.decode(encoded)
np.array_equal(source, decoded)
# True
```

O método `encode()` retorna o fluxo comprimido no alfabeto de saída $\mathcal{Y}$, enquanto `decode()` reconstrói a sequência original no alfabeto de entrada $\mathcal{X}$, sendo possivel observar o `print()` do `encoded()` teve como mesmo resultado o array dado no exemplo final da Seção 2.1.3 quando mostrado os bits codificados.

3.3 COMO FOI FEITO

A arquitetura do módulo segue o padrão das demais classes de codificação da biblioteca. A classe foi estruturada em quatro métodos 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 (p, ℓ, x);
- `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.

3.4 PROCESSO DE BUSCA NA CODIFICAÇÃO (`SOURCE_TO_TOKENS`)

O núcleo da codificação LZ77 está concentrado no método `source_to_tokens()`, responsável por identificar, dentro do *search buffer*, o maior trecho que coincide com o prefixo do *lookahead buffer*. Nesta implementação, a busca é realizada de forma simples e eficiente por meio do método `rfind()` da classe `bytes` do Python, o que resulta em um código conciso e um desempenho aceitável, mesmo sem recorrer a estruturas auxiliares complexas, como tabelas de dispersão (*hash tables*).

A cada iteração, o algoritmo:

1. define o *search buffer* como o trecho de tamanho $S = W - L$;
2. procura, dentro desse trecho, o maior sufixo que coincide com o prefixo atual do *lookahead buffer*;
3. emite o token $\langle p, \ell, x \rangle$, em que 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 original de Ziv e Lempel (ZIV; LEMPEL, 1977), medindo o ponteiro p a partir do início do *search buffer*, em contraste com algumas implementações modernas que utilizam o deslocamento a partir do final do buffer. A utilização do `rfind()` preserva a clareza e a corretude do algoritmo, permitindo a reprodução fiel dos resultados conceituais do LZ77, conforme apresentado no artigo de 1977.

No entanto, é importante observar que o uso de `rfind()` representa uma busca direta por comparação de cadeias, sem otimizações adicionais de indexação. Já implementações projetadas para alto desempenho, como o *FastLZ*, o uso de tabelas de *hash* permite localizar rapidamente posições candidatas a correspondências, reduzindo drasticamente a quantidade de comparações necessárias.

Assim, a implementação proposta na *Komm* prioriza a legibilidade e a adesão ao modelo teórico clássico, enquanto reconhece que técnicas como o uso de *hash tables* podem ter melhor desempenho sem alterar o comportamento lógico do algoritmo.

3.5 TESTES E VALIDAÇÃO

Os testes unitários[2] 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*.

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.

A seguir, apresenta-se um dos testes unitários mais representativos, baseado no exemplo do livro de Abrantes (ABRANTES, s.d.), que verifica a correspondência entre a sequência original e o conjunto de tokens gerados:

Código 3.3 – Teste unitário baseado no exemplo do Abrantes (p. 16).

```python
def test_lz77_abrantes():
    # [Abrantes, p. 16]
    code = komm.LempelZiv77Code(
        window_size=12,
        lookahead_size=4,
        source_cardinality=3,
        search_buffer=[255] * 8,
    )
    alphabet = "ABC"
    source = [alphabet.index(x) for x in "AAAABABCCAABACCAAAABC"]
    expected = [
        (1, 0, "A"),
        (1, 3, "B"),
        (2, 2, "C"),
        (1, 1, "A"),
        (7, 3, "C"),
        (6, 3, "A"),
        (8, 2, "C"),
    ]
    tokens = []
```

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

```
21      for offset, length, symbol in expected:
22          tokens.append((8 - offset, length, alphabet.index(symbol)))
23      np.testing.assert_equal(code.source_to_tokens(source), tokens)
24      np.testing.assert_equal(code.tokens_to_source(tokens), source)
25      np.testing.assert_equal(code.decode(code.encode(source)), source)
```

Esse teste reproduz o exemplo de codificação da sequência `"AAAABABCCAABACCAAAABC"`, utilizando o mesmo tamanho de janela e *lookahead* definidos por Abrantes. No entanto, algumas adaptações foram necessárias devido a uma diferença fundamental entre as abordagens: enquanto o modelo descrito por Abrantes considera o deslocamento do ponteiro p a partir do fim da janela (leitura da direita para a esquerda), a implementação da biblioteca *Komm* adota a convenção clássica de Ziv e Lempel, na qual o ponteiro é medido a partir do início do *search buffer* (leitura da esquerda para a direita).

Para compatibilizar os resultados e garantir a equivalência entre as duas notações, foi necessário ajustar os valores de deslocamento produzidos pelo exemplo original. Esse ajuste é realizado no trecho:

Código 3.4 – Adaptação dos deslocamentos dos tokens.

```
1       for offset, length, symbol in expected:
2           tokens.append((8 - offset, length, alphabet.index(symbol)))
```

Nessa etapa, o valor `(8-offset)`, onde 8 é o tamanho do *search buffer*, converte a convenção utilizada por Abrantes para a convenção adotada pela *Komm*, preservando a correspondência semântica dos tokens. Esse mapeamento garante que cada token $\langle p, \ell, x \rangle$ represente exatamente o mesmo *match* no texto, apesar da diferença na direção da indexação. Assim pode-se verificar as três propriedades centrais do algoritmo:

1. a geração correta dos tokens $\langle p, \ell, x \rangle$;
2. a reconstrução exata da sequência original a partir dos tokens;
3. a preservação da integridade na operação composta `decode(encode(x)) = x`.

A validação automática por meio dos testes unitários demonstra que a implementação segue fielmente a definição formal do LZ77 e mantém compatibilidade total com o padrão de codificação utilizado na *Komm*.

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

Foram utilizados dois tipos de arquivos:

- **Textos**: foram selecionados dois arquivos distintos:
  - o livro *Alice's Adventures in Wonderland*, do Projeto Gutenberg[1], por se tratar de um corpus literário clássico amplamente utilizado em experimentos de compressão sem perdas. O arquivo possui 154,573 bytes e apresenta ampla variedade de caracteres e repetições, sendo adequado para testar diferentes tamanhos de janela no *LZ77*;
  - um arquivo de *log*[2] de sistema, com 214,486 bytes, escolhido por conter diversas linhas repetitivas e padrões recorrentes, características que o tornam um bom candidato para a avaliação do *LZ77*.
- **Imagens**: foram utilizadas imagens no formato bitmap (**BMP**), selecionadas por apresentarem regiões de baixa entropia e padrões espaciais redundantes, permitindo observar o comportamento do algoritmo em fontes bidimensionais. As imagens utilizadas foram:
  - *smiley*, com 246 bytes (Figura 6);
  - *snail*, com 196,666 bytes (Figura 7).

[1] https://www.gutenberg.org/ebooks/11

[2] https://raw.githubusercontent.com/SoftManiaTech/sample_log_files/refs/heads/master/Linux/Linux_2k.log

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

Fonte: https://cse1.net/recaps/graphics.

Figura 7 – Imagem bitmap (BMP) *snail*.

Fonte: https://people.math.sc.edu/Burkardt/data/bmp/bmp.html.

4.2 MÉTRICAS DE AVALIAÇÃO

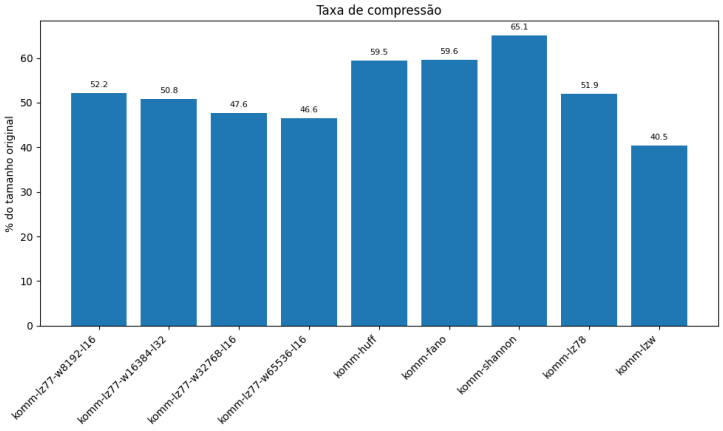
As seguintes métricas foram utilizadas para avaliar o desempenho dos algoritmos:

- **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`.

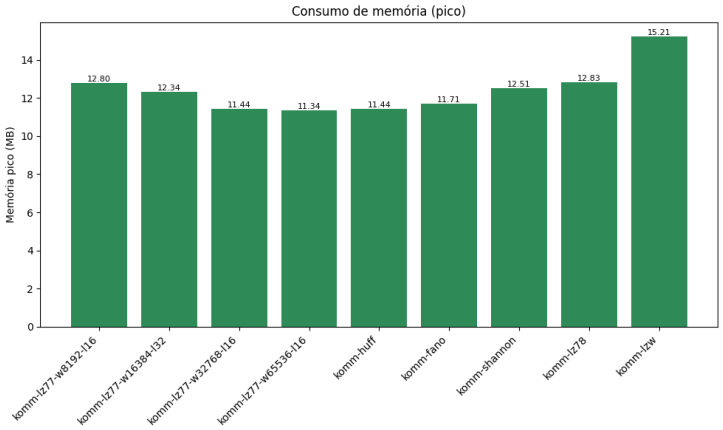
A medição de memória foi realizada com o auxílio das bibliotecas `psutil` e `tracemalloc`, permitindo registrar a memória pico durante os processos de compressão e descompressão.

4.3 RESULTADOS DO LZ77 NA *KOMM*

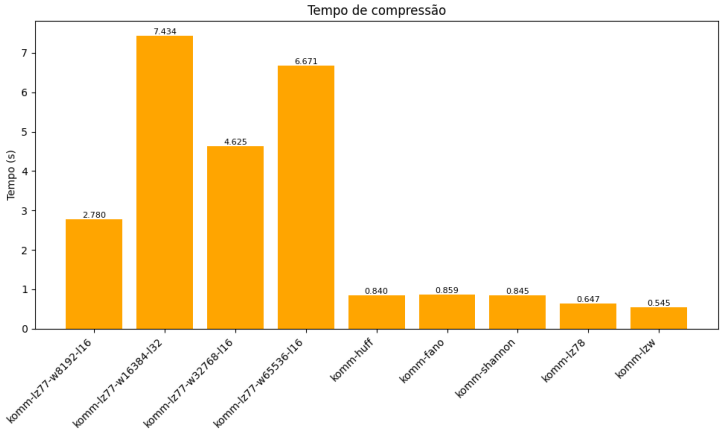
Para avaliar o desempenho do *LZ77* na *Komm*, foram realizados testes variando o tamanho da janela W, assim como o tamanho do *lookahead* L. Os resultados foram comparados com os algoritmos de compressão Huffman, Shannon–Fano, LZ78 e LZW disponíveis na biblioteca.



(a) Taxa de compressão.



(b) Memória pico.

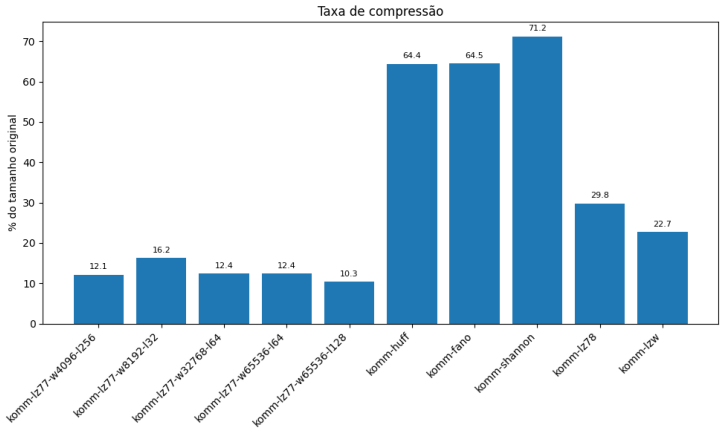


(c) Tempo de compressão.

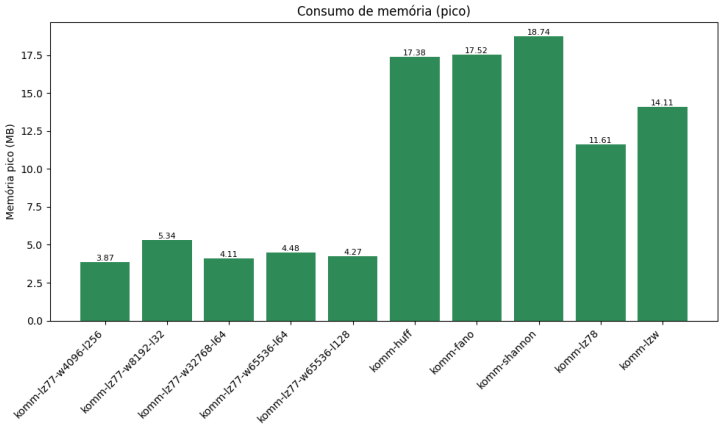
Figura 8 – Resultados da compressão do texto *Alice* com o *LZ77* da *Komm*.

Fonte: Elaborada pelo autor.

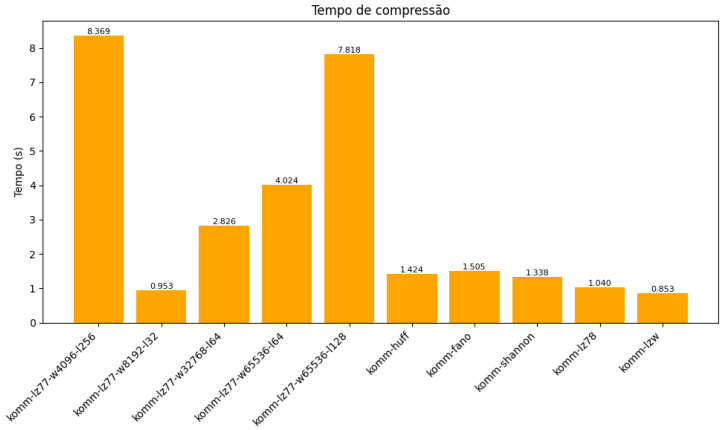
Para compressão do texto *Alice*, os resultados podem ser observados na Figura 8. Apesar dos resultados de compressão serem, na maioria dos casos, um pouco melhores que os dos outros algoritmos e o pico de memória ser equivalente, o tempo de compressão do *LZ77* é significativamente maior. Para finalidades educacionais, contudo, esse custo adicional pode não ser um problema tão grande.



(a) Taxa de compressão.



(b) Memória pico.

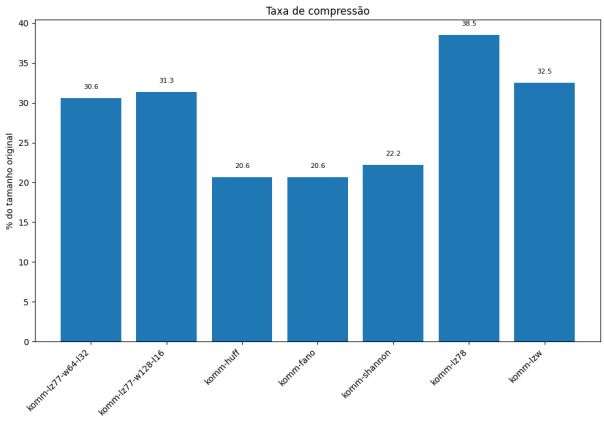


(c) Tempo de compressão.

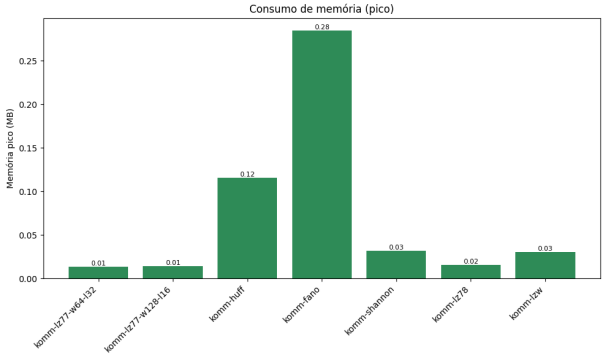
Figura 9 – Resultados da compressão do texto *log* com o *LZ77* da *Komm*.

Fonte: Elaborada pelo autor.

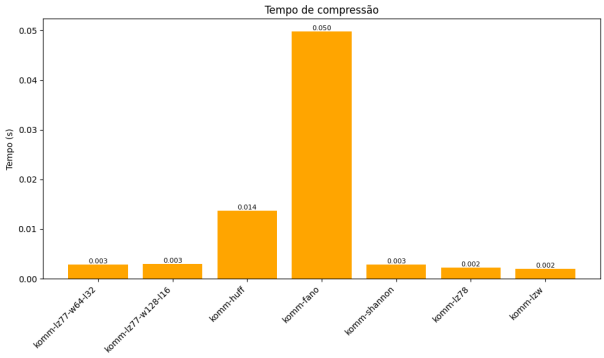
Para compressão do texto de *log*, os resultados podem ser observados na Figura 9. Aqui o algoritmo *LZ77* se saiu melhor que os outros algoritmos em termos de taxa de compressão e uso de memória; contudo, o tempo de compressão continuou sendo significativamente maior em algumas configurações, principalmente naquelas que utilizam um *lookahead* maior.



(a) Taxa de compressão.



(b) Memória pico.



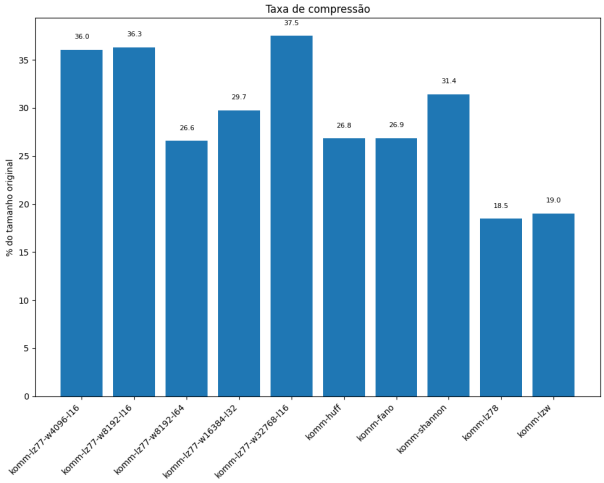
(c) Tempo de compressão.

Figura 10 – Resultados da compressão da imagem *smiley* com o *LZ77* da *Komm*.

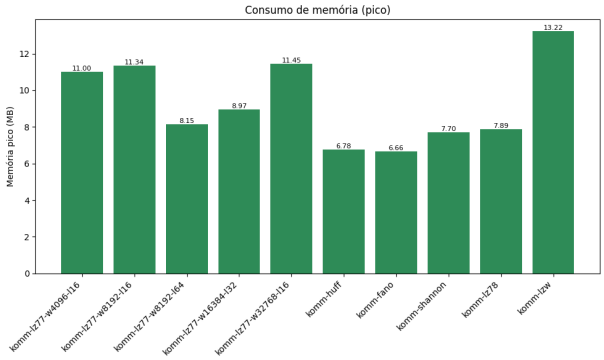
Fonte: Elaborada pelo autor.

Para compressão da imagem *smiley*, os resultados podem ser observados na Figura 10. Neste caso, é possível observar que o *LZ77* não se saiu tão bem quanto nos textos. Além disso, há variação entre as diferentes configurações de janela, mas é possível

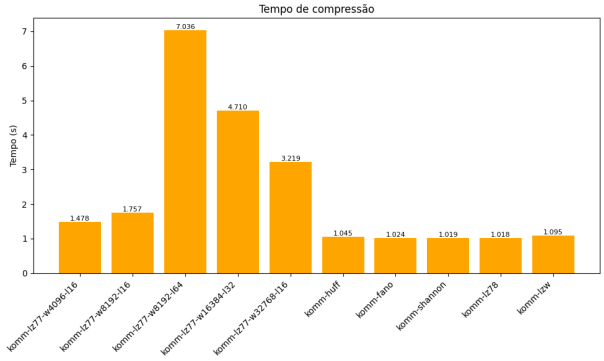
notar que a compressão do arquivo não se beneficiou de janelas maiores. Apesar disso, o uso de memória se manteve relativamente constante e o tempo de compressão se manteve competitivo.



(a) Taxa de compressão.



(b) Memória pico.



(c) Tempo de compressão.

Figura 11 – Resultados da compressão da imagem *snail* com o *LZ77* da *Komm*.

Fonte: Elaborada pelo autor.

Para compressão da imagem *snail*, os resultados podem ser observados na Figura 11. Neste caso, é possível observar que o *LZ77* também não se saiu tão bem quanto nos textos. As taxas de compressão, assim como o uso de memória, se mantiveram rela-

tivamente constantes, porém o tempo de compressão aumentou consideravelmente para janelas com *lookahead* maior.

Dessa forma, é possível observar que o *LZ77* se beneficia de arquivos com padrões lineares e repetições frequentes, como textos, mas apresenta desempenho inferior em imagens com padrões bidimensionais complexos.

4.4 COMPARAÇÃO COM IMPLEMENTAÇÕES EXTERNAS DE LZ77

Foram consideradas três implementações populares do algoritmo *LZ77*: **FastLZ**[3] (em C), **LZ77-Compressor**[4] (em Python) e **lz77enco**[5] (no Octave) . As configurações padrão dessas bibliotecas são:

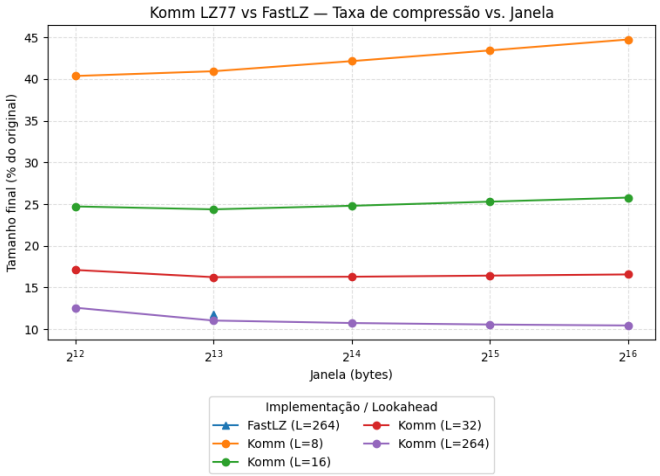
- **FastLZ**: janela de 8 kB e *lookahead* de 264 bytes;
- **LZ77-Compressor**: janela variável (100–400 bytes) e *lookahead* fixo de 15 bytes;
- **Octave**: janela variável, similar à da *Komm*.

Na *Komm*, foi possível parametrizar W e L livremente, permitindo a replicação aproximada das condições dessas bibliotecas.

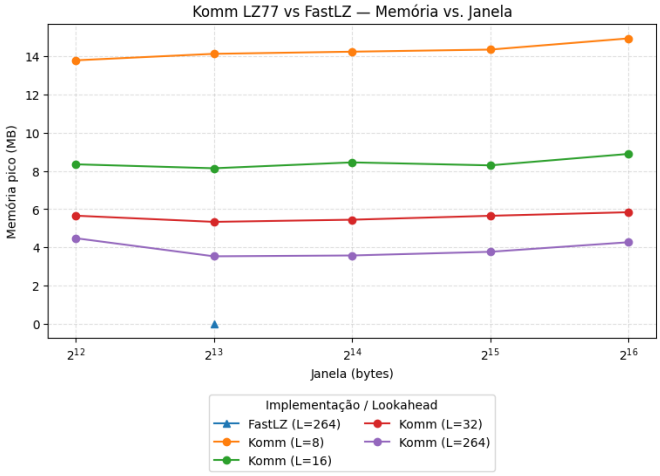
4.4.1 Comparando Komm com FastLZ

Nesta seção são apresentados os resultados comparativos entre a implementação do *LZ77* na *Komm* e a biblioteca *FastLZ* em C, mostrando apenas os resultados dos testes com as configurações mais próximas possíveis entre as duas implementações. Além disso, são apresentados apenas os resultados dos testes com o texto *log* e a imagem *snail*, por serem os mais representativos.

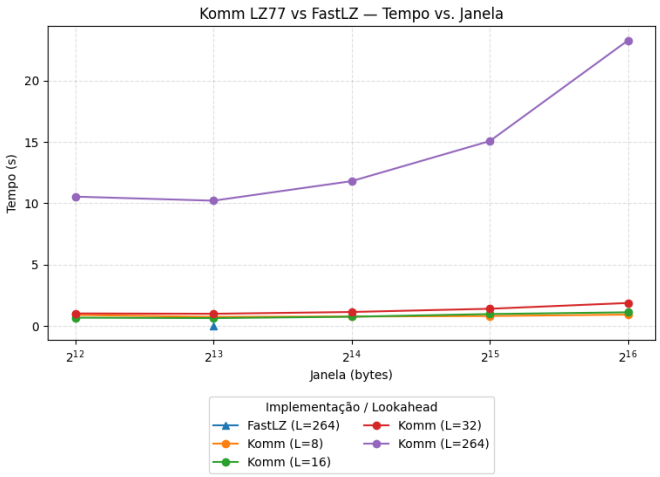
[3] https://github.com/ariya/FastLZ
[4] https://github.com/manassra/LZ77-Compressor
[5] https://octave.sourceforge.io/communications/function/lz77enco.html



(a) Taxa de compressão.



(b) Memória pico.



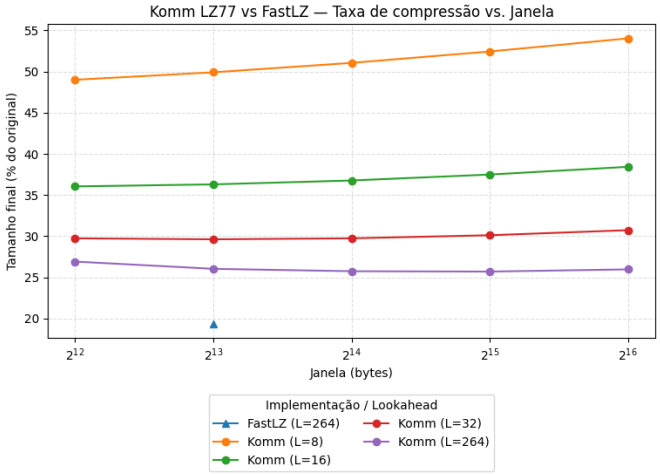
(c) Tempo de compressão.

Figura 12 – Resultados da compressão do texto *log* com a *Komm* e a *FastLZ*.

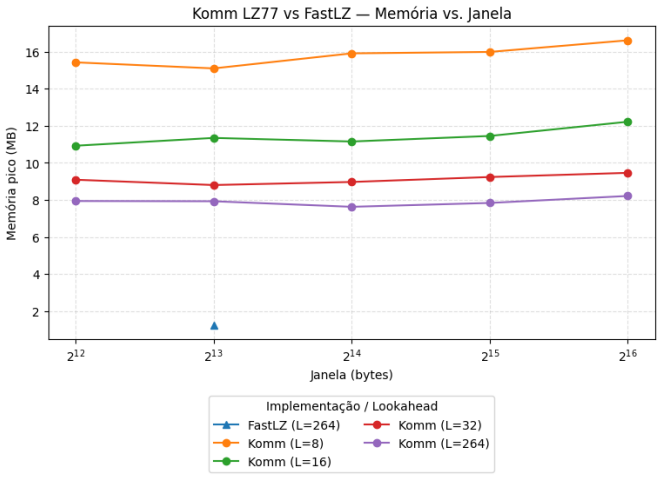
Fonte: Elaborada pelo autor.

É possível observar que a implementação do *FastLZ* em C apresentou desempenho superior em todas as métricas avaliadas, refletindo as vantagens de uma implementação

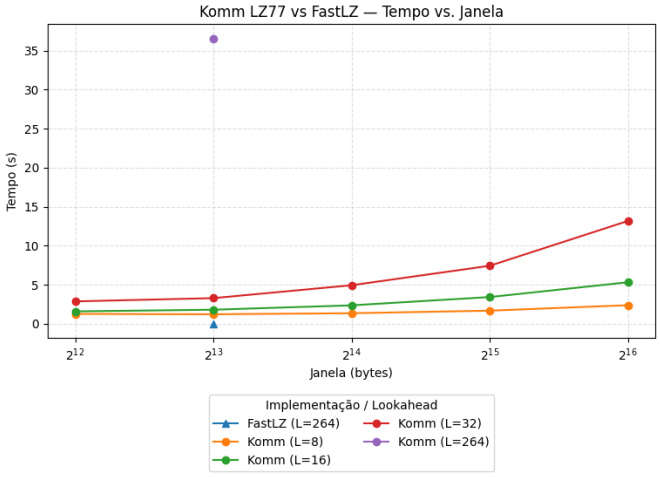
em linguagem compilada. A versão da *Komm*, embora competitiva em termos de taxa de compressão, teve tempos de execução significativamente maiores, especialmente com *lookahead* maiores.



(a) Taxa de compressão.



(b) Memória pico.



(c) Tempo de compressão.

Figura 13 – Resultados da compressão da imagem *snail* com a *Komm* e a *FastLZ*.

Fonte: Elaborada pelo autor.

Na compressão da imagem *snail*, os resultados seguem a mesma tendência observada no texto *log*. A implementação do *FastLZ* superou a *Komm* em todas as métricas, com destaque para o tempo de compressão, que foi ordens de grandeza menor. A taxa de compressão também foi melhor, possivelmente devido ao uso de tabelas de *hash* para busca de padrões.

Embora os experimentos tenham utilizado parâmetros equivalentes, a implementação em C (*FastLZ*) apresentou melhor taxa de compressão e tempo de execução significativamente menor. Essa diferença é explicada não apenas pela eficiência da linguagem compilada, mas também por decisões de projeto distintas em cada implementação.

A versão da *Komm*, escrita em Python, segue o modelo teórico original de Lempel e Ziv (ZIV; LEMPEL, 1977), utilizando tokens de formato fixo $\langle p, \ell, x \rangle$ e o método `rfind()` para localizar correspondências no *search buffer*. Esse método, embora eficiente por ser implementado em C dentro do interpretador Python, realiza uma busca sequencial de sufixos, sem o uso de estruturas auxiliares como tabelas de dispersão (*hash tables*). Tal escolha privilegia a clareza e a correção do algoritmo, mas implica maior custo computacional e gasto extra de bits na representação dos tokens.

Por outro lado, o *FastLZ* foi concebido com foco em desempenho: implementa um mecanismo de busca baseado em *hash tables* para localizar rapidamente possíveis correspondências, e emprega uma codificação de comprimento variável, reduzindo o tamanho dos metadados armazenados.

Em síntese, a diferença observada decorre de dois fatores principais:

1. **Modelo de codificação:** a versão da *Komm* procura seguir de forma fiel o esquema clássico do LZ77, utilizando tokens de tamanho fixo e separação explícita dos campos p, ℓ e x. Já o *FastLZ* não implementa exatamente o LZ77 original, mas uma variante da mesma família, com um formato de código mais compacto e voltado à prática. Nessa abordagem, literais e *matches* são agrupados em blocos e codificados com campos de comprimento variável, reduzindo a representação dos tokens e, em muitos casos, resultando em taxas de compressão superiores.

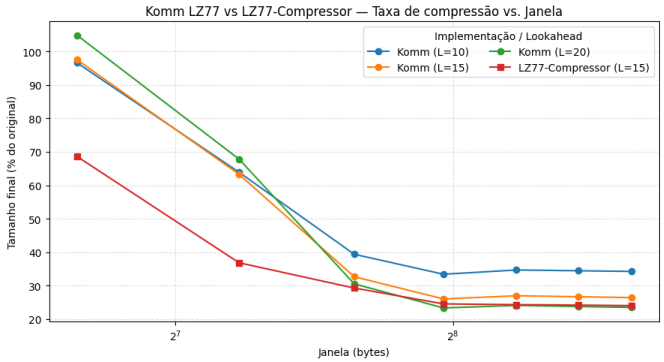
2. **Estratégia de busca:** na implementação da *Komm*, o uso de `rfind()` em Python realiza comparações diretas de cadeias no *search buffer*, sem estruturas auxiliares de indexação. O *FastLZ*, por sua vez, utiliza tabelas de *hash* para localizar rapidamente posições candidatas a correspondências, o que reduz o número de comparações necessárias e contribui tanto para o ganho de desempenho quanto para uma utilização mais eficiente da janela.

Dessa forma, embora ambas as abordagens pertençam à família de algoritmos inspirados em Lempel e Ziv, o *FastLZ* incorpora escolhas de projeto voltadas à eficiência

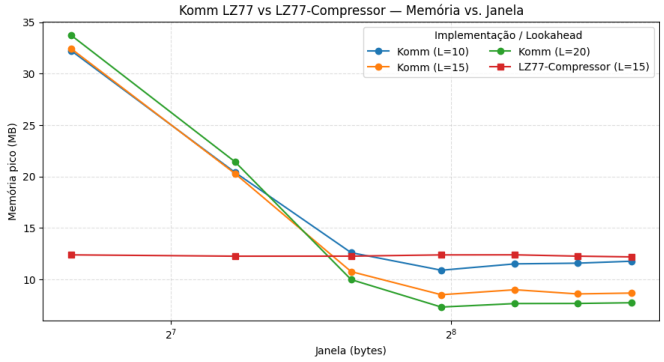
prática (formato de código compacto e busca acelerada), enquanto a versão da *Komm* privilegia a aderência ao modelo teórico e a transparência do funcionamento interno do LZ77.

4.4.2 Comparando Komm com LZ77-Compressor

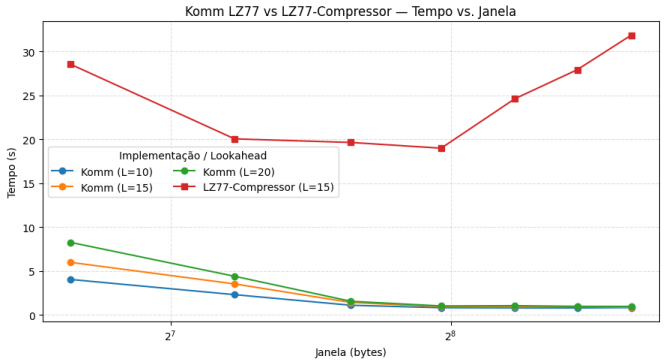
Nesta seção são apresentados os resultados comparativos entre a implementação do *LZ77* na *Komm* e a biblioteca *LZ77-Compressor* em Python, mostrando apenas os resultados dos testes com as configurações mais próximas possíveis entre as duas implementações. Como na seção anterior, são apresentados apenas os resultados dos testes com os textos *log* e a imagem *snail*, por serem os mais representativos.



(a) Taxa de compressão.



(b) Memória pico.

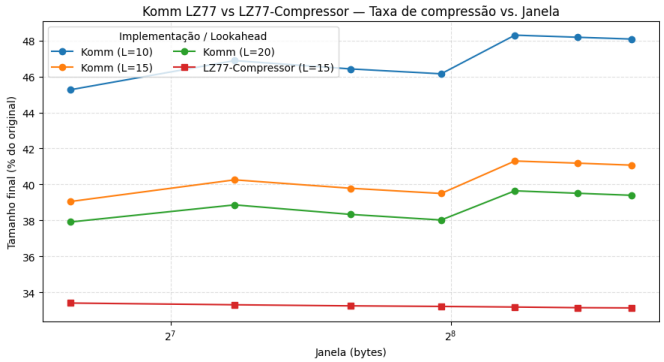


(c) Tempo de compressão.

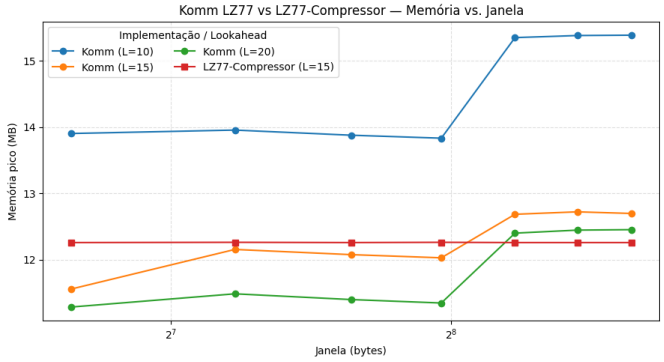
Figura 14 – Resultados da compressão do texto *log* com a *Komm* e a *LZ77-Compressor*.

Fonte: Elaborada pelo autor.

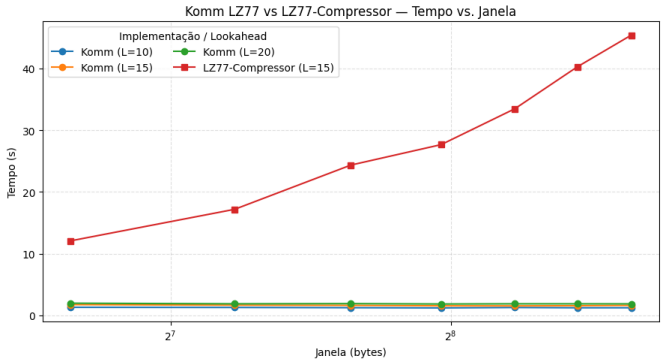
É possível observar que a implementação do *LZ77-Compressor* em Python apresentou desempenho contínuo em termos de taxa de compressão, embora com tempos de execução elevados, especialmente em janelas maiores. A versão da *Komm* mostrou-se competitiva em taxa de compressão em janelas maiores, mas com tempos de execução menores, refletindo as diferenças na implementação e otimização entre as duas bibliotecas.



(a) Taxa de compressão.



(b) Memória pico.



(c) Tempo de compressão.

Figura 15 – Resultados da compressão da imagem *snail* com a *Komm* e a *LZ77-Compressor*.

Fonte: Elaborada pelo autor.

Na compressão da imagem *snail*, os resultados obtidos pela *LZ77-Compressor* foram melhores do que os obtidos em relação à biblioteca *Komm*, em termos de taxa de

compressão. Contudo, o tempo de execução permaneceu elevado, refletindo as limitações da implementação.

É possível observar que a implementação do *LZ77-Compressor* em Python apresentou desempenho estável em termos de taxa de compressão, embora com tempos de execução elevados, especialmente para janelas maiores. A versão da *Komm* mostrou-se competitiva em taxa de compressão nessas mesmas configurações, mas com tempos de execução menores, refletindo diferenças na organização interna do código, nas estruturas de dados utilizadas e nas decisões de projeto entre as duas bibliotecas.

Apesar de ambas as implementações serem escritas em Python e adotarem a mesma ideia básica de janela deslizante do LZ77, elas não são idênticas. A *Komm* segue mais de perto um modelo teórico, com tokens de formato bem definido e uma arquitetura modular pensada para generalidade e uso educacional. Já a *LZ77-Compressor* utiliza convenções próprias de representação dos tokens e uma estratégia de varredura ligeiramente diferente, o que pode levar a escolhas distintas de *matches* e, em alguns casos, a uma taxa de compressão um pouco melhor para determinados tipos de arquivo.

Na compressão da imagem *snail*, por exemplo, os resultados obtidos pela *LZ77-Compressor* foram superiores aos da biblioteca *Komm* em termos de taxa de compressão. Esse comportamento pode ser explicado pela forma como a *LZ77-Compressor* codifica os tokens e pelo conjunto específico de parâmetros (janela e *lookahead*) adotados, que acabaram se ajustando melhor aos padrões presentes nessa imagem. No entanto, o tempo de execução permaneceu elevado, o que reforça que as melhorias em compressão não vieram acompanhadas de uma otimização equivalente em desempenho.

Em resumo, embora as duas bibliotecas implementem variações do mesmo esquema LZ77 e utilizem parâmetros próximos, pequenas diferenças de convenção, formato de token e estratégia de busca são suficientes para produzir variações perceptíveis tanto na taxa de compressão quanto no tempo de execução.

4.4.3 Limitações da função `lz77enco` do Octave

Além das implementações externas em C e Python, também foi avaliada uma implementação didática do algoritmo *LZ77* no GNU Octave. Porém essa função apresentou problemas significativos em relação ao seu uso prático, conforme detalhado a seguir.

Código 4.1 – Exemplo do lz77enco do Octave.

```
1 lz77enco ([0 0 1 0 1 0 2 1 0 2 1 0 2 1 2 0 2 1 0 2 1 2 0 0], 3, 9, 18)
2
3 ans =
4    8   2   1
5    7   3   2
6    6   7   2
```

```
7    2   8   0
```

Código 4.2 – Exemplo do lz77enco dentro da Komm.

```
1  import komm
2
3  lz77 = komm.LempelZiv77Code(
4      window_size=18,
5      lookahead_size=9,
6      source_cardinality=3,
7  )
8
9  source = [0, 0, 1, 0, 1, 0, 2, 1, 0, 2, 1, 0, 2, 1, 2, 0, 2, 1, 0, 2, 1, 2, 0, 0]
10 lz77.source_to_tokens(source)
11 # [(8, 2, 1), (7, 3, 2), (6, 7, 2), (2, 8, 0)]
```

Seguindo a documentação da função `lz77enco`, o comando funciona como esperado e, ao fazer a comparação com a implementação da *Komm*, observa-se que o resultado gerado pelo `lz77enco` do Octave corresponde às expectativas. No entanto, ao aplicar a função a uma sequência contendo um único elemento adicional, o *lz77enco* do Octave deixa de operar corretamente. Já a implementação da *Komm*, em contraste, foi capaz de codificar para ambas entradas. Esse comportamento é mostrado a seguir:

Código 4.3 – Segundo exemplo do lz77enco do Octave.

```
1  lz77enco ([0 0 1 0 1 0 2 1 0 2 1 0 2 1 2 0 2 1 0 2 1 2 0 0 1], 3, 9, 18)
2
3  error: =: nonconformant arguments (op1 is 9x1, op2 is 1x11)
4  error: called from
5      lz77enco at line 61 column 7
```

Código 4.4 – Segundo exemplo do lz77enco dentro da Komm.

```
1  import komm
2
3  lz77 = komm.LempelZiv77Code(
4      window_size=18,
5      lookahead_size=9,
6      source_cardinality=3,
7  )
8
9  source = [0, 0, 1, 0, 1, 0, 2, 1, 0, 2, 1, 0, 2, 1, 2, 0, 2, 1, 0, 2, 1, 2, 0, 0, 1
       ]
10 lz77.source_to_tokens(source)
11 # [(8, 2, 1), (7, 3, 2), (6, 7, 2), (2, 8, 0), (8, 0, 1)]
```

Esse comportamento indica que a função `lz77enco` do Octave possui limitações significativas em sua implementação, não sendo capaz de lidar com casos simples de entrada. Dessa forma, sua utilização em experimentos práticos fica comprometida, reforçando a necessidade de implementações mais robustas e flexíveis, como a desenvolvida na *Komm*.

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 de tempo e memória; (iii) implementações em linguagens compiladas, como C, tendem a dominar em desempenho.

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.

Em suma, a implementação do *LZ77* na *Komm* mostrou-se competitiva em termos de taxa de compressão, especialmente quando configurada com janelas e *lookahead* adequados. Contudo, o tempo de execução permaneceu superior na implementação *FastLZ* em C, refletindo as limitações inerentes à linguagem Python. A implementação *LZ77-Compressor* em Python apresentou desempenho intermediário, com boa taxa de compressão, mas tempo de execução elevado em relação à *Komm*.

5 CONCLUSÃO

Este trabalho teve como objetivo implementar o algoritmo de compressão sem perdas *Lempel–Ziv 77* (LZ77) na biblioteca *Komm*, integrando-o ao conjunto de códigos-fonte abertos já disponíveis para o estudo de sistemas de comunicação digitais. A implementação buscou seguir fielmente o modelo descrito no artigo original de Ziv e Lempel (ZIV; LEMPEL, 1977), priorizando clareza didática, compatibilidade com a infraestrutura existente da biblioteca e modularidade para futuras extensões.

O desenvolvimento foi dividido em quatro métodos internos de codificação e decodificação, permitindo a inspeção direta dos tokens gerados e das etapas intermediárias do processo. Essa abordagem reforçou o caráter educacional da biblioteca, facilitando a visualização do funcionamento interno do LZ77 e a comparação com outros algoritmos já implementados, como Huffman, LZW e LZ78.

Os resultados experimentais confirmaram o comportamento clássico do LZ77: aumentos no tamanho da janela deslizante (W) proporcionaram melhor taxa de compressão, mas com crescimento proporcional do tempo de execução e do consumo de memória. A comparação com implementações externas, em especial o *FastLZ* em C, demonstrou que o desempenho inferior da versão em Python deve-se principalmente à ausência de estruturas de indexação otimizadas (como tabelas de hash) e ao modelo de tokens de tamanho fixo, que privilegia a simplicidade conceitual em detrimento da eficiência máxima.

Apesar dessas limitações, a integração do LZ77 à *Komm* mostrou-se bem-sucedida: todas as simulações preservaram a integridade dos dados, e a estrutura modular proposta oferece uma base sólida para pesquisa, ensino e experimentação de novos métodos de compressão.

5.1 TRABALHOS FUTUROS

Como desdobramento natural deste projeto, destacam-se as seguintes possibilidades de evolução:

- **Implementação da Codificação Aritmética:** continuação da proposta original, incorporando o algoritmo de codificação aritmética à *Komm*, o que permitirá avaliar esquemas híbridos e comparações diretas com Huffman e Tunstall em termos de entropia e eficiência.
- **Otimização do LZ77 com Tabelas de Hash:** desenvolver uma versão alternativa do *LempelZiv77Code* que utilize uma tabela de dispersão (*hash table*) para acelerar

a busca de correspondências, aproximando-se do desempenho de implementações como o *FastLZ*, sem perder a clareza e a compatibilidade do modelo atual.

- **Análise de Compressão Mista:** investigar combinações entre LZ77 e codificações de entropia (Huffman ou Aritmética), como ocorre no formato *DEFLATE*, avaliando o impacto na taxa e no tempo de compressão.
- **Estudo de Variações do LZ77:** explorar variantes do algoritmo, como LZSS, LZR, LZH e LZB implementando-as na *Komm* para ampliar o leque de opções disponíveis para estudo e comparação.

REFERÊNCIAS

ABRANTES, S.A. **Codificação de fonte: duas breves visitas**. FEUP Edições. ISBN 9789727520398. Disponível em: <https://books.google.pt/books?id=tKlChnUo2EsC>.

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_EMC>.

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.