INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE SANTA CATARINA

CAMPUS SÃO JOSÉ

RHENZO HIDEKI SILVA KAJIKAWA

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

SÃO JOSÉ

2025

Rhenzo Hideki Silva Kajikawa

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

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

Área de concentração: Telecomunicações

Orientador: Prof. Roberto Wanderley da Nóbrega, Dr.

São José

2025

Rhenzo Hideki Silva Kajikawa

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

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

São José, 16 de abril de 2025.

Prof. Roberto Wanderley da Nóbrega, Dr.

Instituto Federal de Santa Catarina

Professor Fulano, Dr.

Instituto Federal de Santa Catarina

Professora Fulana, Dra.

Instituto Federal de Santa Catarina

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

AGRADECIMENTOS

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

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

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

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

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

RESUMO

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

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

ABSTRACT

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

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

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

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 e nem de codificação aritmética. 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 e codificação aritmética para torná-la mais completa. Portanto, este trabalho tem como objetivo desenvolver e integrar um versão do LZ77 e um codificador aritmético 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 e codificação aritmética.

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

1.3 ORGANIZAÇÃO DO TEXTO

A ser decidido conforme escrito o TCC

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção será discutido o funcionamento e as principais características do LZ77 e do algoritmo aritmético, 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. No LZ77, o dicionário é construído dinamicamente a partir de partes já codificadas da própria entrada, o que dispensa a necessidade de um dicionário fixo ou pré-definido. Este é utilizado como base em algoritmos amplamente adotados, como o DEFLATE.

Para realizar a codificação, o algoritmo utiliza uma janela deslizante que percorre a sequência de entrada. Essa janela é dividida em duas regiões principais:

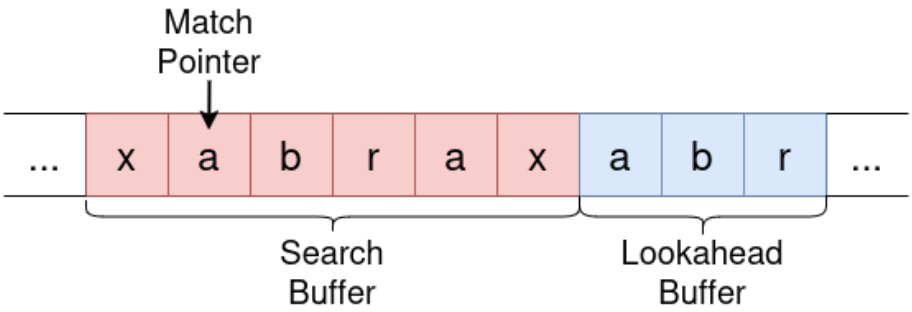
- ***Search buffer***: contém a porção já codificada da sequência, armazenando até n caracteres recentes. Ele funciona como o "dicionário" atual, onde o codificador tenta encontrar correspondências com os próximos caracteres a serem codificados.
- ***Lookahead buffer***: armazena os próximos caracteres a serem analisados e codificados. O codificador busca no *search buffer* a maior sequência que coincida com os caracteres do *lookahead buffer*.

Durante a codificação, o algoritmo utiliza um ponteiro de correspondência (*match pointer*) que percorre o *search buffer* para identificar o maior prefixo do *lookahead buffer* que também ocorra ali. Quando uma correspondência é encontrada, ela é codificada como um trio $\langle o, l, c \rangle$, onde:

- o (*offset*): distância entre o início da correspondência no *search buffer* e a posição atual da janela;
- l (*length*): comprimento da sequência que será codificada;
- c (*character*): próximo caractere literal após a sequência correspondente.

A Figura 1 ilustra essa estrutura, destacando as divisões da janela deslizante e o funcionamento da busca por correspondências.

Figura 1 – Divisão da janela deslizante no algoritmo LZ77



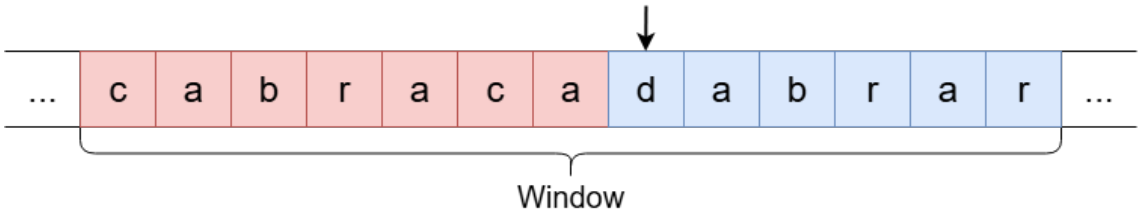
Fonte: Adaptada de Sayood (2012)

Este mecanismo permite substituir sequências repetidas por referências a ocorrências anteriores, resultando em compressões eficazes especialmente para dados com alta redundância.

2.1.1 Exemplo de Codificação com LZ77

Nesta seção será demonstrado um exemplo detalhado do funcionamento da janela deslizante e da representação do trio $\langle o, l, c \rangle$ 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. O estado inicial da janela pode ser visto na Figura 2.

Figura 2 – Estado inicial da janela deslizante no algoritmo LZ77

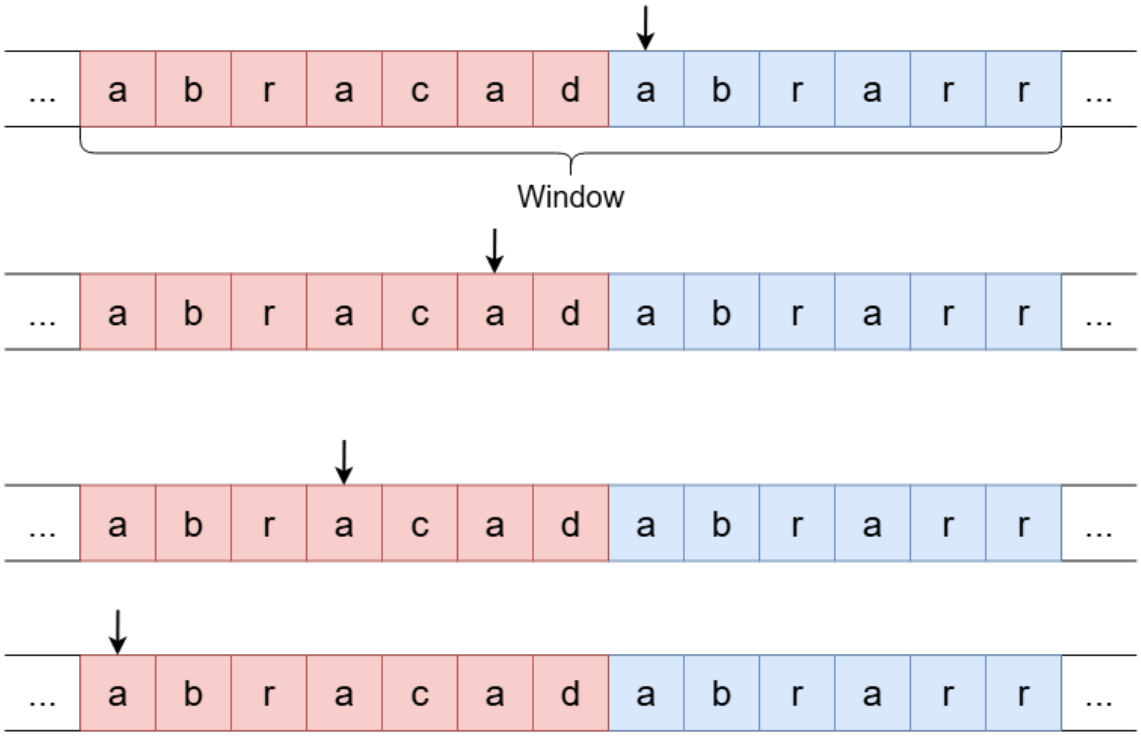


Fonte: Adaptada de Sayood (2012)

Inicialmente, busca-se no *search buffer* 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 trio $\langle 0, 0, \texttt{d} \rangle$, indicando que não houve correspondência (0 de deslocamento e 0 de comprimento) e que o caractere literal transmitido é `"d"`.

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 após primeira 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"`, porém, busca-se sempre a correspondência mais longa possível. Notavelmente, existe uma sequência completa `"abra"` previamente codificada, iniciando a 7 caracteres de distância da posição atual da janela. Essa correspondência possui comprimento 4 caracteres.

Dessa forma, o algoritmo codifica a sequência encontrada como o trio $\langle 7, 4, \texttt{r} \rangle$, onde 7 indica a distância até o início da correspondência no *search buffer*, 4 indica o comprimento da correspondência encontrada (`"abra"`), e `"r"` é o caractere seguinte imediatamente após essa sequência, ainda não codificado. Após isso, a janela avança em 5 posições (4 caracteres da sequência codificada mais 1 caractere literal).

Para realizar o processo inverso, ou seja, decodificar o trio recebido $\langle 7, 4, \texttt{r} \rangle$, o decodificador utiliza o mesmo princípio do algoritmo LZ77, porém no sentido inverso.

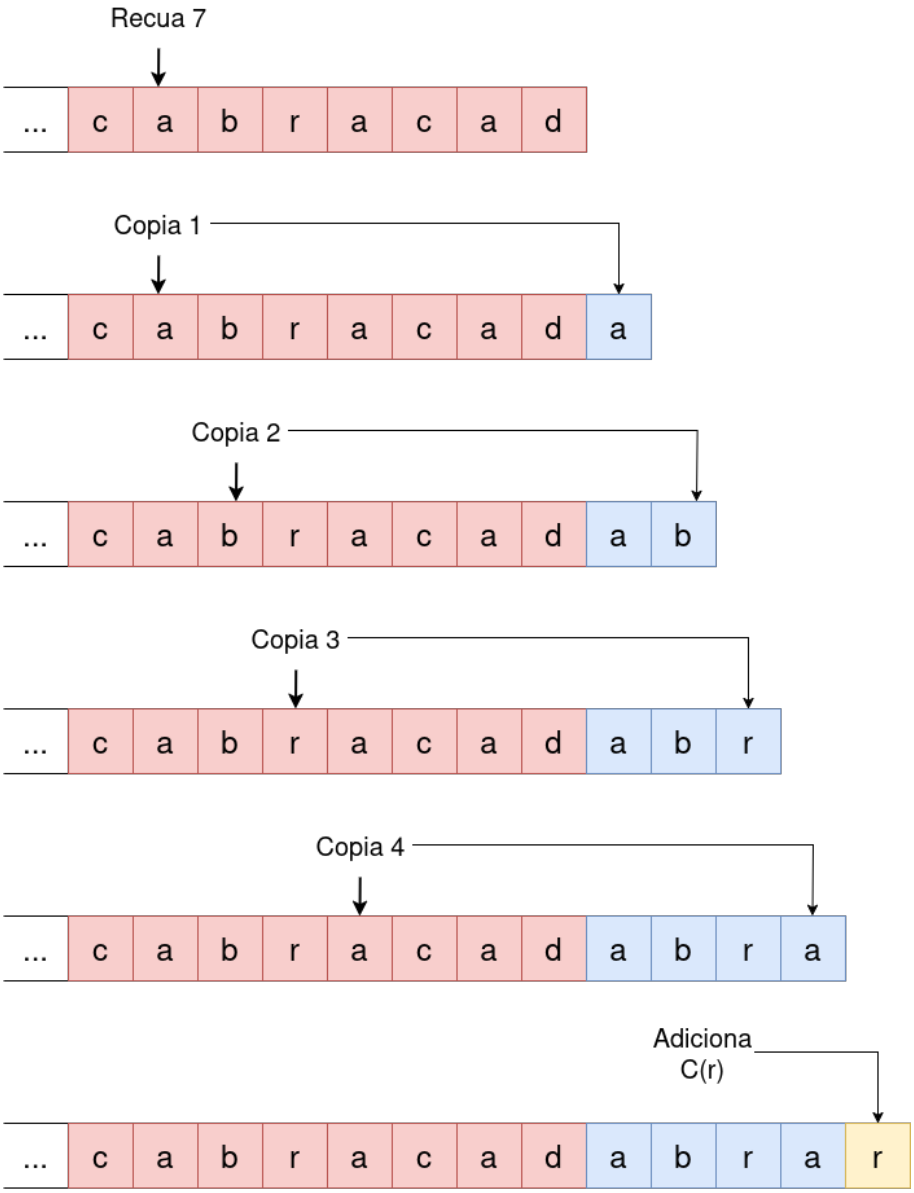
Inicialmente, ele utiliza o offset (o) para retornar exatamente 7 posições na sequência já decodificada até o momento. A partir dessa posição inicial encontrada, copia-se uma sequência de comprimento 4 (valor l), obtendo o trecho `"abra"`. Em seguida, adiciona-se ao final desta sequência copiada o caractere literal adicional (c), que neste exemplo é `"r"`.

Esse processo é ilustrado passo a passo na Figura 4. Inicialmente, há o estado parcial da decodificação com o *buffer* já reconstruído. Em seguida, avança-se caractere a

caractere, copiando-se do *buffer* reconstruído e adicionando o caractere literal no final. O resultado final após a decodificação deste trio será `"abrar"`.

Vale destacar que o decodificador reconstrói o buffer de busca dinamicamente, conforme recebe e processa novos trios, permitindo a reconstrução exata dos dados originais sem perda alguma.

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



Fonte: Adaptada de Sayood (2012)

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

- O deslocamento (o) pode variar de 0 a 7, portanto requer 3 bits;
- O comprimento (l) pode variar de 0 a 6, 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 trio são

$$\underbrace{3}_{\substack{\text{bits para}\\ \text{offset } (o)}} + \underbrace{3}_{\substack{\text{bits para}\\ \text{length } (l)}} + \underbrace{8}_{\substack{\text{bits para}\\ \text{character } (c)}} = 14 \text{ bits},$$

temos um formato de código de comprimento fixo, no qual cada trio $\langle o, l, c\rangle$ ocupará exatamente 14 bits. Em outras palavras, após definido o tamanho da janela (*offset*) 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 ($\langle 7, 4, \texttt{r}\rangle$), obtêm-se:

- O valor do *offset* $o = 7$, em 3 bits é $111_{(2)}$.
- 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 trio em bits será:

$$111 \mid 100 \mid 01110010$$

Resultando na sequência binária final: $11110001110010_{(2)}$.

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 ALGORITMO ARITMÉTICO

O algoritmo aritmético de compressão foi desenvolvido simultaneamente por Jorma J. Rissanen, da IBM Research, e por Richard Pasco, da Universidade de Stanford, tendo ambos publicado seus artigos em maio de 1976 (PASCO, 1976), (RISSANEN, 1976). Para uma explicação mais detalhada, pode-se consultar também a série de vídeos `"Information Theory"` por mathematicalmonk

O algoritmo aritmético é um método de codificação por fluxo (*stream code*) que representa uma sequência completa de símbolos por meio de um único número real fracionário pertencente ao intervalo $[0, 1[$. Considerando que o conjunto dos números reais nesse intervalo é infinito, torna-se possível atribuir uma *tag* exclusiva para cada sequência distinta de símbolos, estreitando progressivamente esse intervalo conforme cada símbolo é processado.

Ao contrário de métodos que atribuem códigos fixos ou códigos prefixos, como o Huffman, o algoritmo aritmético utiliza diretamente as probabilidades de cada símbolo para subdividir e refinar o intervalo, alcançando, teoricamente, taxas de compressão próximas ao limite da entropia da fonte.

2.2.1 Algoritmo com precisão infinita

Para esta primeira análise, será assumido que não há problemas relacionados à divisão infinita dos valores. Portanto, supõe-se precisão infinita durante os cálculos realizados pelo algoritmo.

Observação: Na codificação aritmética é comum, e altamente recomendável, reservar um símbolo especial para indicar o fim da mensagem, conhecido como *End of File* (EoF). Isso se deve ao fato de que a codificação resulta em um único número fracionário e, sem o EoF, o decodificador não teria como saber quando a mensagem termina, a menos que o seu comprimento total seja previamente conhecido. O uso de um símbolo EoF torna a decodificação autossuficiente e precisa.

Considere o exemplo definido pelos seguintes parâmetros:

$$
\begin{aligned}
X &= \{a, b, c\} \\
\text{EoF (End of File)} &: \ a \\
p &= (p_1, p_2, p_3) = (p_a, p_b, p_c) \\
p &= (p_1, p_2, p_3) = (0.2,\ 0.4,\ 0.4) \\
c &= (c_1, c_2, c_3) = (0,\ 0.2,\ 0.6)
\end{aligned}
$$

onde:

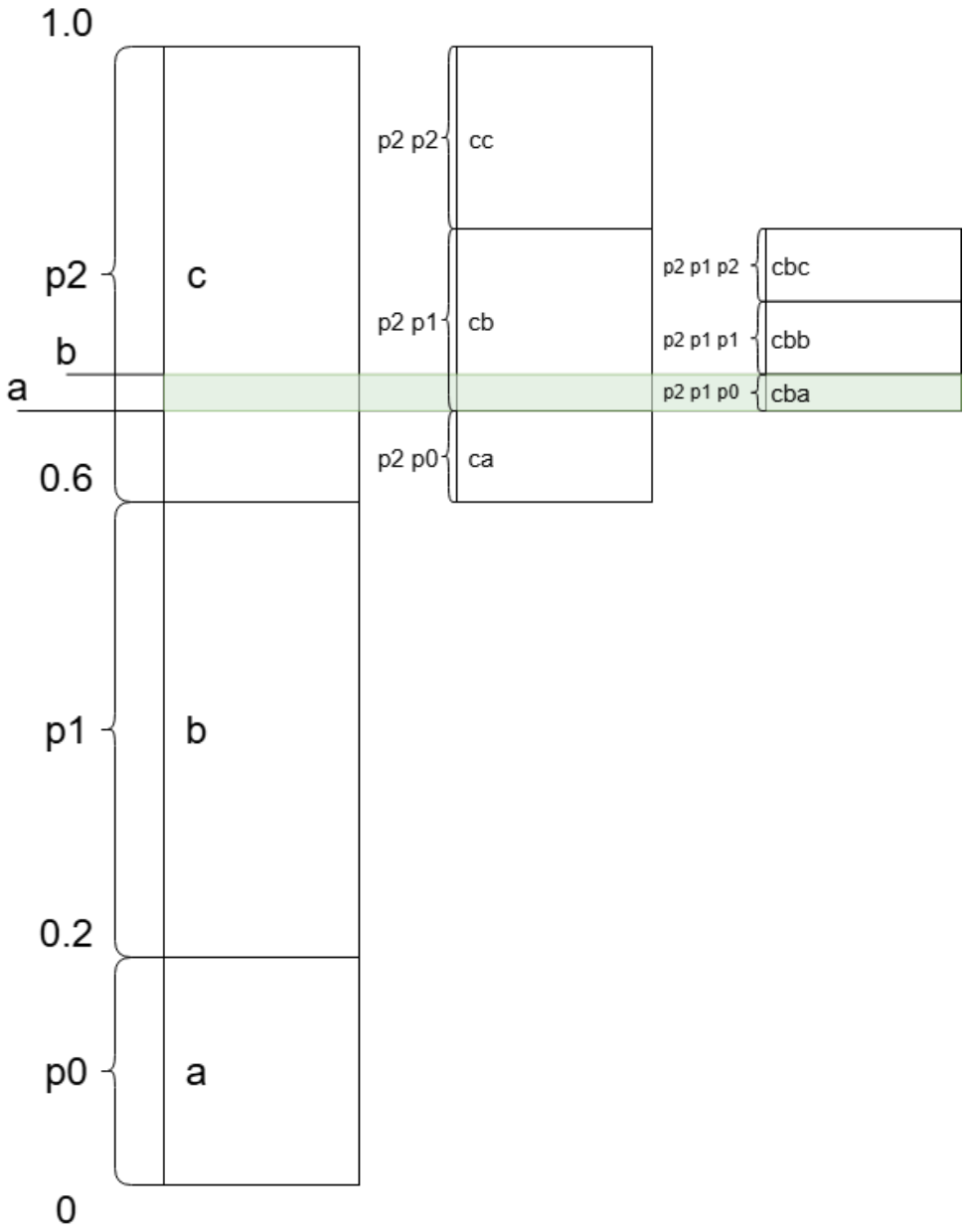
- X é o alfabeto dos símbolos possíveis;
- EoF (End of File) é o símbolo especial que indica o fim da sequência;
- p_i é a probabilidade individual associada a cada símbolo i;
- c_i é a probabilidade acumulada, definida por:

$$c_i = \sum_{j=0}^{i-1} p_j.$$

A Figura 5 ilustra o processo de codificação aritmética para a sequência `"cba"`.

O fluxo de codificação é contínuo, e todos os símbolos da mensagem são codificados de forma conjunta. Isso resulta em um código dinâmico, particularmente eficiente para fontes com distribuições probabilísticas variáveis.

Figura 5 – Codificação da palavra `"cba"` utilizando codificação aritmética



Fonte: Elaborada pelo autor

O processo inicia com o intervalo inicial $[low, high[= [0, 1[$. A cada símbolo processado, esse intervalo é subdividido proporcionalmente às probabilidades acumuladas dos símbolos.

Formalmente, para cada símbolo s correspondente ao índice j, o subintervalo é calculado por

$$[\, low + (high - low)c_{j-1},\ low + (high - low)c_j \,[\,,$$

garantindo que a largura do subintervalo resultante seja exatamente a probabilidade associada ao símbolo.

Aplicando esses cálculos ao exemplo, os subintervalos a cada etapa ficam definidos

da seguinte maneira:

1. **Símbolo `"c"`**:

$$[\,low, high\,[\ =\ [\,c_2,\ c_2 + p_2\,[\ =\ [\,0{,}6;\ 1\,[\,.$$

2. **Símbolo `"b"`** (novo intervalo sobre o intervalo anterior):

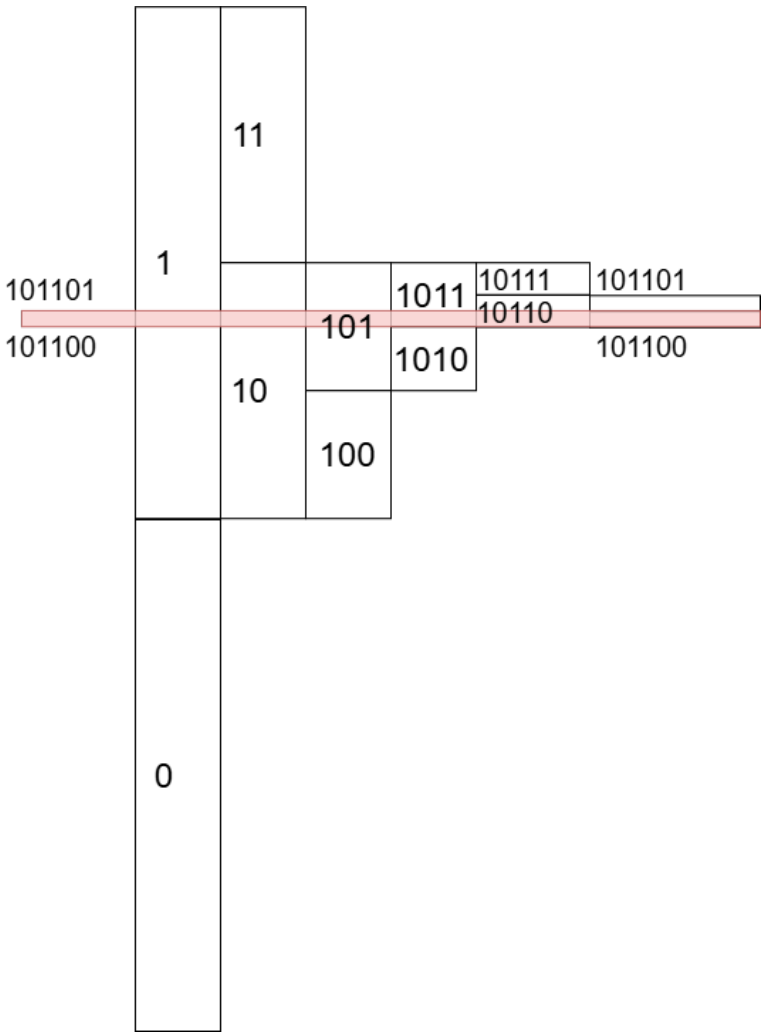
$$[\,low, high\,[= [\,0{,}6 + (1 - 0{,}6) \cdot c_1,\ 0{,}6 + (1 - 0{,}6) \cdot (c_1 + p_1)\,[= [\,0{,}68;\ 0{,}84\,[\,.$$

3. **Símbolo `"a"`** (novo intervalo sobre o intervalo anterior):

$$[\,low, high\,[= [\,0{,}68 + (0{,}84 - 0{,}68) \cdot c_0,\ 0{,}68 + (0{,}84 - 0{,}68) \cdot (c_0 + p_0)\,[= [\,0{,}68;\ 0{,}712\,[\,.$$

Esse intervalo final representa a sequência completa codificada.

Figura 6 – Codificação da palavra `"cba"` em bits



Fonte: Elaborada pelo autor

No exemplo em questão, o intervalo final de codificação é $[0{,}68; 0{,}712[$. A árvore de subdivisões mostra:

- A primeira divisão $\{0, 1\}$ separa $[0; 0{,}5[$ e $[0{,}5; 1[$. Como o nosso intervalo está em $[0{,}68; 0{,}712[$, ele pertence à segunda metade, logo o primeiro bit é `"1"`.
- Em seguida, subdivide-se $[0{,}5, 1[$ em $[0{,}5; 0{,}75[$ (prefixo `"10"`) e $[0{,}75; 1[$ (prefixo `"11"`). Visto que ainda estamos em $[0{,}68; 0{,}712[\subset [0{,}5; 0{,}75[$, o segundo bit é `"0"`, formando o prefixo `"10"`.
- O processo continua, refinando o intervalo `"10"` em `"100"` vs. `"101"` e assim por diante, até que o prefixo binário completo caia estritamente dentro de $[0{,}68; 0{,}712[$.

Observando o diagrama, chegamos ao prefixo

$$\texttt{0.101100}_2$$

que define o subintervalo $[\texttt{0.101100}_2, \texttt{0.101101}_2[\; = \; [0{,}6875; 0{,}703125[$, inteiramente contido em $[0{,}68; 0{,}712[$. Assim, a sequência `"101100"` é a codificação em bits de precisão mínima que captura fielmente o valor fracionário representativo da mensagem original usando precisão efetivamente infinita.

Para converter o intervalo final em uma sequência de bits, considera-se inicialmente a representação infinita do espaço $[0, 1[$ em subdivisões binárias. Cada dígito binário (`"0"` ou `"1"`) corresponde a uma metade do intervalo atual, conforme ilustra o diagrama da Figura 6. A codificação prossegue extraindo o menor prefixo binário que, como número real, cai totalmente dentro do intervalo final calculado para a sequência `"cba"`.

2.2.2 Algoritmo com Precisão Finita e Redimensionamento

A análise realizada anteriormente considera que o algoritmo aritmético dispõe de precisão infinita nos cálculos numéricos. Porém, em implementações práticas, é necessário adaptar o algoritmo para trabalhar com precisão finita, já que computadores apresentam limitações na precisão dos números de ponto flutuante.

Essa adaptação consiste na aplicação periódica de uma operação chamada de *redimensionamento* (*rescaling*), cujo objetivo é evitar que o intervalo codificado torne-se pequeno demais, causando perda numérica de precisão. O procedimento de redimensionamento é ilustrado na Figura 7.

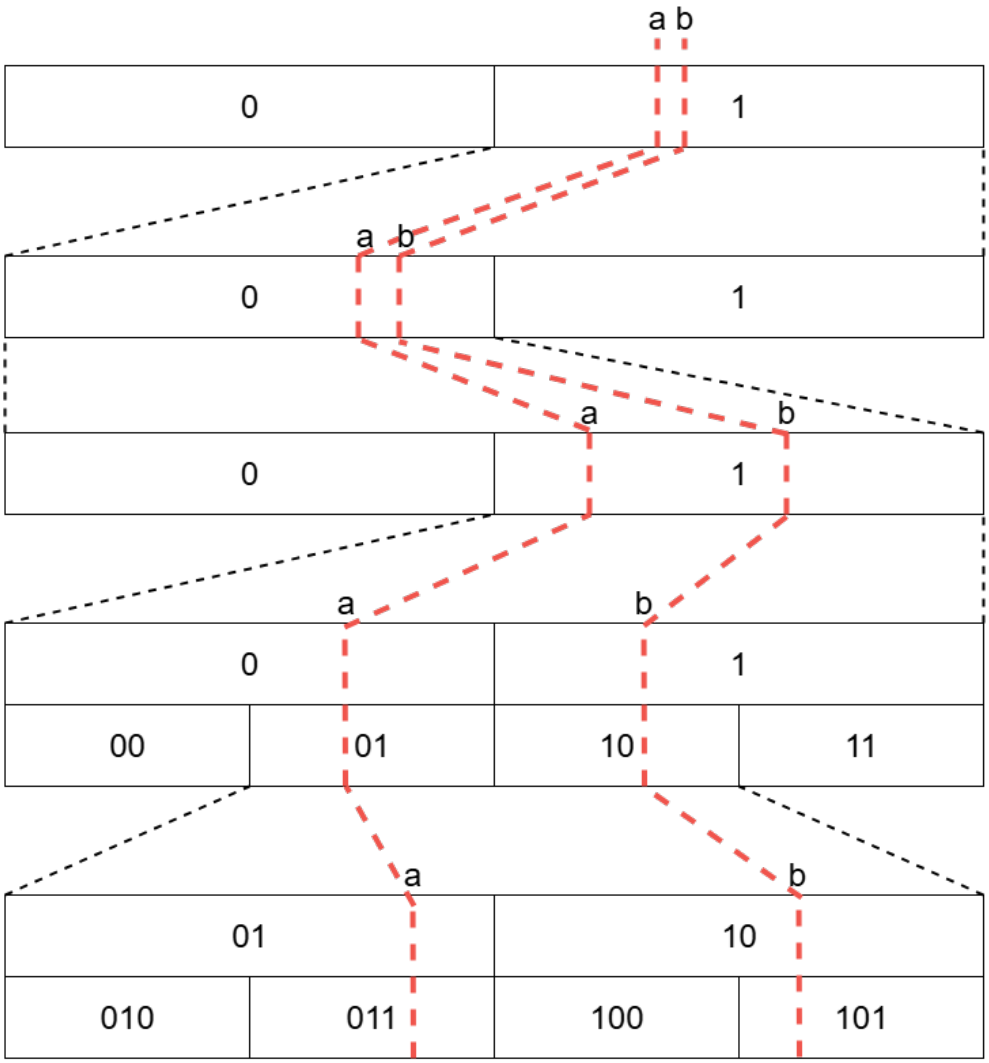
Esse procedimento pode ser descrito pelos seguintes critérios básicos:

1. **Caso $b < \frac{1}{2}$:** Quando todo o intervalo atual estiver na primeira metade do intervalo $[0{,}1[$, ou seja, $[a, b[\subset [0, \frac{1}{2}[$, emite-se o bit '0' e realiza-se o escalonamento:

$$a \leftarrow 2 \cdot a, \quad b \leftarrow 2 \cdot b$$

conforme Figura 8.

Figura 7 – Operação geral de redimensionamento na codificação aritmética



Fonte: Elaborada pelo autor

2. **Caso $a \geq \frac{1}{2}$:** Quando todo o intervalo atual estiver na segunda metade do intervalo, $[a, b[\subset [\frac{1}{2}, 1[$, emite-se o bit '1' e efetua-se o redimensionamento do intervalo:

$$a \leftarrow 2 \cdot \left(a - \frac{1}{2}\right), \quad b \leftarrow 2 \cdot \left(b - \frac{1}{2}\right)$$

ilustrado na Figura 9.

3. **Caso $a \geq \frac{1}{4}$ e $b < \frac{3}{4}$:** Quando o intervalo codificado situa-se na metade central do intervalo unitário, ou seja, entre $\frac{1}{4}$ e $\frac{3}{4}$, é necessário utilizar um contador s, inicialmente igual a zero, que indica a quantidade de redimensionamentos consecutivos ocorridos nesse caso intermediário. Aplica-se o escalonamento:

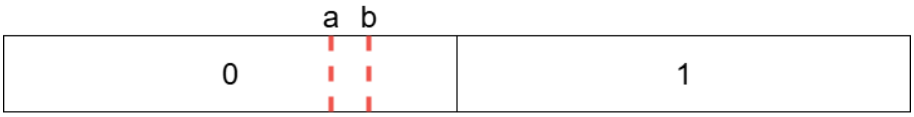
$$a \leftarrow 2 \cdot \left(a - \frac{1}{4}\right), \quad b \leftarrow 2 \cdot \left(b - \frac{1}{4}\right), \quad s \leftarrow s + 1$$

Essa operação é ilustrada na Figura 10. O contador s registra a quantidade de bits ainda pendentes a serem emitidos.

Quando finalmente o intervalo migrar para uma das metades externas do intervalo, será emitido o bit correspondente (‘0’ ou ‘1’), seguido por s bits adicionais complementares (se o bit emitido for ‘0’, os s bits seguintes serão ‘1’, e vice-versa). Após a emissão, o contador s é zerado.

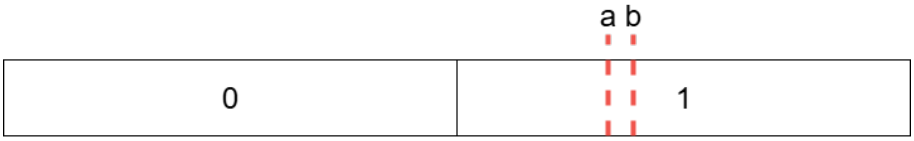
As Figuras 8, 9 e 10 exemplificam detalhadamente esses critérios.

Figura 8 – Redimensionamento quando $b < \frac{1}{2}$



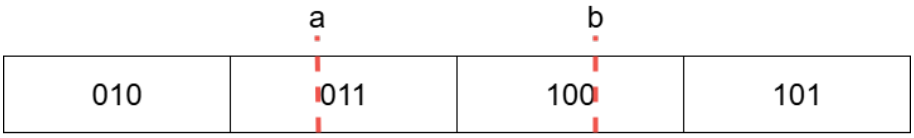
Fonte: Elaborada pelo autor

Figura 9 – Redimensionamento quando $a \geq \frac{1}{2}$



Fonte: Elaborada pelo autor

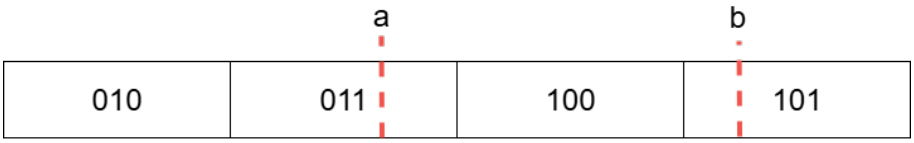
Figura 10 – Redimensionamento quando o intervalo está na metade central $\frac{1}{4} \leq a < b < \frac{3}{4}$



Fonte: Elaborada pelo autor

Além desses critérios básicos, também podem ocorrer subdivisões adicionais específicas, mostradas nas Figuras 11 e 12, onde são detalhadas subdivisões específicas adicionais.

Figura 11 – Subdivisão específica em $\frac{3}{4}$



Fonte: Elaborada pelo autor

Para fazer esta implementação da codificação aritmética com *precisão finita*, define-se inicialmente a *precisão* como o número de bits utilizados para representar os valores numéricos. Por exemplo, para uma precisão de $p = 32$ bits, são definidos três valores auxiliares:

$$\text{Todo} = 2^p, \quad \text{Metade} = \frac{\text{Todo}}{2}, \quad \text{quarto} = \frac{\text{Todo}}{4}.$$

Figura 12 – Subdivisão específica em $\frac{1}{4}$

Fonte: Elaborada pelo autor

Seja o alfabeto de símbolos:

$$\mathcal{X} = \{0, 1, \ldots, n\}, \quad \text{com um símbolo especial EoF} = 0.$$

O vetor de probabilidades:

$$p = (p_0, p_1, \ldots, p_n), \quad p_i = \frac{f_i}{\sum_{i=0}^{n} f_i},$$

onde f_i representa a frequência observada para o símbolo i.

Para inicialização, é necessário representar a variável R (*range*) como um valor inteiro compatível com a precisão estabelecida. Seja a sequência de entrada $x_1, x_2, \ldots, x_k \in \mathcal{X} \setminus \{0\}$, com o símbolo $x_{k+1} = 0$ representando o EoF.

Define-se:

$$C_0 = 0, \quad C_j = r_0 + r_1 + \cdots + r_j, \quad \text{para } j = 1, \ldots, n,$$

onde r_j representa o número acumulado de ocorrências (ou frequência acumulada) para os símbolos.

O vetor de deslocamento é dado por:

$$d_j = c_j + r_j, \quad \text{para } j = 0, \ldots, n.$$

Essas definições garantem que os intervalos de codificação e decodificação possam ser calculados corretamente utilizando aritmética inteira, preservando a precisão numérica durante todo o processo.

2.3 BIBLIOTECA KOMM

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

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

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

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

3 DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento dos módulos LZ77 e codificação aritmética na biblioteca *Komm*. A implementação foi realizada em Python, linguagem-base do projeto.

3.1 MÓDULO LZ77

Esta seção documenta a implementação do algoritmo *Lempel–Ziv 77* (LZ77) na *Komm*, cobrindo decisões de projeto, integração e validação. A biblioteca já inclui diversos esquemas de compressão sem perdas (Huffman, Tunstall, LZ78 e LZW), e a contribuição aqui foi a implementação completa do LZ77 com foco tanto prático quanto didático.

3.1.1 Objetivos e escopo

Os objetivos deste módulo foram:

- implementar as rotinas de codificação (`encode`) e decodificação (`decode`) gerando **tokens** no formato (p, ℓ, x);
- integrar a classe ao padrão já utilizado, documentação e testes da *Komm*;
- disponibilizar uma **pipeline modular** para fins educacionais, viabilizando a inspeção de resultados intermediários e a comparação com exemplos da literatura (que, em geral, não mostram a saída em base binária).

3.1.2 Janela deslizante e notação

Adotamos a notação usadas nas bibliografias que foram utilizadas para o estudo do algoritmo LZ77:

$$W = S + L,$$

onde S é o tamanho do *search buffer*, L é o tamanho do *lookahead buffer* e W é o tamanho total da janela. Denotamos por $|\mathcal{X}|$ a cardinalidade do alfabeto de **entrada** e por $|\mathcal{Y}|$ a cardinalidade do alfabeto de **saída**.

Cada token tem o formato

$$(p, \ell, x),$$

em que p é o *pointer* (deslocamento no *search buffer*), ℓ é o comprimento da ocorrência e $x \in \mathcal{X}$ é o símbolo literal subsequente.

Origem do ponteiro.

Neste trabalho, seguindo o artigo original do LZ77, p é medido a partir do **início** do *search buffer*. *Observação*: algumas implementações didáticas medem a partir do **fim** do *search buffer*; ambas convenções aparecem na literatura. Mantemos a primeira e documentamos a diferença quando necessário.

3.1.3 Arquitetura e estrutura da implementação

A classe `LempelZiv77Code` concentra a lógica do algoritmo e expõe a classe da seguinte maneira:

Código 3.1 – Estrutura simplificada da classe LZ77

```
@dataclass
class LempelZiv77Code:
    source_cardinality: int      # |X|
    target_cardinality: int      # |Y|
    window_size: int             # W
    lookahead_size: int          # L (logo S = W - L)

    def encode(self, data): ...
    def decode(self, encoded): ...
```

Exemplo de uso:

Código 3.2 – Uso típico do LZ77 na *Komm*

```
lz77 = LempelZiv77Code(
    source_cardinality=256,     # |X|
    target_cardinality=2,       # |Y| (binário)
    window_size=32768,          # W
    lookahead_size=258          # L (então S = 32510)
)
encoded = lz77.encode(b"... dados ...")
decoded = lz77.decode(encoded)
assert decoded == b"... dados ..."
```

Com intuito pedagógico, a pipeline foi dividida em **quatro** módulos internos — dois para `encode` e dois para `decode`:

1. **Codificação**
   - `source_to_tokens`: identifica os tokens (p, ℓ, x) sobre a janela $W = S + L$;
   - `tokens_to_target`: mapeia os tokens para o fluxo no alfabeto de saída $\mathcal{Y}$.
2. **Decodificação**

- `target_to_tokens`: reconstrói os tokens a partir do fluxo em $\mathcal{Y}$;
- `tokens_to_source`: recompõe a sequência original em $\mathcal{X}$.

Essa organização favorece: (i) clareza didática; (ii) comparação direta com esquemas de manuais/artigos; e (iii) inspeção de saídas intermediárias (inclusive em binário), recurso raramente mostrado na literatura.

3.1.4 Larguras de campos na saída

Seja a saída codificada em base $|\mathcal{Y}|$. As larguras (em símbolos do alfabeto de saída) dos campos do token são:

$$w_p = \left\lceil \log_{|\mathcal{Y}|} S \right\rceil, \qquad w_\ell = \left\lceil \log_{|\mathcal{Y}|} L \right\rceil, \qquad w_x = \left\lceil \log_{|\mathcal{Y}|} |\mathcal{X}| \right\rceil,$$

e o custo total por token é

$$n \;=\; w_p + w_\ell + w_x.$$

3.1.5 Testes e validação

Os testes unitários (`test_lz77.py`) cobrem:

- cadeias com alta redundância e com sobreposição;
- exemplos reproduzidos da literatura.

A validação principal usa *round trip* (`decode`(`encode`(x)) $= x$) e compara tokens/fluxos intermediários com os parâmetros (W, S, L) e larguras (w_p, w_ℓ, w_x).

4 RESULTADOS

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

4.1 CONJUNTOS DE DADOS E PROTOCOLO EXPERIMENTAL

Foram utilizados dois tipos de arquivos: (i) **textos** e (ii) **imagens**. O texto escolhido foi o livro *Alice's Adventures in Wonderland*, do Projeto 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*.

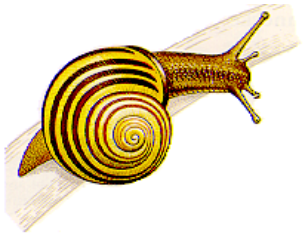
As imagens utilizadas foram bitmaps (**BMP**) simples, escolhidas por possuírem regiões de baixa entropia e padrões espaciais redundantes, que permitem observar o comportamento do algoritmo em fontes bidimensionais. A Figura 13 mostra a imagem *smiley* (246 bytes) e a Figura 14 a imagem *snail* (196.666 bytes).

Figura 13 – Imagem bitmap (BMP) *smiley*.

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

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

Figura 14 – Imagem bitmap (BMP) *snail*.



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

4.2 MÉTRICAS DE AVALIAÇÃO

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

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

4.3 RESULTADOS DO LZ77 NA *KOMM*

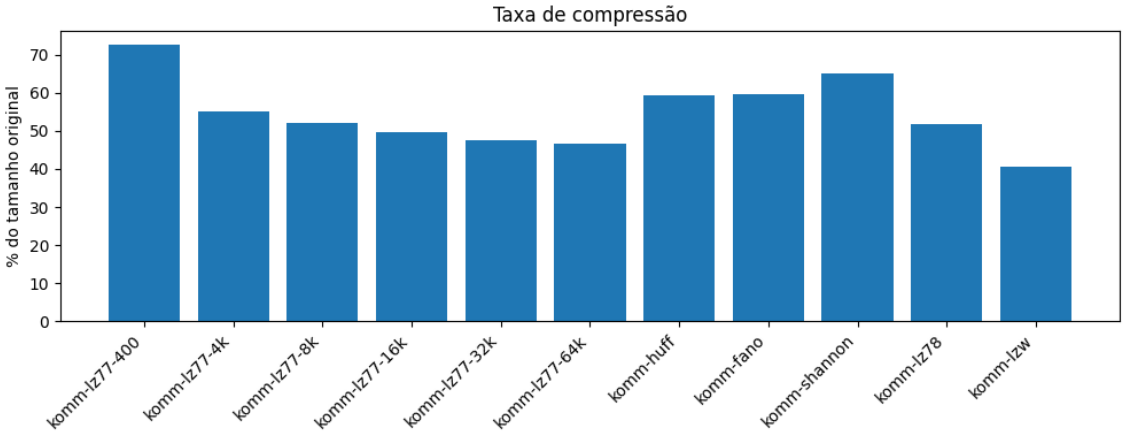
Foram consideradas duas implementações populares: **FastLZ**[2] (C) e **LZ77-Compressor**[3] (Python). As configurações efetivas foram: $W = 8\,\text{KiB}$ e $L = 264$ bytes no *FastLZ*; no *LZ77-Compressor*, W variou entre 100 e 400 bytes e $L = 15$ bytes. Na *Komm*, W e L são parametrizáveis; para comparação, utilizaram-se configurações próximas às disponíveis nas bibliotecas externas quando pertinente, além de uma configuração de referência com $W = 64\,\text{kB}$.

Análise (texto):

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

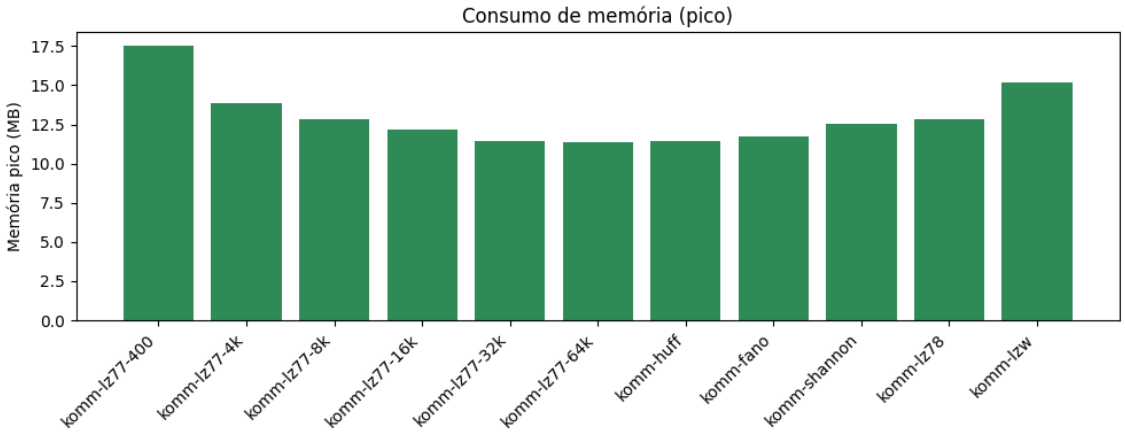
[2] https://github.com/ariya/FastLZ
[3] https://github.com/ariya/FastLZ

Figura 15 – Texto *Alice*: taxa de compressão (menor é melhor).



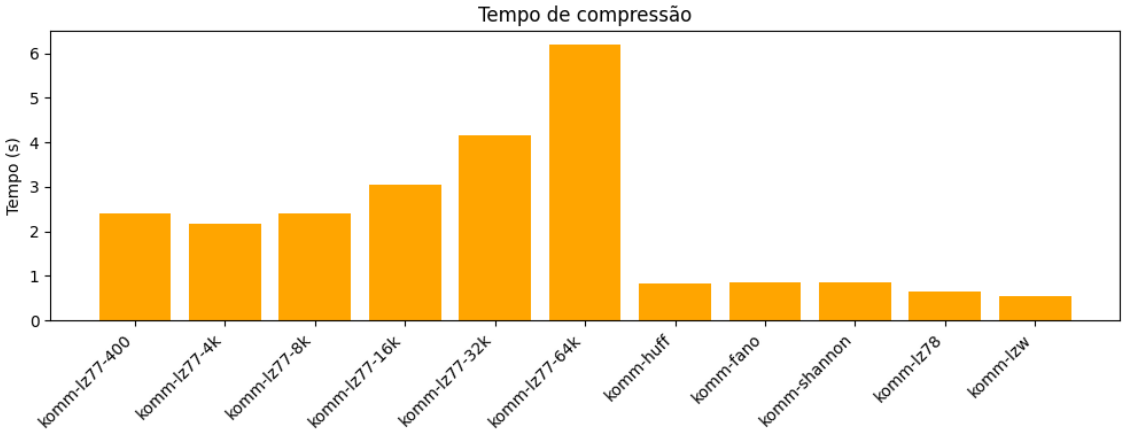
Fonte: Elaborada pelo autor.

Figura 16 – Texto *Alice*: memória pico.



Fonte: Elaborada pelo autor.

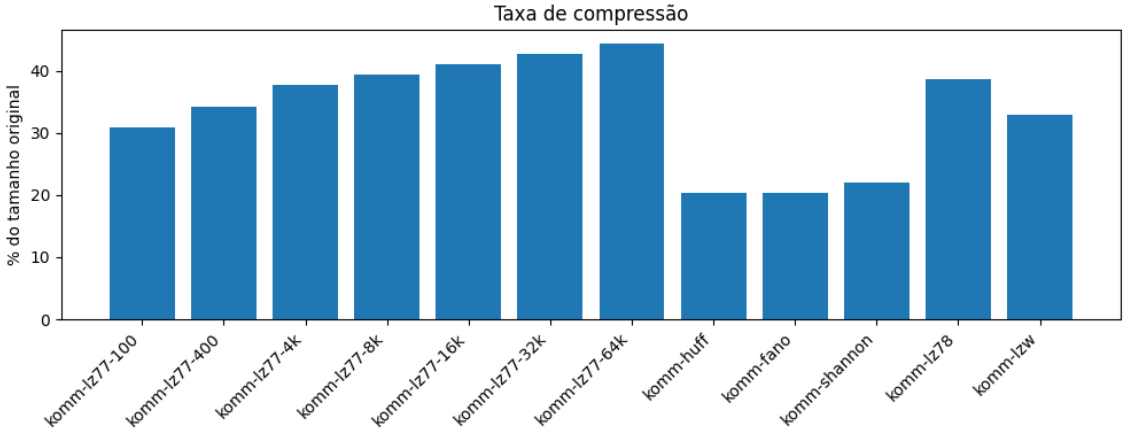
Figura 17 – Texto *Alice*: tempo de compressão.



Fonte: Elaborada pelo autor.

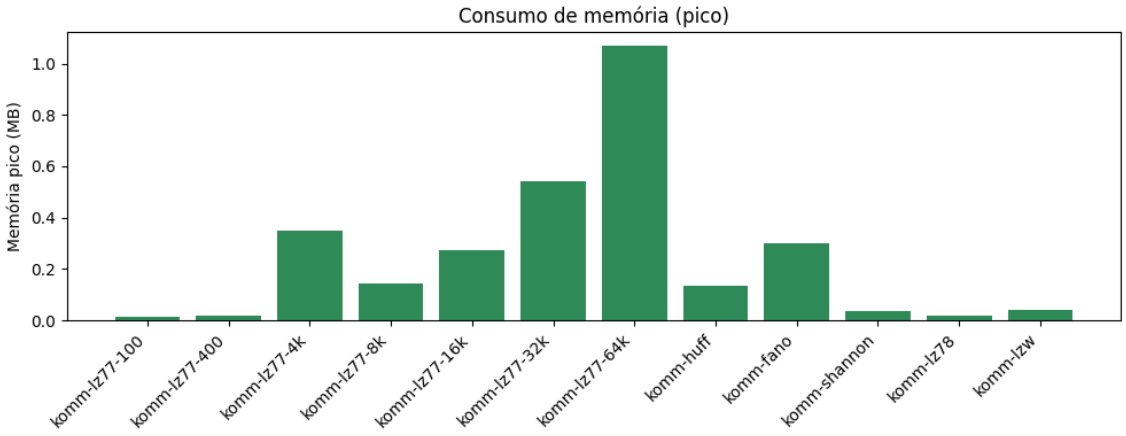
- Em comparação, Huffman e Shannon–Fano mantêm desempenho rápido e baixo uso de memória, porém com taxas de compressão inferiores.

Figura 18 – Imagem *smiley*: taxa de compressão (menor é melhor).



Fonte: Elaborada pelo autor.

Figura 19 – Imagem *smiley*: memória pico.

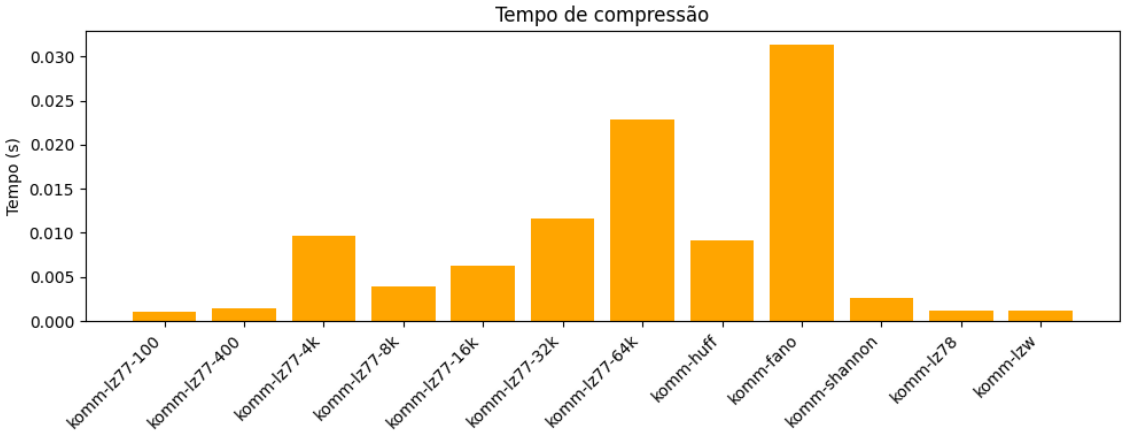


Fonte: Elaborada pelo autor.

Análise (imagens):

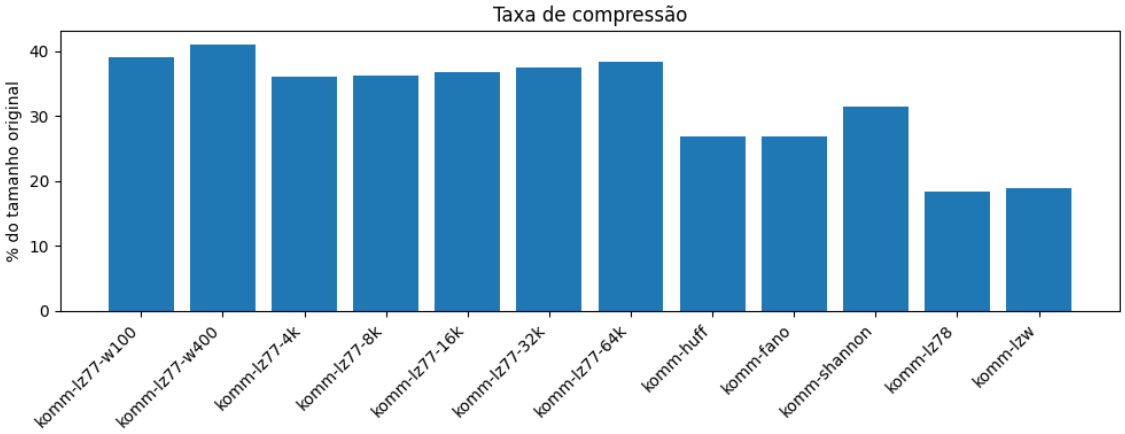
- Nas imagens *smiley* e *snail*, o *LZ77* apresentou menor ganho de compressão em comparação ao texto, o que é esperado dada a menor redundância sequencial.
- Janelas maiores continuam a reduzir ligeiramente a taxa de compressão, porém com crescimento do custo computacional.
- A relação entre tempo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

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



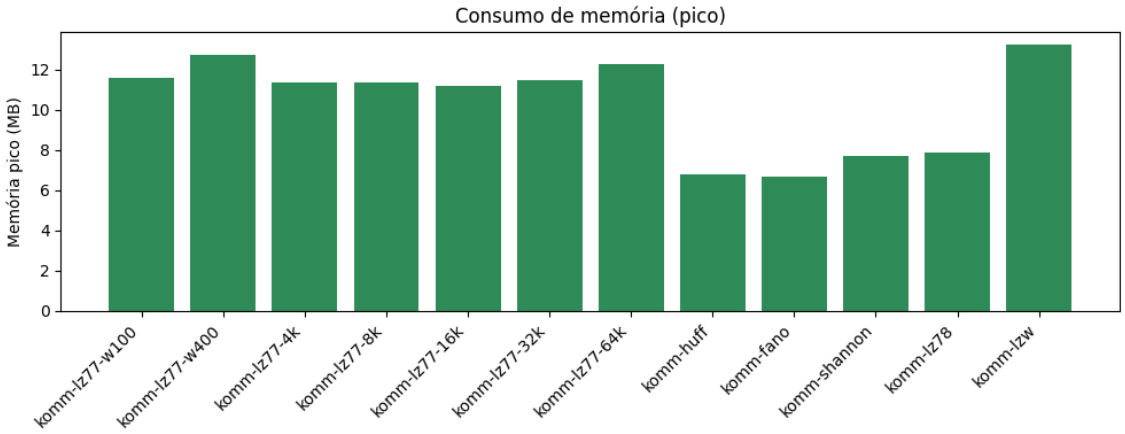
Fonte: Elaborada pelo autor.

Figura 21 – Imagem *snail*: taxa de compressão (menor é melhor).



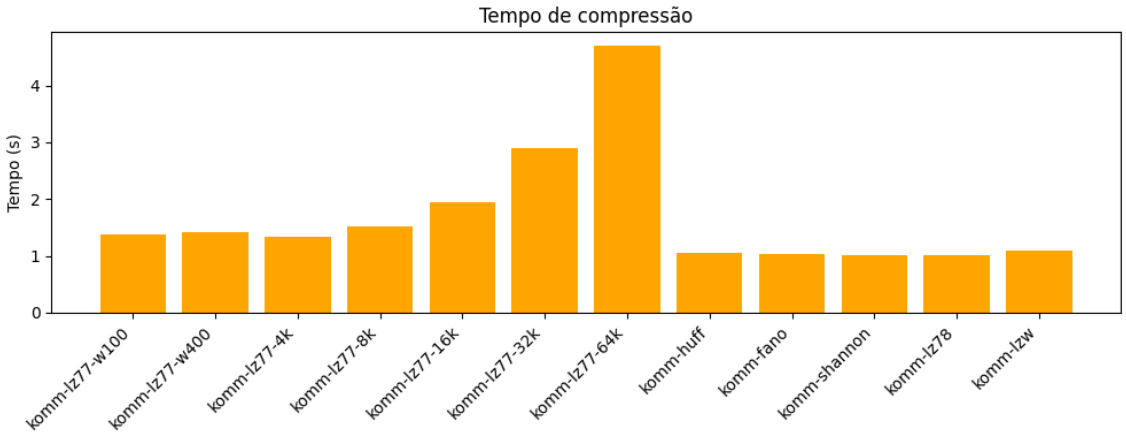
Fonte: Elaborada pelo autor.

Figura 22 – Imagem *snail*: memória pico.



Fonte: Elaborada pelo autor.

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



Fonte: Elaborada pelo autor.

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

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

Fonte: Elaborada pelo autor.

4.4 COMPARAÇÃO COM IMPLEMENTAÇÕES EXTERNAS DE LZ77

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

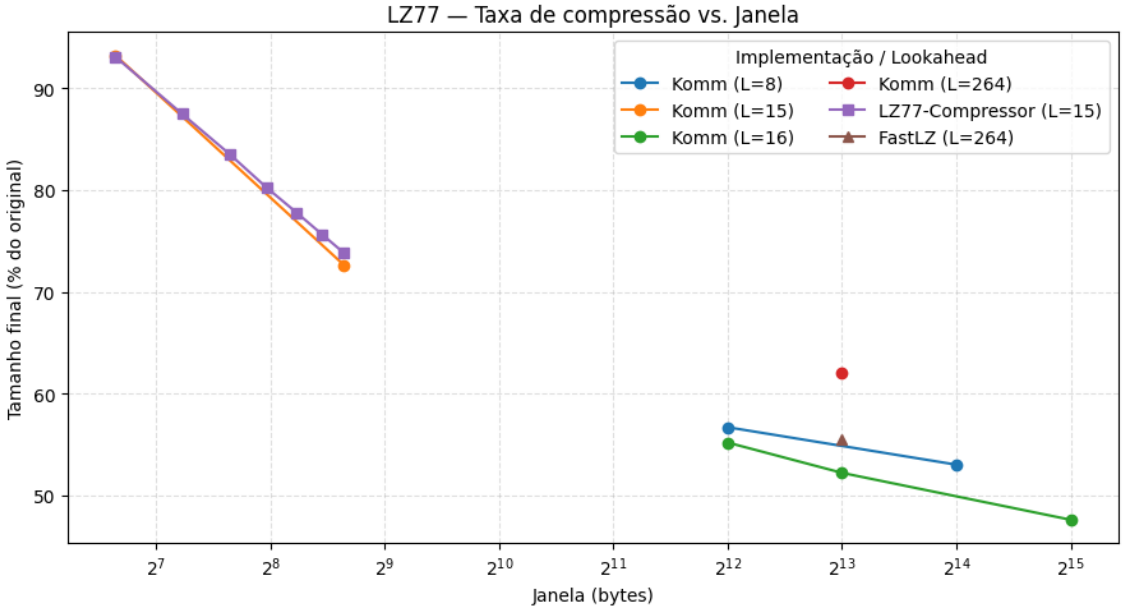
- **FastLZ**: janela de 8 kB e *lookahead* de 264 bytes;
- **LZ77-Compressor**: janela variável (100–400 bytes) e *lookahead* fixo de 15 bytes.

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

[4] https://github.com/ariya/FastLZ
[5] https://github.com/manassra/LZ77-Compressor

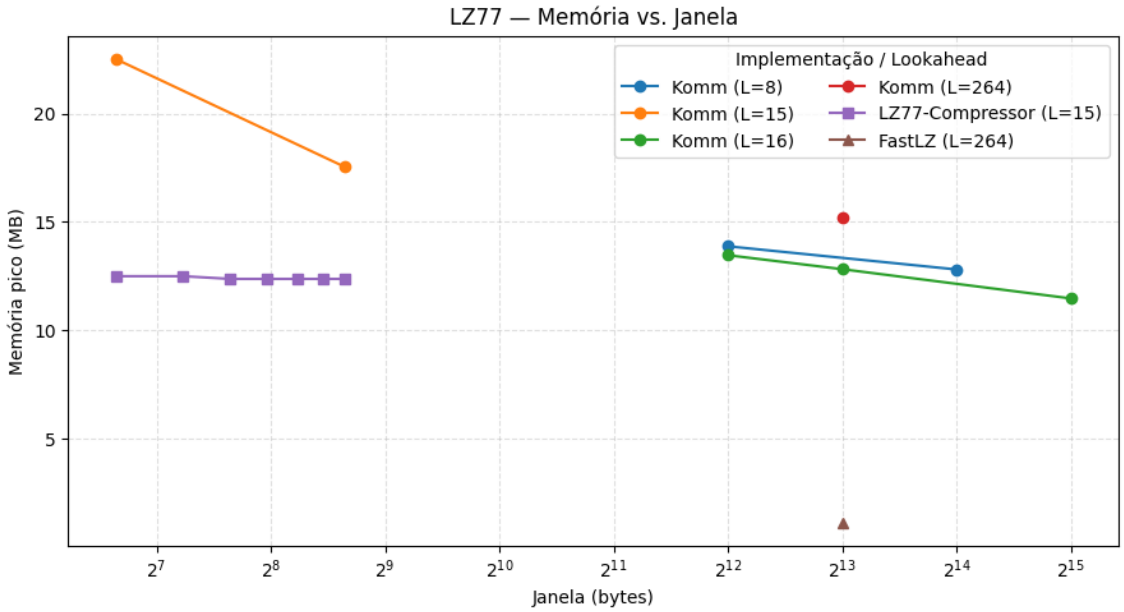
$W = 64\,\text{kB}$.

Figura 24 – Texto *Alice*: taxa de compressão (menor é melhor).



Fonte: Elaborada pelo autor.

Figura 25 – Texto *Alice*: memória pico.

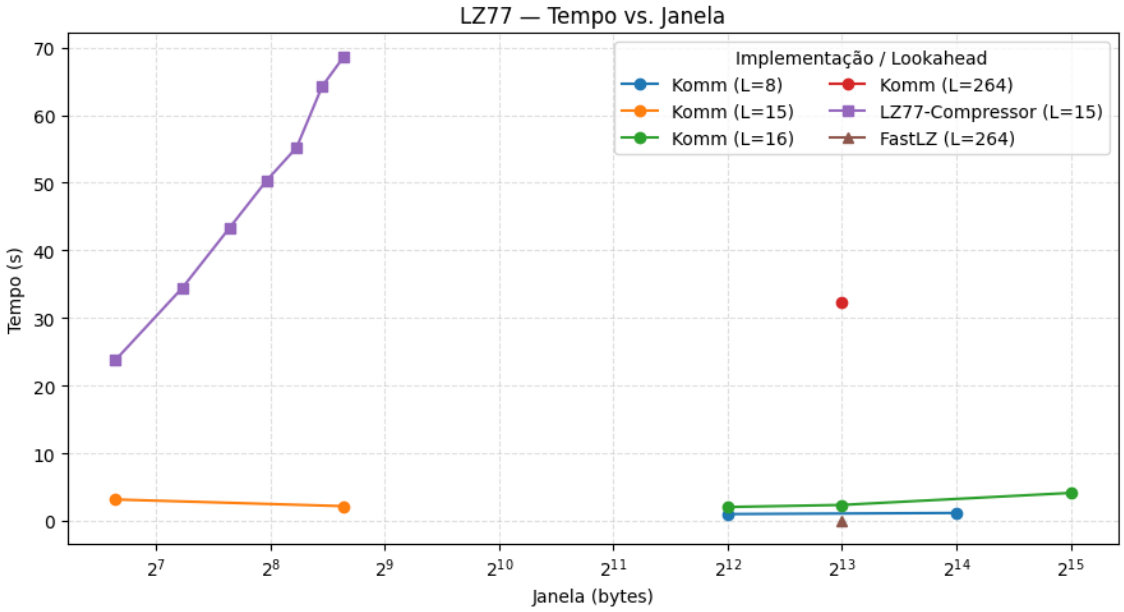


Fonte: Elaborada pelo autor.

Análise (texto):

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

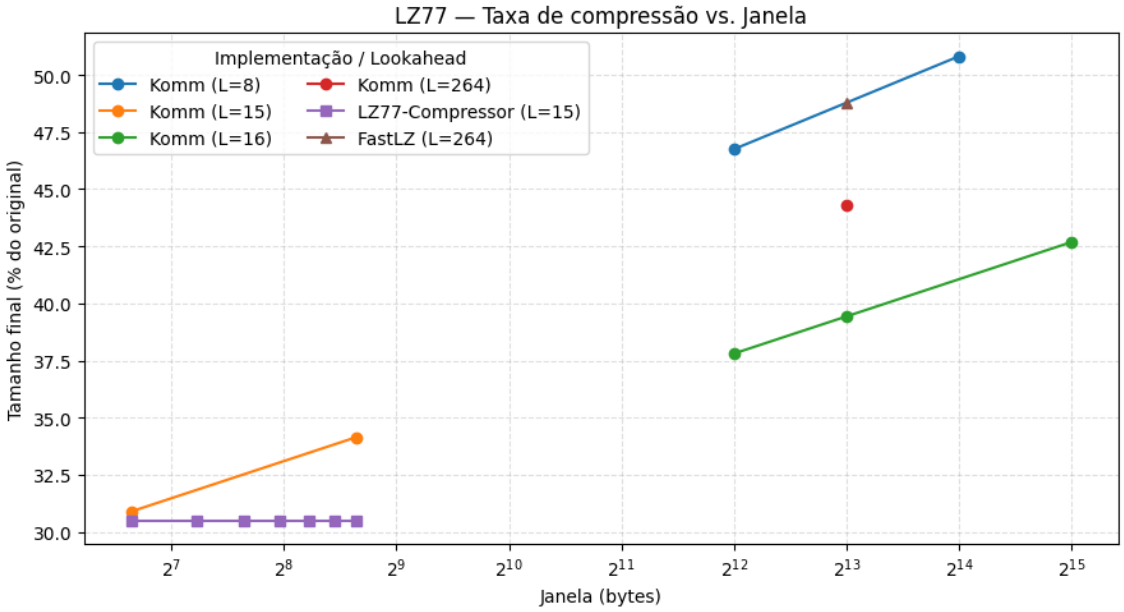
Figura 26 – Texto *Alice*: tempo de compressão.



Fonte: Elaborada pelo autor.

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

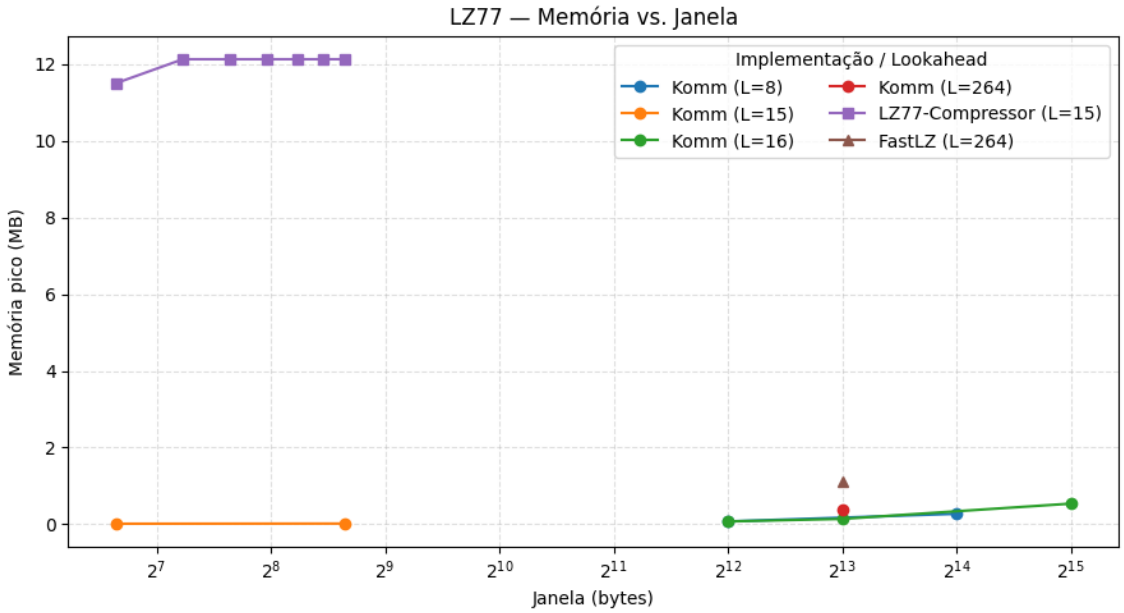
Figura 27 – Imagem *smiley*: taxa de compressão (menor é melhor).



Fonte: Elaborada pelo autor.

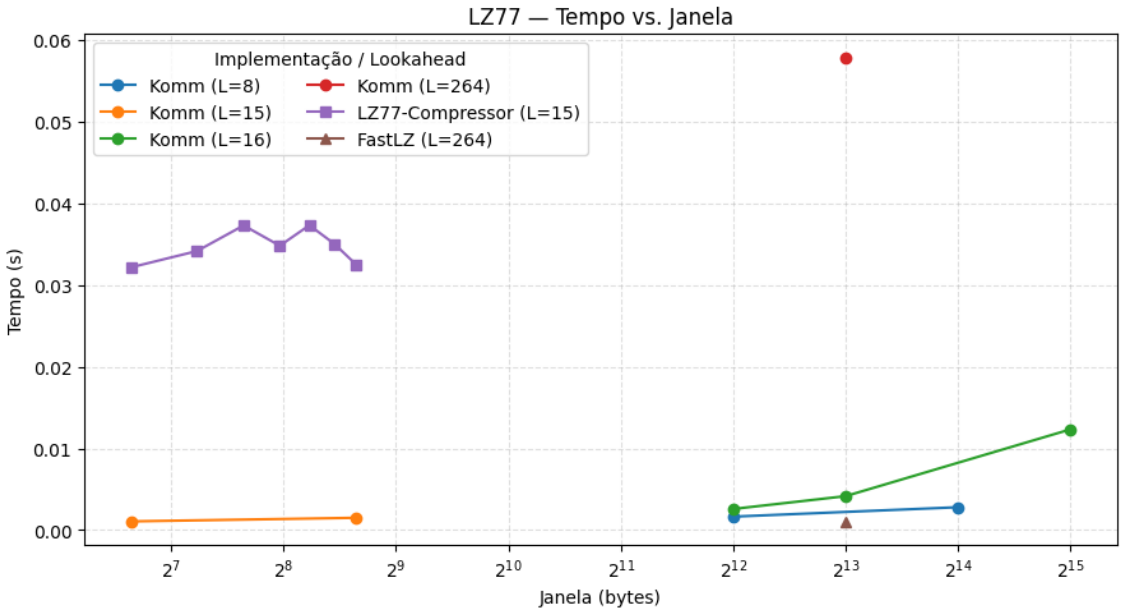
Análise (imagens):

Figura 28 – Imagem *smiley*: memória pico.



Fonte: Elaborada pelo autor.

Figura 29 – Imagem *smiley*: tempo de compressão.



Fonte: Elaborada pelo autor.

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

4.5 DISCUSSÃO

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

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

REFERÊNCIAS

DEUTSCH, L. Peter. **DEFLATE Compressed Data Format Specification version 1.3**. RFC Editor, mai. 1996. 17 p. RFC 1951. (Request for Comments, 1951). DOI: 10.17487/RFC1951. Disponível em: <https://www.rfc-editor.org/info/rfc1951>.

MACKAY, D.J.C. **Information Theory, Inference and Learning Algorithms**. Cambridge University Press, 2003. ISBN 9780521642989. Disponível em: <https://books.google.com.br/books?id=AKuMj4PN_EMC>.

NELSON, M. **The Data Compression Book**. BPB Publications, 2008. ISBN 9788170297291. Disponível em: <https://books.google.com.br/books?id=pndwnAEACAAJ>.

PASCO, Richard Clark. **Source coding algorithms for fast data compression.** 1976. Tese (Doutorado) – Stanford University, Stanford, CA, USA. AAI7626055.

RISSANEN, J. J. Generalized Kraft Inequality and Arithmetic Coding. **IBM Journal of Research and Development**, v. 20, n. 3, p. 198–203, 1976. DOI: 10.1147/rd.203.0198.

SAYOOD, K. **Introduction to Data Compression**. Elsevier Science, 2012. (The Morgan Kaufmann Series in Multimedia Information and Systems). ISBN 9780124157965. Disponível em: <https://books.google.com.br/books?id=mkCMxnHm6hsC>.

ZIV, J.; LEMPEL, A. A universal algorithm for sequential data compression. **IEEE Transactions on Information Theory**, v. 23, n. 3, p. 337–343, 1977. DOI: 10.1109/TIT.1977.1055714.

APÊNDICE A – MEU PRIMEIRO APÊNDICE

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

ANEXO A – MEU PRIMEIRO ASSUNTO DE ANEXO

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

ANEXO B – SEGUNDO ASSUNTO QUE PESQUISEI

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

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

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