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 NA BIBLIOTECA KOMM

SÃO JOSÉ

2025

Rhenzo Hideki Silva Kajikawa

Estudo e Implementação dos Algoritmos 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 dos Algoritmos 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.

Prof. Roberto Wanderley da Nóbrega, Dr.

Instituto Federal de Santa Catarina

Prof. Diego da Silva de Medeiros, 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." (Rhenzo, ANO)

RESUMO

A biblioteca de compressão de dados Komm será estendida com a integração de um algoritmos 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 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; Python.

ABSTRACT

The Komm data compression library will be extended by integrating an aditional lossless algorithms: 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 QUADROS

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 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 ultima 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 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

O resto deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os 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. Finalmente, 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, 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.

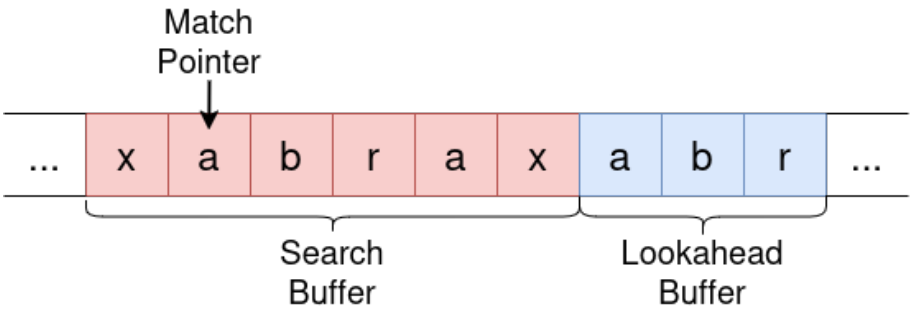
A saída do codificador é, portanto, uma sequência de tokens $\langle p_i, \ell_i, x_i \rangle$, que representam os fatores sucessivos do texto de entrada.

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 $\langle 6, 0, \texttt{d} \rangle$. Após o avanço da janela, o algoritmo encontra no histórico a sequência `"abra"`, gerando o token $\langle 0, 4, \texttt{r} \rangle$, e assim sucessivamente.

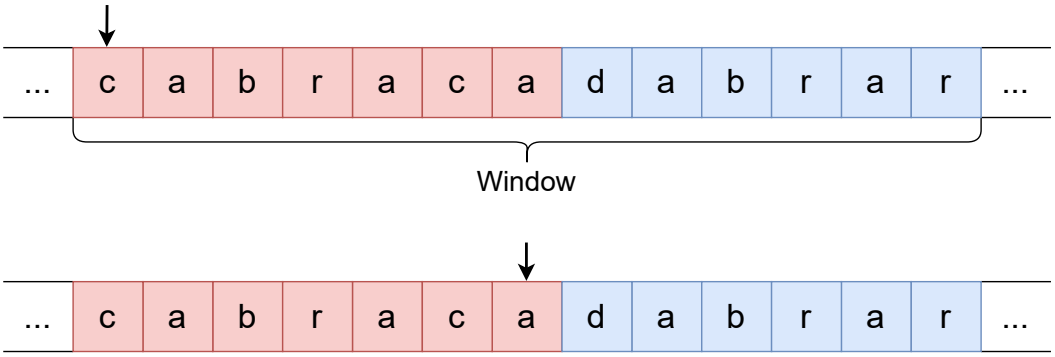
Durante a decodificação, cada token é expandido de acordo com p e ℓ, copiando o trecho correspondente do histórico decodificado e adicionando o símbolo literal x. Dessa forma, o processo é perfeitamente reversível e sem perdas.

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



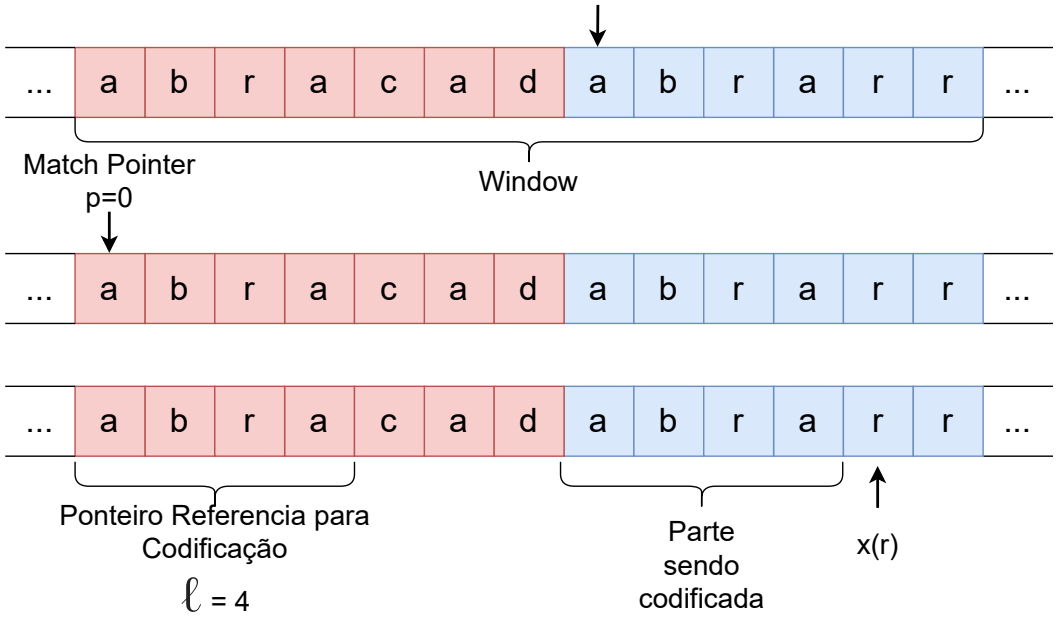
Fonte: Adaptada de Sayood (2012)

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

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 $\langle 0, 4, \texttt{r} \rangle$, 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).

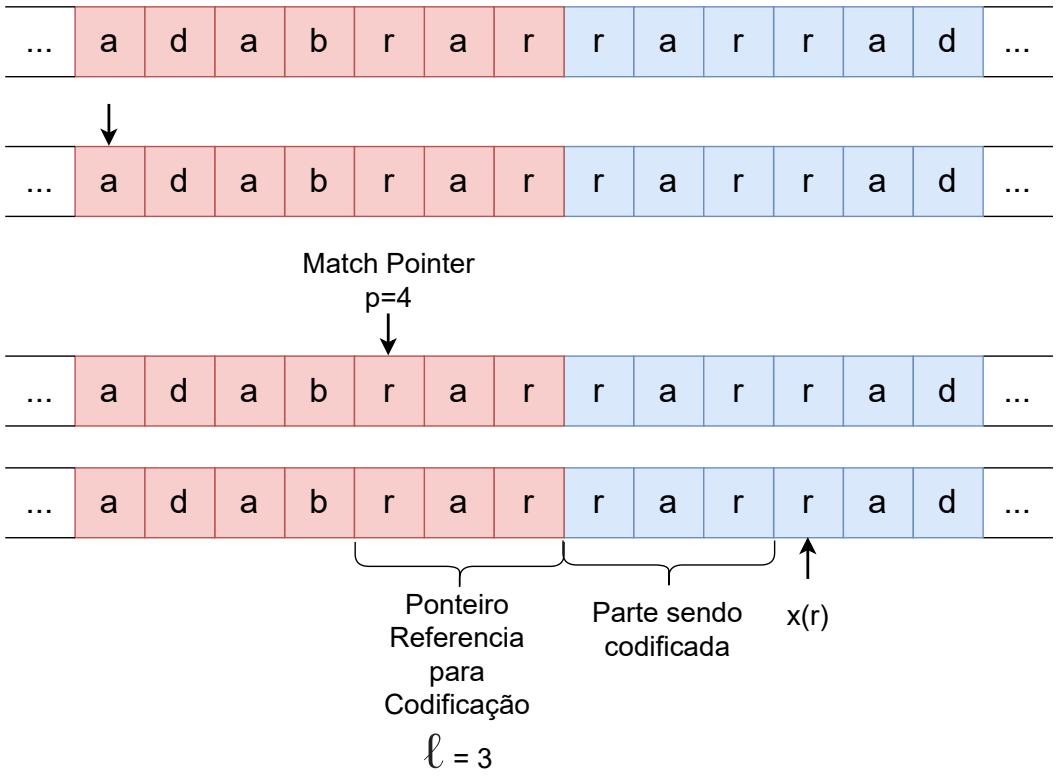
Após a segunda codificação, a janela deslizante avança novamente, e o processo se repete. Procura-se no *search buffer* a maior correspondência para o início do *lookahead buffer*, que agora começa com `"rar"`. Neste caso, a o deslocamento até a correspondência mais longa é 4, com comprimento 3 (`"rar"`), seguido do caractere literal `"r"`. Assim, o token gerado é $\langle 4, 3, \texttt{r} \rangle$, 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 (c), que neste exemplo é `"r"`.

Esse processo é ilustrado passo a passo na Figura 5. Inicialmente, há o estado

Figura 4 – Estado da janela após primeira codificação no algoritmo LZ77



Fonte: Adaptada de Sayood (2012)

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.

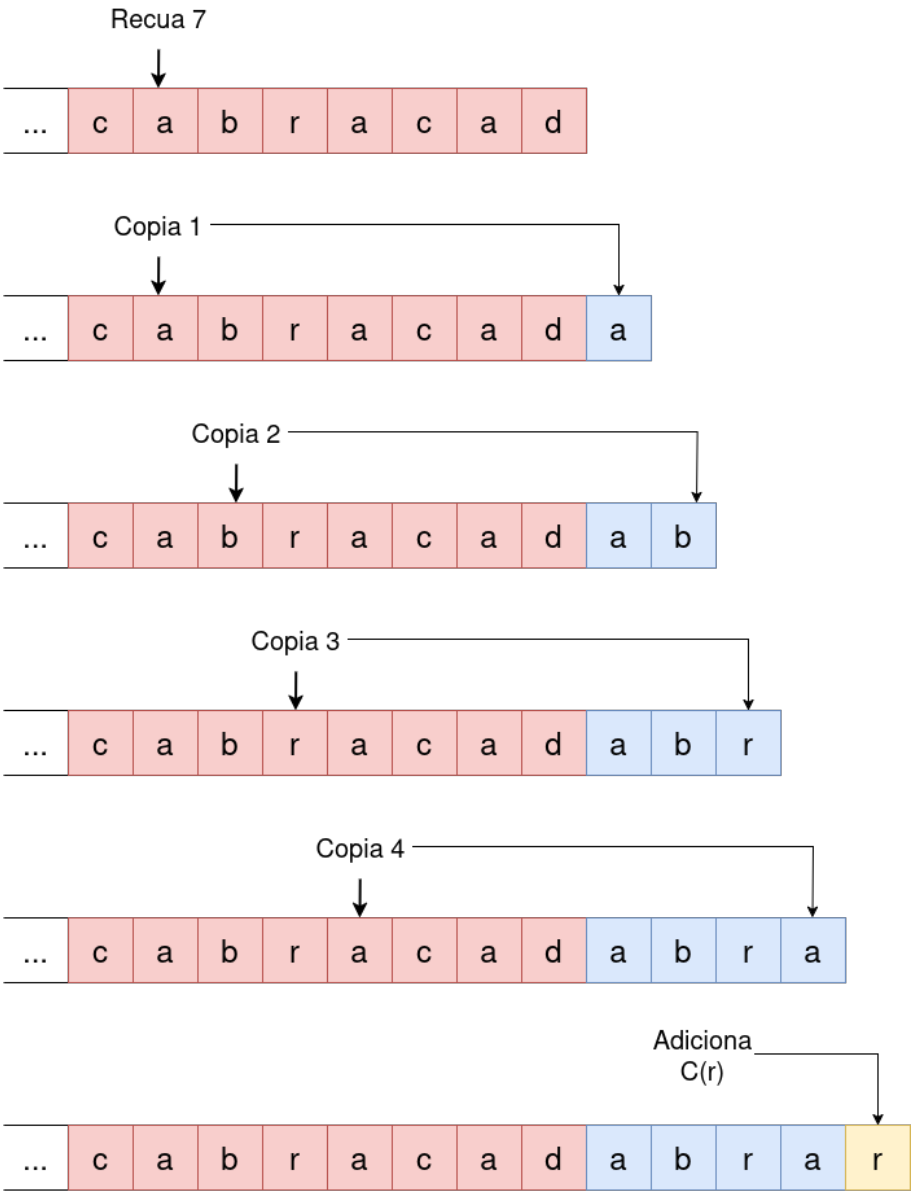
Utilizando o exemplo anterior do token $\langle 7, 4, \texttt{r} \rangle$, onde a janela total possui 13 caracteres, temos a seguinte definição dos campos em bits:

- 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

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

Figura 5 – Decodificação do exemplo $\langle 7, 4, \texttt{r} \rangle$



Fonte: Adaptada de Sayood (2012)

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 ($\langle 0, 4, \texttt{r} \rangle$), 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* 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

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*.

```python
import komm

# Instanciação do codificador
lz77 = komm.LempelZiv77Code(
    window_size=13,
    lookahead_size=6,
    source_cardinality=256,
    target_cardinality=2
)

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

# Codificação e decodificação
encoded = lz77.encode(source)
decoded = lz77.decode(encoded)

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.

```python
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
```

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. 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.

3.1.4 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. Implementações voltadas a desempenho, como as presentes em bibliotecas escritas em C (por exemplo, *FastLZ*), ou abordagens discutidas por Casablanca (2023), utilizam tabelas de hash para armazenar ocorrências de subsequências de comprimento fixo, permitindo localizar candidatos de correspondência de forma muito mais rápida. Nesses casos, a busca é reduzida a poucas operações de leitura e comparação, ao custo de maior uso de memória.

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.

3.1.5 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 Sayood (SAYOOD, 2012), que verifica a correspondência entre a sequência original e o conjunto de tokens gerados:

Código 3.3 – Teste unitário baseado no exemplo de Sayood (p. 122).

```
def test_lz77_sayood():
    # [Sayood, p. 122]
    code = komm.LempelZiv77Code(
        window_size=13,
        lookahead_size=6,
        source_cardinality=256,
        search_buffer=[ord(x) for x in "cabraca"],
    )
    source = [ord(x) for x in "dabrarrarrad"]
    expected = [(1, 0, "d"), (7, 4, "r"), (3, 5, "d")]
    tokens = []
    for offset, length, symbol in expected:
        tokens.append((7 - offset, length, ord(symbol)))
    np.testing.assert_equal(code.source_to_tokens(source), tokens)
    np.testing.assert_equal(code.tokens_to_source(tokens), source)
    np.testing.assert_equal(code.decode(code.encode(source)), source)
```

Esse teste reproduz o exemplo da codificação de `"cabracadabrarrarrad"`, utilizando o mesmo tamanho de janela e *lookahead* definidos por Sayood. O teste verifica três propriedades centrais do algoritmo:

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

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: (i) **textos** e (ii) **imagens**. O texto escolhido foi 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 tamanho de 154.573 bytes e apresenta ampla variedade de caracteres e repetições, sendo adequado para testar diferentes tamanhos de janela no *LZ77*.

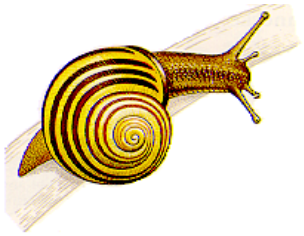
As imagens utilizadas foram bitmaps (**BMP**) simples, escolhidas por possuírem regiões de baixa entropia e padrões espaciais redundantes, que permitem observar o comportamento do algoritmo em fontes bidimensionais. A Figura 6 mostra a imagem *smiley* (246 bytes) e a Figura 7 a imagem *snail* (196.666 bytes).

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

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

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

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`.

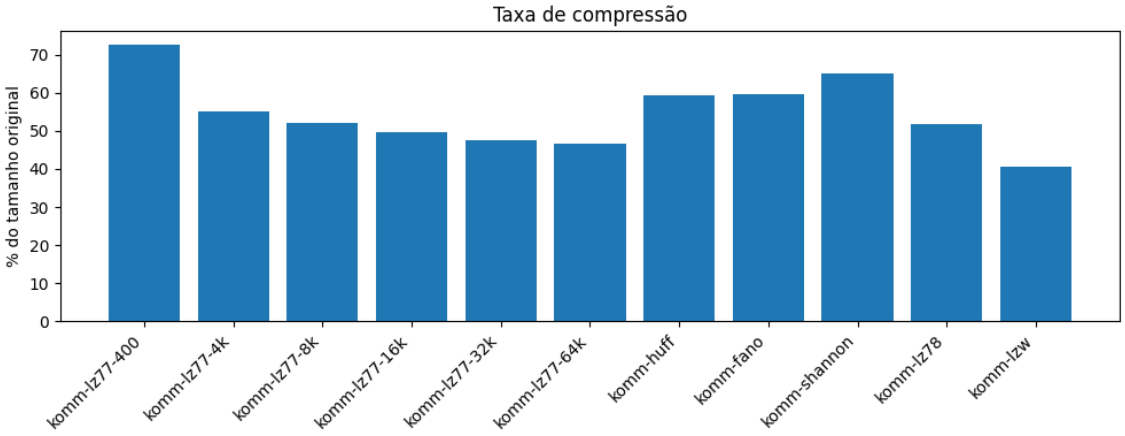
4.3 RESULTADOS DO LZ77 NA *KOMM*

Para avaliar o desempenho do *LZ77* na *Komm*, foram realizados testes variando o tamanho da janela W entre 100 bytes e 64 kB, mantendo o tamanho do *lookahead* L fixo em 16 bytes. Os resultados foram comparados com os algoritmos de compressão Huffman, Shannon–Fano, LZ78 e LZW disponíveis na biblioteca.

Análise (texto):

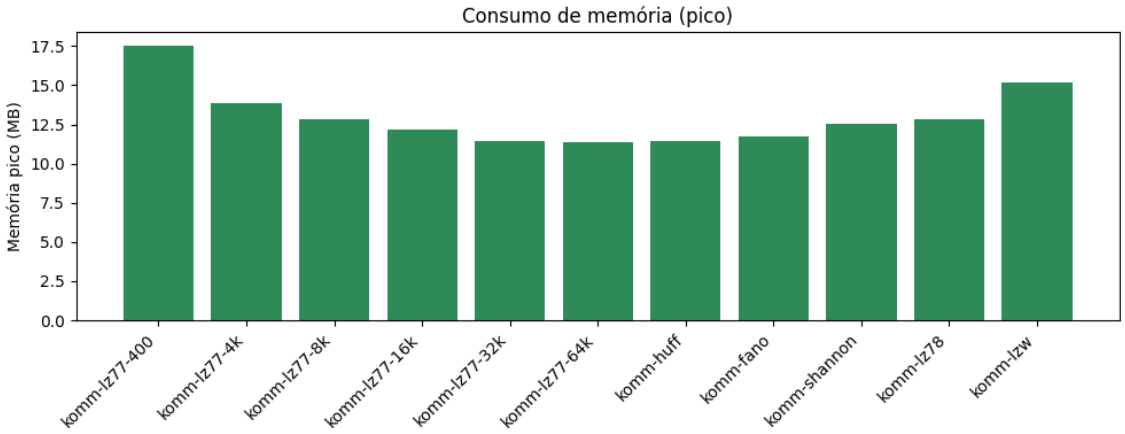
- O *LZ77* apresentou o *trade-off* esperado: janelas maiores proporcionam melhor compressão, mas aumentam o tempo e o consumo de memória.
- Para o arquivo *Alice*, o ganho de compressão é significativo ao aumentar W de 400 B para 32 kB, estabilizando a partir de 64 kB.
- Em comparação, Huffman e Shannon–Fano mantêm desempenho rápido e baixo uso de memória, porém com taxas de compressão inferiores.

Figura 8 – Texto *Alice*: taxa de compressão.



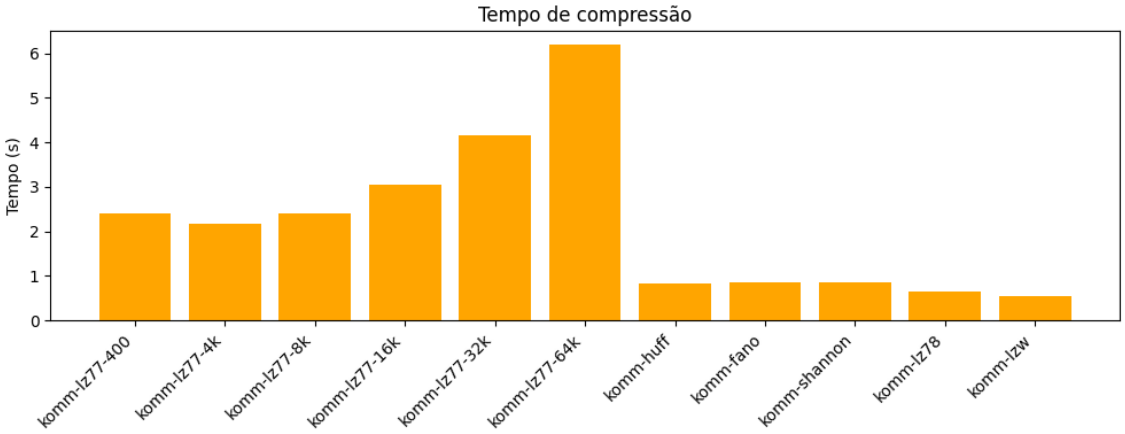
Fonte: Elaborada pelo autor.

Figura 9 – Texto *Alice*: memória pico.



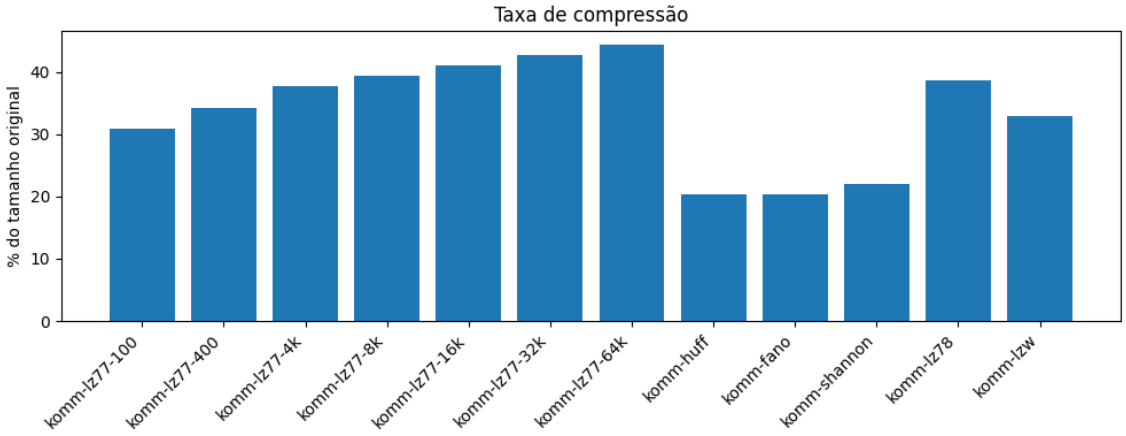
Fonte: Elaborada pelo autor.

Figura 10 – Texto *Alice*: tempo de compressão.



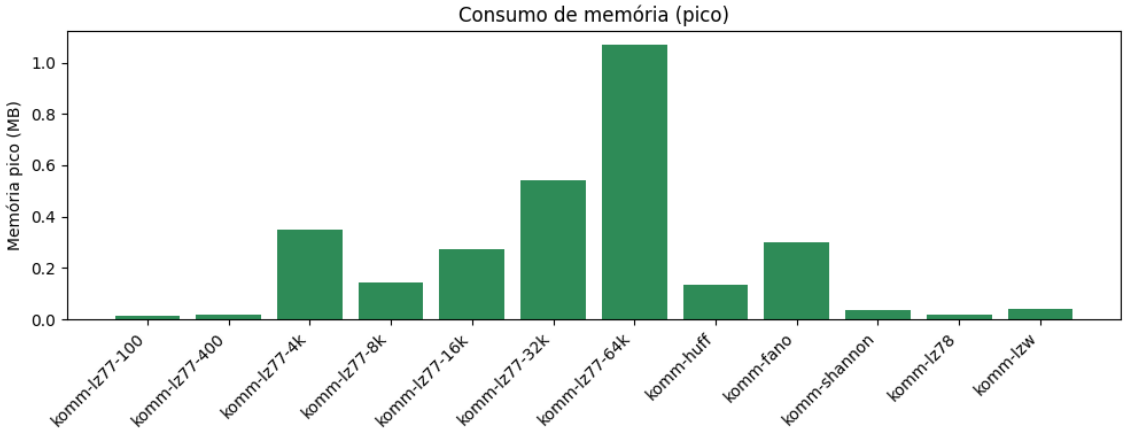
Fonte: Elaborada pelo autor.

Figura 11 – Imagem *smiley*: taxa de compressão.



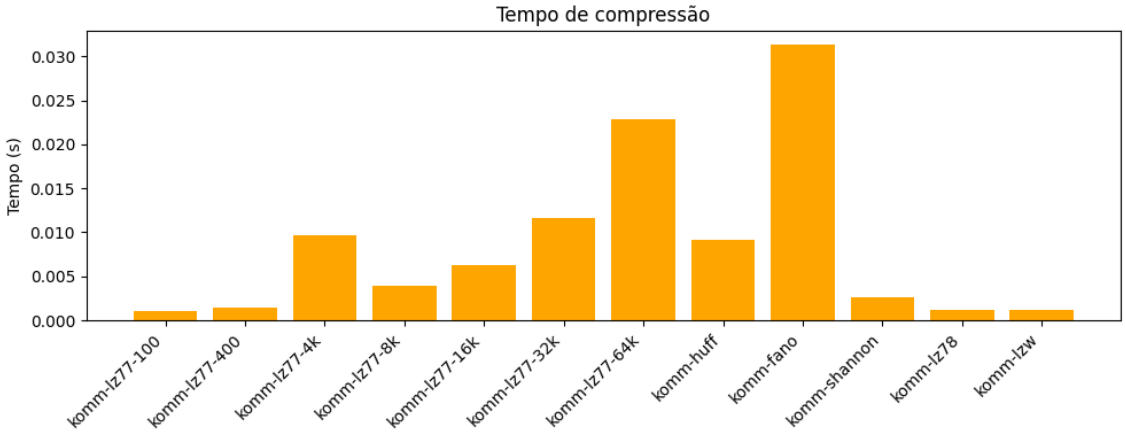
Fonte: Elaborada pelo autor.

Figura 12 – Imagem *smiley*: memória pico.



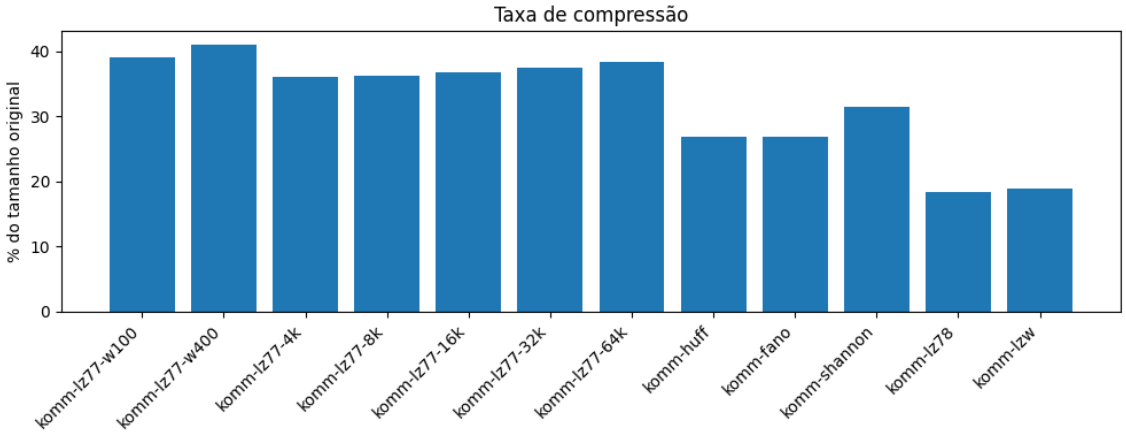
Fonte: Elaborada pelo autor.

Figura 13 – Imagem *smiley*: tempo de compressão.



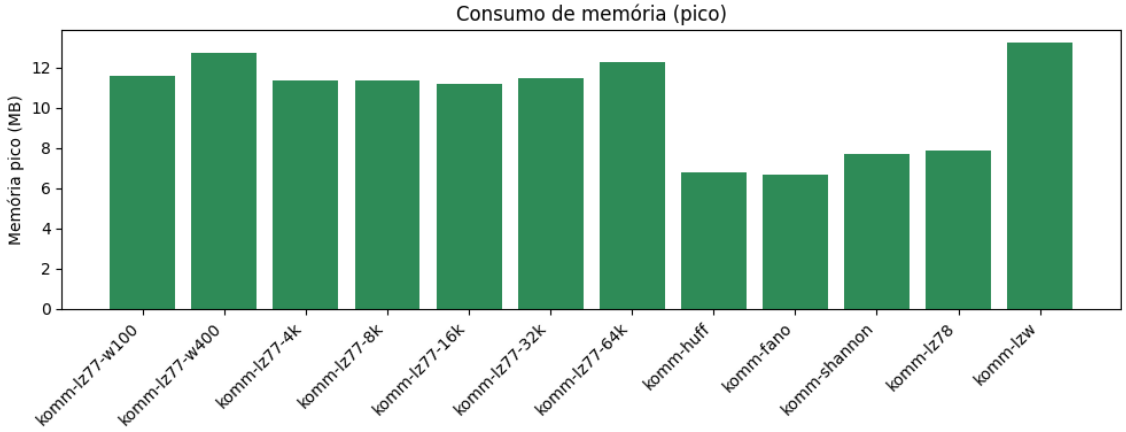
Fonte: Elaborada pelo autor.

Figura 14 – Imagem *snail*: taxa de compressão.



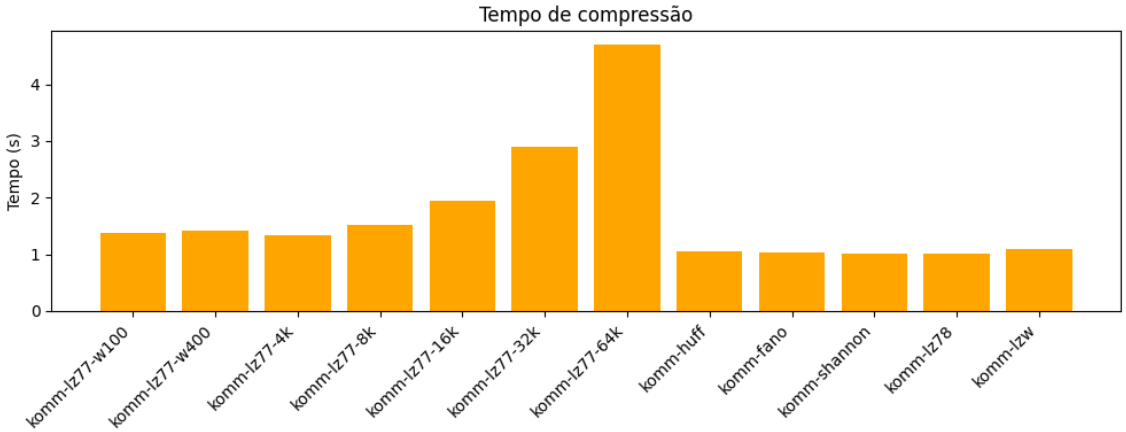
Fonte: Elaborada pelo autor.

Figura 15 – Imagem *snail*: memória pico.



Fonte: Elaborada pelo autor.

Figura 16 – Imagem *snail*: tempo de compressão.



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.

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.

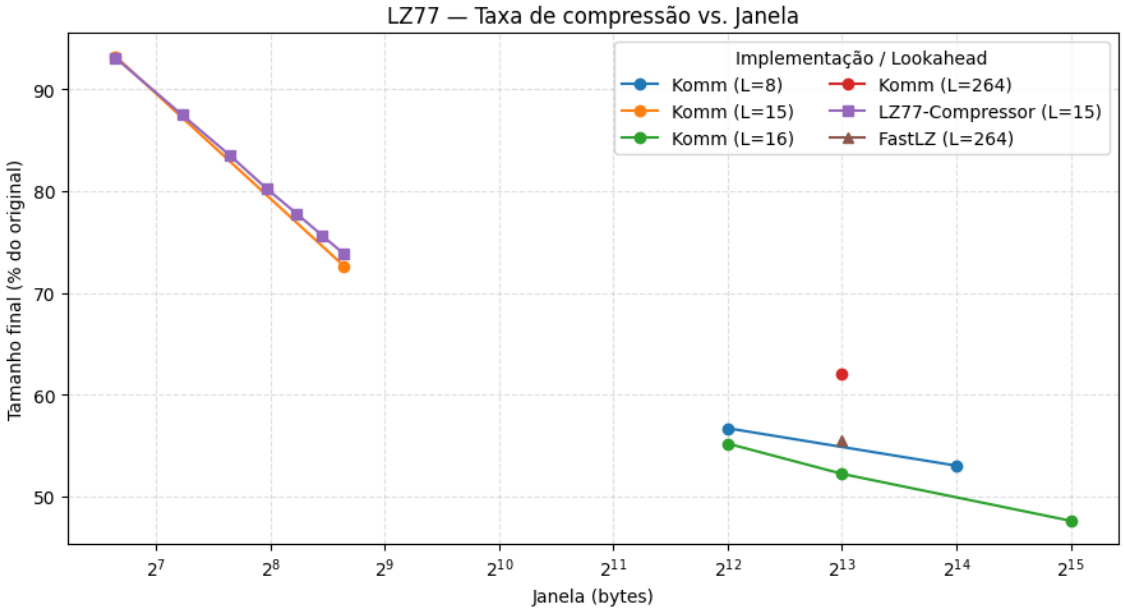
4.4 COMPARAÇÃO COM IMPLEMENTAÇÕES EXTERNAS DE LZ77

Foram consideradas duas implementações populares do algoritmo *LZ77*: **FastLZ**[2] (em C) e **LZ77-Compressor**[3] (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 $W = 64\,\text{kB}$.

Figura 17 – Texto *Alice*: taxa de compressão.

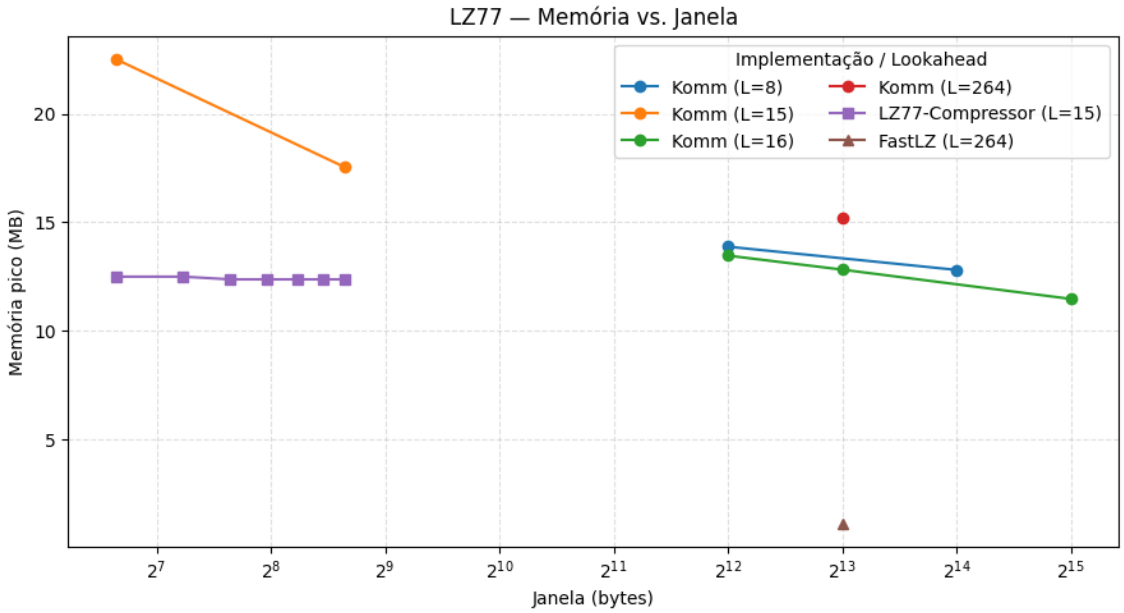


Fonte: Elaborada pelo autor.

Análise (texto):

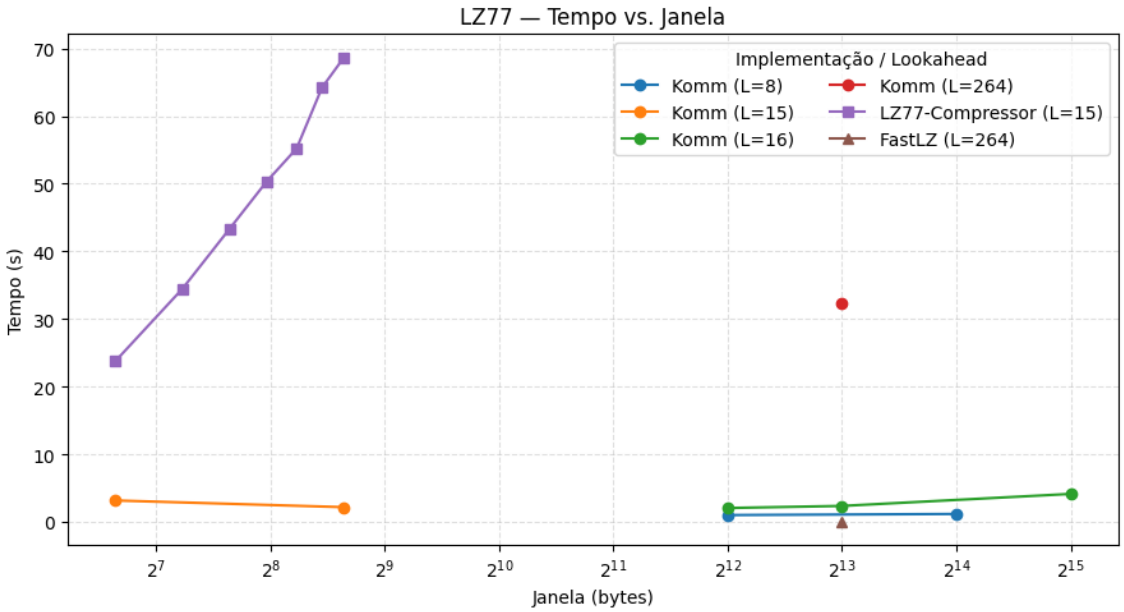
- A implementação *FastLZ* apresentou tempos de compressão inferiores, resultado coerente com sua implementação em C voltada à velocidade.
- A versão da *Komm*, com $W = 64\,\text{kB}$, obteve taxas de compressão próximas, mostrando boa eficiência mesmo em Python.
- A biblioteca *LZ77-Compressor* apresentou maior tempo de execução, mas consumo de memória reduzido.

Figura 18 – Texto *Alice*: memória pico.



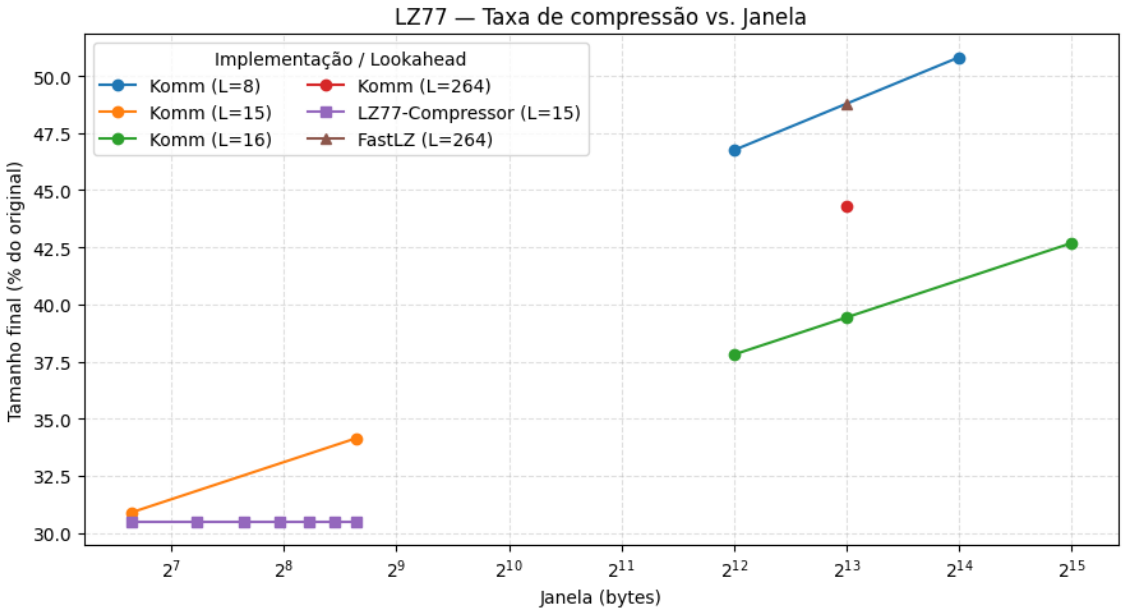
Fonte: Elaborada pelo autor.

Figura 19 – Texto *Alice*: tempo de compressão.



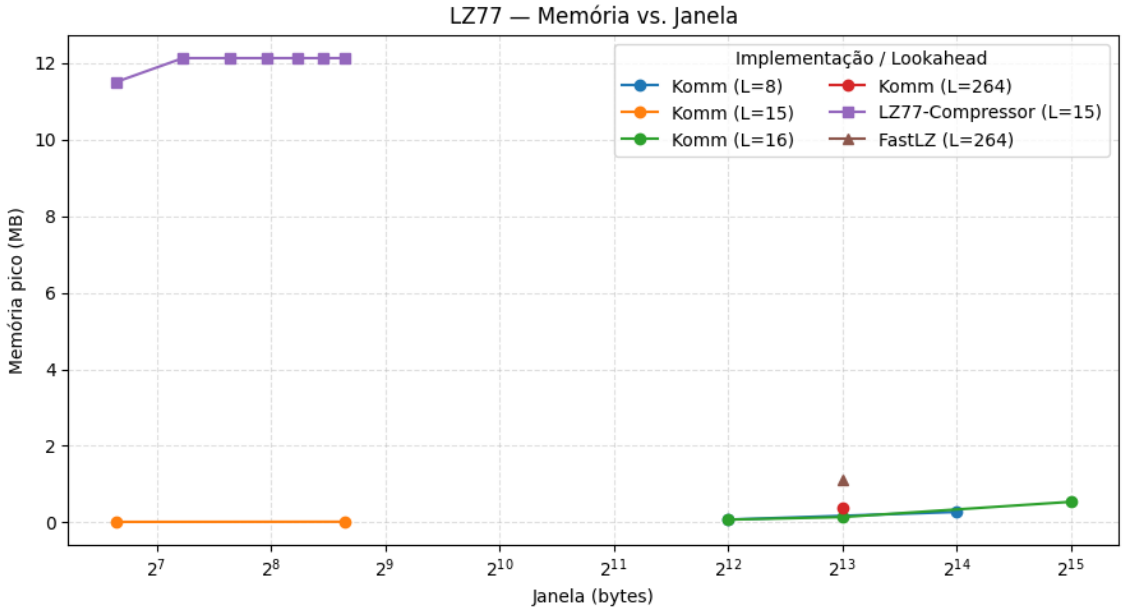
Fonte: Elaborada pelo autor.

Figura 20 – Imagem *smiley*: taxa de compressão.



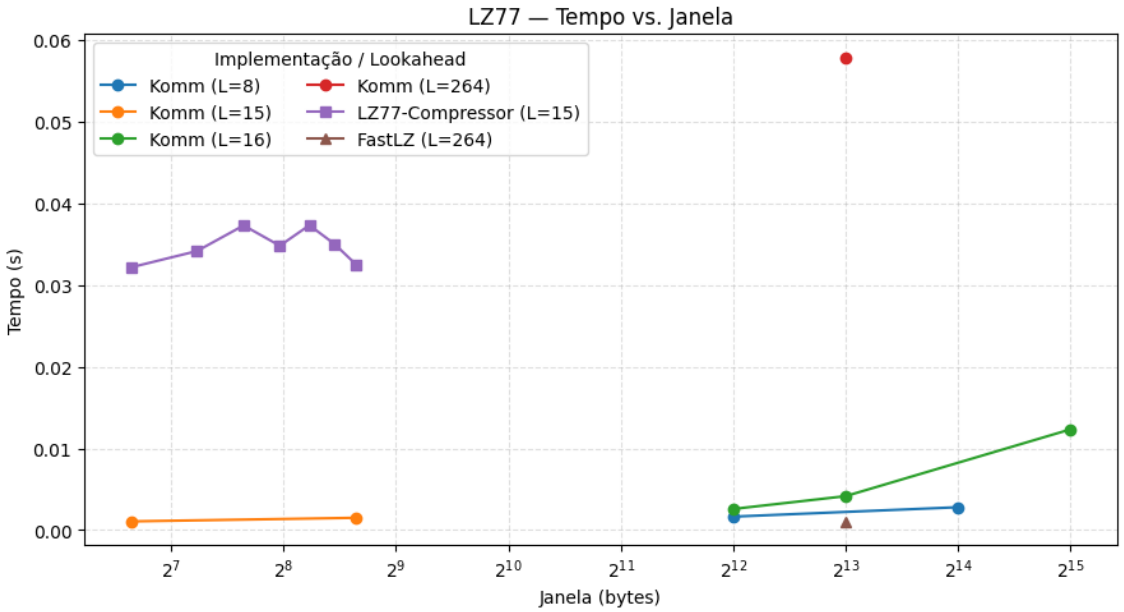
Fonte: Elaborada pelo autor.

Figura 21 – Imagem *smiley*: memória pico.



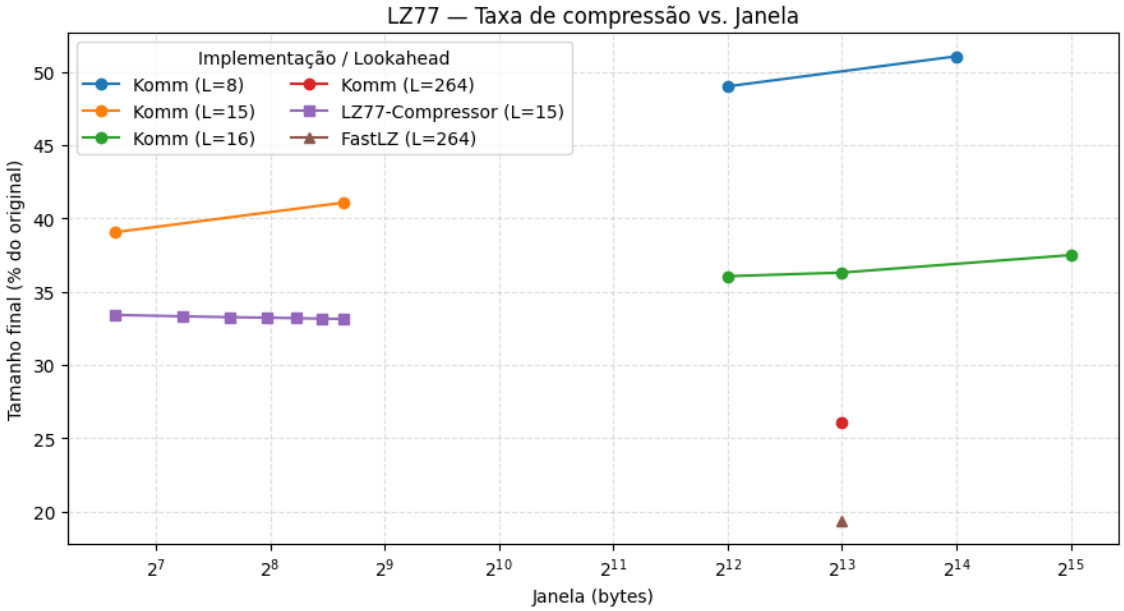
Fonte: Elaborada pelo autor.

Figura 22 – Imagem *smiley*: tempo de compressão.



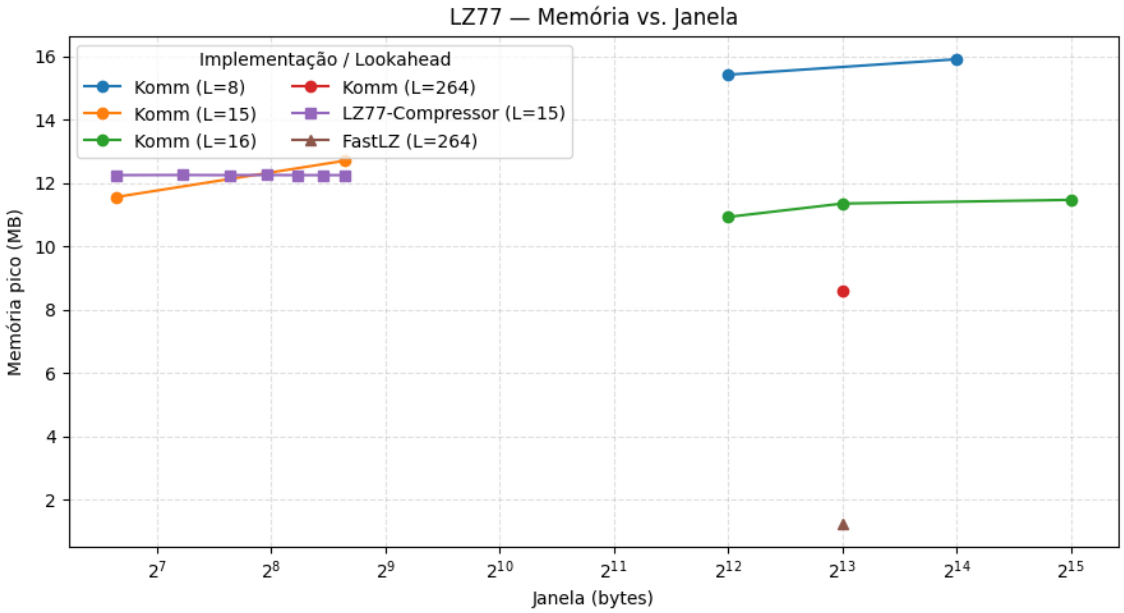
Fonte: Elaborada pelo autor.

Figura 23 – Imagem *snail*: taxa de compressão.



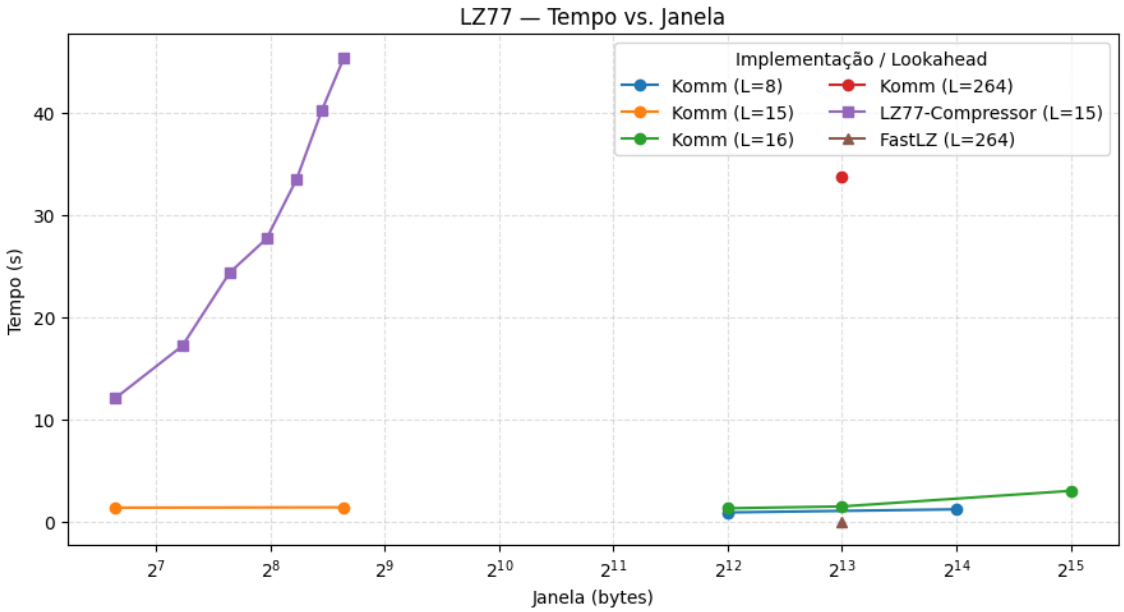
Fonte: Elaborada pelo autor.

Figura 24 – Imagem *snail*: memória pico.



Fonte: Elaborada pelo autor.

Figura 25 – Imagem *snail*: tempo de compressão.



Fonte: Elaborada pelo autor.

Análise (imagens):

- A implementação da *Komm* competitiva comparada as implementações externas em relação a taxa de compressão.
- O *FastLZ* na sua grande maioria.
- 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.

4.5.1 Diferenças entre as implementações em C e Komm

Embora os experimentos tenham utilizado parâmetros equivalentes ($W = 8$ kB, $L = 264$ bytes), 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 overhead de representação.

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.

[2] https://github.com/ariya/FastLZ
[3] https://github.com/manassra/LZ77-Compressor

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

1. **Modelo de codificação:** a versão da *Komm* utiliza tokens de tamanho fixo e separação explícita dos campos p, ℓ e x, enquanto o *FastLZ* emprega um formato variável, agrupando literais e matches em blocos compactos.
2. **Estratégia de busca:** o uso de `rfind()` em Python realiza comparações diretas de cadeias, enquanto o *FastLZ* utiliza índices hash para reduzir o número de comparações.

Essas diferenças justificam tanto a melhor taxa de compressão do *FastLZ* quanto seu tempo de execução reduzido (milissegundos contra dezenas de segundos na versão em Python). Contudo, a implementação na *Komm* apresenta maior transparência conceitual e modularidade, sendo mais adequada para ensino, extensão e análise experimental do algoritmo.

4.5.2 Limitações da função `lz77enco` do Octave

Além das implementações externas em C e Python, também foi avaliado 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.

```
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)

ans =
   8   2   1
   7   3   2
   6   7   2
   2   8   0
```

Seguindo a documentação da função `lz77enco`, o comando funciona como esperado, e também fazendo a comparação com a implementação da *Komm*, observa-se que o resultado gerado pelo `lz77enco` do Octave correponde com as expectativas. Porém, ao tentar codificar uma sequência com um elemento adicional no sequencia, a biblioteca do Octave já não é capaz de codificar corretamente:

Código 4.2 – Segundo exemplo do lz77enco do Octave.

```
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)

error: =: nonconformant arguments (op1 is 9x1, op2 is 1x11)
error: called from
    lz77enco at line 61 column 7
```

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*.

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 módulos 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.

REFERÊNCIAS

CASABLANCA, Andrés Correa. **Exploring the LZ77 Algorithm**. Acesso em: 6 nov. 2025. 2023. Disponível em: <https://blog.coderspirit.xyz/blog/2023/06/04/exploring-the-lz77-algorithm/>.

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.