UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO YUJII SILVA KADOOKA

EXPLORANDO ALGORITMOS DE COMPRESSÃO DE DADOS: TEORIA, IMPLEMENTAÇÃO E DESEMPENHO

BAURU
Novembro/2025

GUSTAVO YUJII SILVA KADOOKA

EXPLORANDO ALGORITMOS DE COMPRESSÃO DE DADOS: TEORIA, IMPLEMENTAÇÃO E DESEMPENHO

Trabalho de Conclusão de Curso em Ciência da Computação apresentado ao Departamento de Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, Campus Bauru.
Orientador: Profa. Dra. Andréa Carla Gonçalves Vianna

BAURU
Novembro/2025

K11e

Kadooka, Gustavo Yujii Silva

Explorando Algoritmos de Compressão de Dados: Teoria, Implementação e Desempenho / Gustavo Yujii Silva Kadooka. -- Bauru, 2025

96 p. : il., tabs., fotos

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (UNESP), Faculdade de Ciências, Bauru

Orientadora: Andréa Carla Gonçalves Vianna

1. Compressão de dados (Computação). 2. Teoria da informação. 3. Compressão sem perdas – Huffman, LZ77, LZW, GZIP. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Dados fornecidos pelo autor.

Gustavo Yujii Silva Kadooka

Explorando Algoritmos de Compressão de Dados: Teoria, Implementação e Desempenho

Trabalho de Conclusão de Curso em Ciência da Computação apresentado ao Departamento de Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Profa. Dra. Andréa Carla Gonçalves Vianna
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Profa. Dra. Simone das Graças Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Profa. Dra. Juliana da Costa Feitosa
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Bauru, 12 de novembro de 2025.

À minha amada família, fonte inesgotável de inspiração e força.

Agradecimentos

Agradeço à Universidade Estadual Paulista “Júlio de Mesquita Filho” – Faculdade de Ciências, Campus de Bauru –, por ter sido o espaço onde pude me desenvolver como estudante, profissional e indivíduo. Sou grato a todos os professores e funcionários da instituição pelo conhecimento compartilhado e pelo apoio ao longo da graduação.

À minha orientadora, Profa. Dra. Andréa Carla Gonçalves Vianna, pela disponibilidade, paciência diante das dificuldades e por acreditar no meu potencial durante o desenvolvimento deste trabalho.

Aos meus colegas de curso, que estiveram presentes ao longo desta jornada, dividindo aprendizados, desafios e conquistas. O convívio e a troca de experiências tornaram a caminhada mais leve e significativa.

À minha irmã Gabriely, pelas palavras de incentivo e carinho em todos os momentos, mesmo nos mais difíceis.

E, com o meu mais profundo e sincero agradecimento, menciono especialmente meus pais – Flávia e Silvio –, por todo o apoio emocional, pelos ensinamentos de vida, pela confiança depositada em mim e por sempre me encorajarem a seguir em frente. Esta conquista é, acima de tudo, de vocês.

Resumo

Este trabalho apresenta um estudo abrangente, a implementação prática e a análise de desempenho de algoritmos clássicos de compressão de dados, a saber: Huffman, LZ77, LZW e GZIP. São explorados os conceitos teóricos fundamentais de cada algoritmo, seguidos pela implementação prática utilizando a linguagem de programação C++. As implementações dos algoritmos lidam com os formatos de arquivo de texto simples (.txt), imagens bitmap (.bmp) e arquivos de áudio PCM (.wav), demonstrando versatilidade e aplicabilidade em diferentes domínios de dados. Para facilitar a interação do usuário, foi desenvolvida uma interface gráfica utilizando a biblioteca GTK, permitindo a seleção dos algoritmos de compressão e o arquivo de entrada. Ademais, para realizar uma análise comparativa rigorosa, foram utilizados scripts em Python para processar os dados experimentais e gerar representações gráficas dos principais indicadores de desempenho, como a taxa de compressão e o tempo de execução para diferentes tipos de arquivos. Os resultados experimentais indicam que não há um algoritmo que se destaque universalmente, pois cada um apresenta vantagens específicas conforme o tipo de arquivo e as características dos dados. Isso ressalta a importância da escolha do método de compressão adequado às necessidades específicas da aplicação. O projeto contribui tanto com uma ferramenta prática para compressão de dados quanto com um recurso educacional que auxilia na compreensão dos algoritmos clássicos e seu comportamento em cenários reais.

Palavras-chave: Compressão de dados. Algoritmos de compressão. Huffman. LZ77. LZW. GZIP. C++. Python. GTK.

Abstract

This work presents a comprehensive study, implementation, and performance analysis of classical data compression algorithms, namely Huffman, LZ77, LZW, and GZIP. The theoretical concepts behind each algorithm are explored in depth, followed by practical implementations using C++ programming language. The implementations support plain text (.txt), bitmap images (.bmp), and PCM audio files (.wav), demonstrating versatility and applicability in different data domains. To facilitate user interaction, a graphical user interface (GUI) was developed with the GTK library, enabling selection of compression algorithms, and input files. Furthermore, to conduct a rigorous comparative analysis, Python scripts were employed to process experimental data and generate graphical representations of key performance metrics such as compression ratio and execution time for different file types. The experimental results indicate that no single algorithm universally outperforms others; rather, each one exhibits specific advantages depending on the file type and data characteristics. This highlights the importance of choosing the appropriate compression method tailored to the application's needs. The project contributes as a practical tool for data compression and an educational resource that aids in understanding fundamental compression algorithms and their behavior in real-world scenarios.

Keywords: Data compression, compression algorithms, Huffman coding, LZ77, LZW, GZIP, C++, Python, GTK.

Lista de figuras

Figura 1 – Entropia de uma moeda em função de p 23
Figura 2 – Exemplo de uma Árvore de Huffman 29
Figura 3 – Etapa 0 da construção da árvore de Huffman 29
Figura 4 – Etapa 1 da construção da árvore de Huffman 30
Figura 5 – Etapa 2 da construção da árvore de Huffman 30
Figura 6 – Etapa 3 da construção da árvore de Huffman 31
Figura 7 – Etapa 4 da construção da árvore de Huffman 31
Figura 8 – Etapa 5 da construção da árvore de Huffman 32
Figura 9 – Etapa 6 da construção da árvore de Huffman 32
Figura 10 – Etapa 7 da construção da árvore de Huffman 33
Figura 11 – Etapa 8 da construção da árvore de Huffman 33
Figura 12 – Etapa 9 da construção da árvore de Huffman 34
Figura 13 – Etapa 10 da construção da árvore de Huffman 34
Figura 14 – Etapa 11 da construção da árvore de Huffman 35
Figura 15 – Estrutura do projeto GKompress 43
Figura 16 – Interface principal do sistema GKompress 44
Figura 17 – Tela de processamento e configuração dos algoritmos 44
Figura 18 – *Header* do arquivo BMP 45
Figura 19 – *Header* do arquivo WAV 46
Figura 20 – Visão geral das taxas de compressão por algoritmo 77
Figura 21 – *Scatter plot* dos resultados dos algoritmos Huffman 78
Figura 22 – Resultados de compressão do algoritmo LZ77 79
Figura 23 – *Scatter plot* dos resultados do algoritmo LZW 79
Figura 24 – Resultados de compressão do algoritmo GZIP 80
Figura 25 – *Scatter plot* dos resultados para arquivos TXT 81
Figura 26 – Resultados de compressão para arquivos BMP 82
Figura 27 – *Scatter plot* dos resultados para arquivos WAV 83

Lista de códigos

1 Estrutura da classe BMP 47
2 Método de leitura de dados da imagem BMP 48
3 Estrutura da classe WAV 49
4 Estrutura da classe BitWriter 51
5 Implementação do método writeBit 51
6 Implementação do método writeBits 51
7 Estrutura da classe BitReader 52
8 Implementação do método readBit 52
9 Implementação do método readBits 53
10 Estrutura do nó de Huffman 55
11 Construção da árvore de Huffman 56
12 Geração dos códigos de Huffman 56
13 Codificação dos dados com códigos de Huffman 57
14 Empacotamento *bit-a-bit* dos códigos de Huffman 57
15 Compressão de arquivos BMP com Huffman 58
16 Estrutura da classe base LZ77 59
17 Algoritmo de busca por correspondências no LZ77 61
18 Algoritmo de descompressão do LZ77 62
19 Estrutura da classe base LZW 63
20 Algoritmo de codificação LZW 64
21 Algoritmo de descompressão LZW 66
22 Compressão de arquivos WAV com LZW 67
23 Estrutura das classes GZIP 68
24 Implementação da compressão GZIP 69
25 Implementação da descompressão GZIP 70

Lista de tabelas

Tabela 1 – Frequências e probabilidades dos símbolos da fonte. 24
Tabela 2 – Exemplo de codificação de Huffman para uma mensagem. 28
Tabela 3 – Exemplo de funcionamento do algoritmo LZ77. 37
Tabela 4 – Composição do conjunto de dados utilizado nos experimentos. . . . 73
Tabela 5 – Resultados médios de compressão para arquivos TXT 81
Tabela 6 – Resultados médios de compressão para arquivos BMP 82
Tabela 7 – Resultados médios de compressão para arquivos WAV 83

Lista de abreviaturas e siglas

LZ77	Lempel-Ziv 1977 (algoritmo de compressão)
LZW	Lempel-Ziv-Welch (algoritmo de compressão)
GZIP	GNU zip (algoritmo de compressão)
GTK	GIMP Toolkit (biblioteca gráfica para interface)

Sumário

Lista de códigos **9**

1 INTRODUÇÃO **15**
1.1 Problema **16**
1.2 Objetivos **16**
1.2.1 Objetivo geral 17
1.2.2 Objetivos específicos 17

2 FUNDAMENTAÇÃO TEÓRICA **18**
2.1 Teoria da Informação **18**
2.1.1 Quantificação da Informação de Hartley 18
2.1.2 Conteúdo informacional de Shannon 21
2.1.3 Entropia 22
2.1.4 Redundância 23
2.1.5 Desigualdade de Kraft-McMillan e taxa de compressão 25
2.2 Compressão de dados **26**
2.3 Algoritmos clássicos de compressão sem perdas **27**
2.3.1 Codificação de Huffman 27
2.3.2 LZ77 36
2.3.3 LZW 39
2.3.4 GZIP 40

3 IMPLEMENTAÇÃO **42**
3.1 Interface Gráfica **44**
3.2 Manipulação de *Headers* BMP e WAV **45**
3.2.1 Implementação da Classe BMP 47
3.2.2 Implementação da Classe WAV 49
3.3 Classe *BitStream* **50**
3.4 Classe *BitWriter* **50**
3.5 Classe *BitReader* **52**
3.6 Importância da Classe *BitStream* **53**
3.7 Huffman **53**
3.8 LZ77 . . . . **59**
3.9 LZW **63**
3.10 GZIP **68**

4	**ANÁLISE COMPARATIVA**	**73**
4.1	**Geração do Conjunto de Dados**	**73**
4.1.1	Justificativa da Geração Sintética	73
4.1.2	Geração de Imagens BMP	74
4.1.3	Geração de Arquivos de Texto	74
4.1.4	Geração de Arquivos de Áudio WAV	75
4.2	**Metodologia de Avaliação**	**76**
4.3	**Análise Geral dos Resultados**	**76**
4.4	**Desempenho por Algoritmo**	**77**
4.4.1	Algoritmos Huffman	77
4.4.2	Algoritmo LZ77	78
4.4.3	Algoritmo LZW	79
4.4.4	Algoritmo GZIP	80
4.5	**Análise por Tipo de Arquivo**	**80**
4.5.1	Arquivos de Texto (TXT)	80
4.5.2	Arquivos de Imagem (BMP)	81
4.5.3	Arquivos de Áudio (WAV)	82
4.6	**Análise Estatística**	**83**
4.6.1	Teste de Significância	84
4.6.2	Correlação com Tamanho do Arquivo	84
4.7	**Discussão dos Resultados**	**85**
4.7.1	Algoritmos Huffman	85
4.7.2	Algoritmos Baseados em Dicionário (LZ77 e LZW)	86
4.7.3	Algoritmo GZIP	86
5	**CONCLUSÃO**	**87**
5.1	**Principais Contribuições**	**87**
5.2	**A Entropia como Fator Determinante**	**87**
5.2.1	Impacto da Entropia por Tipo de Arquivo	88
5.2.2	Algoritmos e Sua Relação com a Entropia	88
5.3	**Implicações Práticas**	**89**
5.4	**Limitações e Trabalhos Futuros**	**89**
5.4.1	Expansão do Conjunto de Dados e Formatos	89
5.4.2	Análise de Arquivos Já Comprimidos	90
5.4.3	Otimização de Performance e Benchmarks	90
5.4.4	Desenvolvimento de Algoritmos Híbridos e Adaptativos	91
5.4.5	Análise Teórica Aprofundada	91
5.4.6	Extensões do Sistema GKompress	92
5.5	**Considerações Finais**	**92**

REFERÊNCIAS **94**

1 Introdução

A compressão de dados é uma técnica essencial na Ciência da Computação, utilizada para reduzir o tamanho dos arquivos, otimizando o uso de recursos e acelerando a transmissão de dados – aspectos cruciais na era digital em que vivemos.

O desenvolvimento da área teve início na década de 1950, com os primeiros métodos voltados à otimização do espaço de armazenamento. Entre estes métodos, destaca-se o algoritmo de Huffman (HUFFMAN, 1952), que emprega uma codificação de prefixo variável, atribuindo códigos menores aos símbolos mais frequentes e maiores aos menos frequentes. Essa abordagem se consolidou como um dos pilares da área e é utilizada em formatos como ZIP e GZIP.

Posteriormente, novos algoritmos surgiram. Um dos mais significativos é o LZ77, proposto por Abraham Lempel e Jacob Ziv (ZIV; LEMPEL, 1977), que adota uma estratégia baseada em dicionários, substituindo sequências repetidas por referências a posições anteriores. A partir desta ideia, o LZW vem à tona aprimorando o LZ77 ao construir dinamicamente um dicionário de *strings* durante a compressão (WELCH, 1984), sendo amplamente utilizado em formatos como GIF e TIFF.

Outro algoritmo relevante é o GZIP, amplamente adotado em sistemas *Unix* e aplicações *web*. Ele combina a eficiência da codificação de Huffman com o LZ77, alcançando boas taxas de compressão sem perdas. Sua popularidade decorre da combinação de alta eficácia com a facilidade de descompressão.

Apesar do surgimento de novas técnicas, os algoritmos de Huffman, LZ77, LZW e GZIP continuam a formar a espinha dorsal dos métodos clássicos de compressão de dados e são amplamente utilizados em diversas aplicações.

Nos últimos anos, surgiram algoritmos mais modernos, como o Brotli – desenvolvido pela Google – que combina codificação de Huffman e transformações de fluxo de dados para obter taxas de compressão superiores às do GZIP (ALAKUIJALA *et al.*, 2016). Outro exemplo é o Zstandard (Zstd), criado pelo Facebook, que alia velocidade e compressão eficiente, sendo adequado tanto para compressão em tempo real quanto para grandes volumes de dados (COLLET, 2016).

Além disto, algoritmos baseados em aprendizado de máquina têm ganhado destaque, principalmente na compressão de imagens e áudio. Redes neurais como GANs (para imagens) e modelos como o WaveNet (para áudio) (DeepMind, 2016) vêm sendo explorados para melhorar a qualidade das compressões. No campo da compressão de vídeo, codecs como HEVC (H.265) e VVC vêm sendo desenvolvidos

para oferecer compressões mais eficazes em vídeos de alta definição.

A compressão de dados desempenha, portanto, um papel estratégico em diversas áreas tecnológicas, impactando diretamente sistemas de armazenamento, transmissão de dados, *streaming*, redes e computação em nuvem. A análise de desempenho dos algoritmos, considerando métricas como taxa de compressão e tempo de execução, é essencial para a escolha do método mais adequado a cada situação.

Neste contexto, apresenta-se um estudo aprofundado sobre os algoritmos clássicos de compressão Huffman, LZ77, LZW e GZIP. São abordados os fundamentos teóricos de cada método, suas implementações práticas em linguagem C++ com suporte a arquivos nos formatos .txt, .bmp e .wav, e uma interface gráfica desenvolvida com GTK. A avaliação experimental é realizada por meio de comparações de desempenho, com auxílio de *scripts* em Python para geração de gráficos e análise dos resultados. O objetivo é compreender o comportamento dos algoritmos em diferentes tipos de dados, destacando suas vantagens, limitações e aplicações mais adequadas.

O trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os fundamentos teóricos sobre compressão de dados, abordando a Teoria da Informação e os algoritmos Huffman, LZ77, LZW e GZIP. O Capítulo 3 descreve a implementação dos algoritmos, incluindo aspectos técnicos e ferramentas utilizadas. No Capítulo 4, é realizada uma análise comparativa com base nos testes realizados, considerando taxa de compressão e tempo de execução. Por fim, o Capítulo 5 apresenta as conclusões do trabalho e sugestões para pesquisas futuras.

1.1 Problema

Como os diferentes algoritmos clássicos de compressão de dados (Huffman, LZ77, LZW e GZIP) se comparam em termos de eficiência de compressão e tempo de execução, e qual o impacto dessas métricas em aplicações práticas que lidam com texto, imagem e áudio?

1.2 Objetivos

Este trabalho possui um objetivo geral e objetivos específicos que orientam o desenvolvimento da pesquisa e das implementações realizadas.

1.2.1 Objetivo geral

Investigar e comparar o desempenho de algoritmos clássicos de compressão de dados sem perdas, a saber: Huffman, LZ77, LZW e GZIP, por meio da implementação prática dessas técnicas, aplicando-as a diferentes tipos de arquivos e analisando sua eficiência em termos da taxa de compressão e tempo de execução.

1.2.2 Objetivos específicos

a) Estudar os fundamentos teóricos dos algoritmos de compressão sem perdas selecionados, compreendendo o seu funcionamento, estrutura e aplicação histórica;
b) Implementar, em linguagem C++, os algoritmos Huffman, LZ77, LZW e GZIP, com foco na compressão e descompressão de arquivos de texto (.txt), imagem *bitmap* (.bmp) e áudio *PCM* (.wav);
c) Desenvolver uma interface gráfica com a biblioteca GTK, que permita ao usuário escolher o algoritmo e o algoritmo de entrada de forma interativa;
d) Criar *scripts* em Python para a análise comparativa, organizando e visualizando os resultados experimentais por meio de gráficos e tabelas;
e) Avaliar os pontos fortes e fracos de cada algoritmo, considerando o tipo de dado, a estrutura do arquivo, o tempo de processamento e a taxa de compressão; e
f) Identificar o algoritmo mais adequado a cada cenário específico, com base nos resultados experimentais obtidos.

No Capítulo 2, serão apresentados os fundamentos teóricos que embasam este trabalho, incluindo os conceitos fundamentais da Teoria da Informação, como entropia e redundância, além da descrição detalhada dos algoritmos clássicos de compressão sem perdas Huffman, LZ77, LZW e GZIP, estabelecendo a base conceitual necessária para compreender as implementações e análises subsequentes.

2 Fundamentação teórica

Este capítulo apresenta os fundamentos teóricos que embasam o desenvolvimento deste trabalho. Para compreender o funcionamento e as diferenças entre os algoritmos de compressão de dados abordados, é necessário introduzir alguns conceitos essenciais.

Primeiramente, será apresentada a Teoria da Informação, a qual fornece o embasamento matemático para a compressão de dados sem perdas, por meio de conceitos como entropia e redundância. Em seguida, serão discutidos os princípios gerais da compressão de dados, com foco nas técnicas sem perdas, suas aplicações e métricas de desempenho. Por fim, serão detalhados os algoritmos clássicos de compressão sem perdas utilizados neste trabalho: Huffman, LZ77, LZW e GZIP, incluindo suas características, funcionamento e aplicações práticas.

A estrutura deste capítulo está organizada da seguinte forma:

a) **Seção 2.1** – Apresenta os principais conceitos da Teoria da Informação, com ênfase na entropia de Shannon e seu papel na compressão de dados;

b) **Seção 2.2** – Discute os fundamentos da compressão de dados sem perdas, destacando seus objetivos, métricas e distinções em relação à compressão com perdas; e

c) **Seção 2.3** – Descreve detalhadamente os algoritmos Huffman, LZ77, LZW e GZIP, explicando seus princípios de funcionamento, vantagens, limitações e casos de uso.

2.1 Teoria da Informação

A Teoria da Informação, proposta por Claude Shannon em 1948, constitui o alicerce teórico para diversos processos de codificação e compressão de dados na Ciência da Computação e Engenharia de Comunicações. Seu objetivo central é quantificar a informação contida em uma mensagem e estabelecer limites teóricos para a eficiência da transmissão e armazenamento de dados (SHANNON, 1948).

2.1.1 Quantificação da Informação de Hartley

Em 1928, Ralph V. L. Hartley publicou o artigo *Transmission of Information* (HARTLEY, 1928), no qual propôs uma medida quantitativa para a informação baseada unicamente em aspectos físicos e objetivos do processo de comunicação.

Sua motivação era eliminar fatores subjetivos, como interpretação ou significado, e estabelecer uma definição útil para aplicações em engenharia.

Hartley introduz seu raciocínio a partir da observação de sistemas de comunicação reais, como o sistema de cabo telegráfico submarino operado manualmente. Nesse sistema, o operador utiliza uma chave com três posições, por exemplo, tensão positiva, tensão nula e tensão negativa, cada uma representando um símbolo físico distinto. A mensagem enviada é uma sequência dessas seleções, e o receptor interpreta os sinais registrados por um oscilógrafo em uma fita fotosensível.

O ponto fundamental é que, em cada instante de envio, o operador faz uma escolha consciente entre múltiplos símbolos possíveis. À medida que as seleções ocorrem, mais possibilidades de mensagens são descartadas, o que equivale a dizer que a informação vai se tornando mais específica.

Hartley argumenta que a quantidade de informação transmitida deve depender do número de símbolos disponíveis a cada escolha e do número total de escolhas feitas. Por exemplo, se há $s = 3$ símbolos possíveis por escolha e n escolhas são feitas, o número total de sequências distintas possíveis é dado pela equação (2.1).

$$N = s^n \tag{2.1}$$

Esse valor representa a quantidade de mensagens distintas que o sistema é capaz de transmitir fisicamente. No entanto, usar diretamente esse número como medida de informação não é prático, pois ele cresce exponencialmente com o número de seleções. Para obter uma medida linear e proporcional ao número de escolhas, Hartley propõe tomar o logaritmo desse valor conforme a equação (2.2).

$$H = \log_b N = \log_b(s^n) = n \log_b s \tag{2.2}$$

Essa expressão garante que:

a) A medida de informação seja proporcional ao número de escolhas n;

b) Duas mensagens independentes tenham informação total igual à soma das informações individuais (propriedade de aditividade); e

c) O valor seja expresso em uma unidade prática — como bits, ao usar base $b = 2$.

A seguir são apresentados alguns exemplos que ilustram a aplicação da fórmula de Hartley:

Exemplo 1: Considere um sistema em que cada seleção permite escolher entre três símbolos distintos (por exemplo, $s = 3$). Se forem feitas 5 seleções ($n = 5$), o

número total de mensagens possíveis é dado pela equação (2.3).

$$N = 3^5 = 243 \tag{2.3}$$

A quantidade de informação total transmitida é aproximadamente 7,925 bits, conforme a equação (2.4).

$$H = \log_2 243 \approx 7{,}925 \text{ bits} \tag{2.4}$$

Ou, usando a fórmula direta da equação (2.5).

$$H = 5 \log_2 3 \approx 5 \times 1{,}585 \approx 7{,}925 \text{ bits} \tag{2.5}$$

Isso significa que, em média, cada seleção transmite aproximadamente 1,585 bits de informação.

Exemplo 2: Suponha um sistema em que o transmissor pode escolher entre 2 símbolos diferentes (como 0 e 1, em um código binário). Uma única escolha transmite a quantidade de informação de 1 bit, conforme a equação (2.6).

$$I = \log_2 2 = 1 \text{ bit} \tag{2.6}$$

Com 8 escolhas, tem-se 8 bits de informação, conforme a equação (2.7).

$$H = 8 \log_2 2 = 8 \text{ bits} \tag{2.7}$$

Exemplo 3: Considere agora um sistema com 4 símbolos distintos (por exemplo, A, B, C, D). Cada escolha transmite a quantidade de informação de 2 bits, conforme a equação (2.8).

$$I = \log_2 4 = 2 \text{ bits} \tag{2.8}$$

Após 5 seleções, a quantidade total de informação transmitida é 10 bits, conforme a equação (2.9).

$$H = 5 \log_2 4 = 10 \text{ bits} \tag{2.9}$$

Exemplo 4: Em um sistema decimal com 10 símbolos (de 0 a 9), uma única seleção transmite aproximadamente 3,32 bits de informação, conforme a equação (2.10).

$$I = \log_2 10 \approx 3{,}32 \text{ bits} \tag{2.10}$$

Três seleções transmitiriam aproximadamente 9,97 bits de informação, conforme a equação (2.11).

$$H = 3 \log_2 10 \approx 9{,}97 \text{ bits} \tag{2.11}$$

Esses exemplos ilustram que, quanto maior o número de símbolos possíveis por escolha, maior é o conteúdo de informação por escolha individual. A grande sacada de Hartley foi perceber que a informação é proporcional ao número de seleções, e que o logaritmo do número de possibilidades elimina a explosão exponencial, fornecendo uma medida prática e linear.

2.1.2 Conteúdo informacional de Shannon

A abordagem de Hartley, no entanto, assume que todos os símbolos têm igual probabilidade de ocorrência, o que nem sempre é verdadeiro na prática. Para resolver essa limitação, Shannon generalizou o conceito e definiu o conteúdo informacional[1] associada a um evento x com probabilidade $P(x)$ conforme a equação (2.12).

$$I(x) = -\log_b P(x) \tag{2.12}$$

A ideia principal é que o valor informacional de uma mensagem depende do grau em que a mesma é surpreendente. Se um evento altamente provável ocorrer, a mensagem carrega pouca informação; por outro lado, se um evento improvável acontece, diz-se que carrega muita informação.

Por definição, se $P(x) = 0$ então $I(x) = 0$, ou seja, se o evento for impossível, o valor informacional é nulo.

Considere dois eventos distintos:

a) Evento A: "Vai chover amanhã", com probabilidade $P(A) = 0{,}9$;
b) Evento B: "Vai nevar amanhã", com probabilidade $P(B) = 0{,}01$.

Aplicando a fórmula do conteúdo informacional com base 2, obtêm-se as equações (2.13) e (2.14).

$$I(A) = -\log_2(0{,}9) \approx 0{,}152 \text{ bits} \tag{2.13}$$

$$I(B) = -\log_2(0{,}01) \approx 6{,}644 \text{ bits} \tag{2.14}$$

Isso significa que a ocorrência de A transmite pouca informação, pois é esperada. Já a ocorrência de B transmite uma quantidade significativamente maior de informação, pois representa um evento altamente improvável — portanto, mais surpreendente.

A informação está relacionada com a incerteza: quanto menor a probabilidade de um evento, maior o seu conteúdo informacional.

[1] Em inglês, *information content*, *surprisal* ou *self-information*

2.1.3 Entropia

A entropia de uma fonte, então, é o valor esperado do conteúdo de informação de seus símbolos, conforme definido pela equação (2.15).

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i) \tag{2.15}$$

Essa formulação considera a estrutura estatística dos dados, permitindo medir a incerteza média da fonte — e, consequentemente, o limite teórico mínimo para compressão sem perdas.

Considere uma variável aleatória X que representa o lançamento de uma moeda, em que os possíveis resultados são:

a) $x_1 =$ cara, com probabilidade $P(x_1) = p$; e

b) $x_2 =$ coroa, com probabilidade $P(x_2) = 1 - p$.

A entropia da variável X é dada pela equação (2.16).

$$H(X) = -p \log_2 p - (1 - p) \log_2 (1 - p) \tag{2.16}$$

Essa função atinge seu valor máximo quando $p = 0{,}5$, ou seja, quando a moeda é justa e os dois eventos são igualmente prováveis. Nesse caso, a entropia é de 1 bit, conforme a equação (2.17).

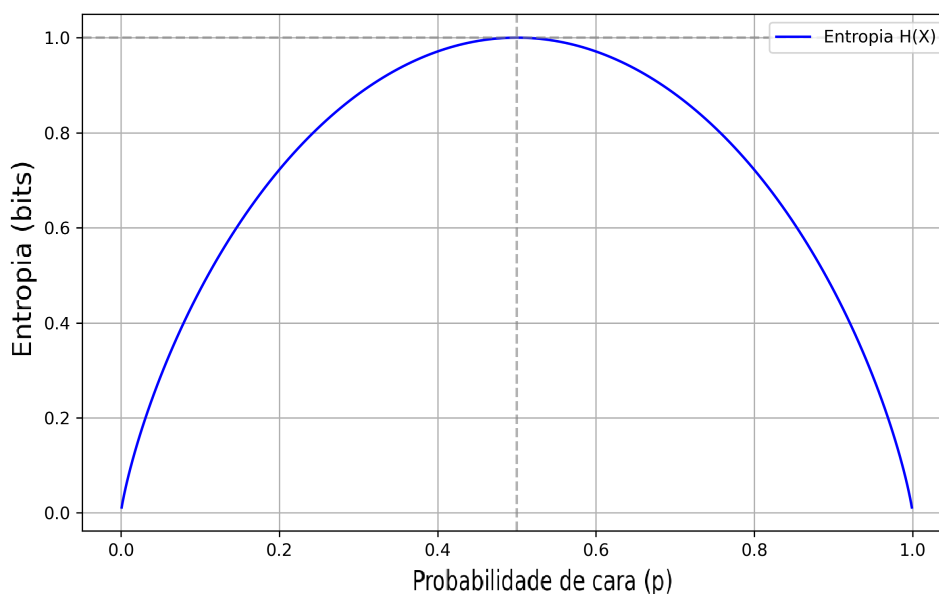
$$H(X) = -0{,}5 \log_2 0{,}5 - 0{,}5 \log_2 0{,}5 = -2 \times 0{,}5 \times (-1) = 1 \text{ bit} \tag{2.17}$$

Esse valor representa o máximo de incerteza possível para uma variável binária: não se sabe nada sobre qual será o resultado. Por outro lado, se a moeda estiver viciada e sempre cair do mesmo lado (por exemplo, $p = 0$ ou $p = 1$), então a entropia é de 0 bits, conforme a equação (2.18).

$$H(X) = -1 \log_2 1 = 0 \text{ bits} \tag{2.18}$$

Ou seja, não há incerteza — o resultado é completamente previsível, e, portanto, não há informação nova sendo transmitida. A Figura 1 ilustra graficamente como a entropia varia em função da probabilidade p.

Figura 1 – Entropia de uma moeda em função de p



Fonte: Elaborada pelo autor

2.1.4 Redundância

A redundância representa a diferença entre o comprimento médio real de codificação e a entropia da fonte. Em outras palavras, trata-se da quantidade de informação excedente que pode estar presente nos dados, muitas vezes introduzida por razões como segurança, integridade ou simplicidade de codificação. Quanto maior for essa redundância estatística, maior será o potencial de compressão sem perdas.

A relação entre a entropia $H(X)$ de uma fonte e o comprimento médio L de um código pode ser expressa pela equação (2.19) em que R é a redundância. Um código é considerado eficiente quando sua média de bits por símbolo se aproxima da entropia da fonte, ou seja, quando $R \to 0$.

$$R = L - H(X) \tag{2.19}$$

O principal objetivo dos algoritmos de compressão de dados é justamente reduzir essa redundância presente na representação original da informação. Ao identificar e eliminar padrões repetitivos, estruturas previsíveis ou codificações desnecessárias, é possível representar os dados com uma quantidade menor de bits, mantendo integralmente a informação. Assim, a compressão atua aproximando o código do limite teórico mínimo imposto pela entropia.

Considere uma fonte com três símbolos e as seguintes frequências, conforme apresentado na Tabela 1.

Tabela 1 – Frequências e probabilidades dos símbolos da fonte.

Símbolo	Frequência	Probabilidade
A	50	0,5
B	30	0,3
C	20	0,2

Fonte: Elaborada pelo autor

A entropia dessa fonte, em bits por símbolo, é dada pela equação (2.20).

$$H(X) = -(0{,}5\log_2 0{,}5 + 0{,}3\log_2 0{,}3 + 0{,}2\log_2 0{,}2) \approx 1{,}485 \text{ bits/símbolo} \tag{2.20}$$

Após aplicar um algoritmo de compressão de dados qualquer, suponha que os símbolos sejam codificados da seguinte forma:

a) A $\rightarrow$ 1 bit;

b) B $\rightarrow$ 2 bits; e

c) C $\rightarrow$ 2 bits.

O comprimento médio da codificação é dado pela equação (2.21).

$$L = (0{,}5 \times 1) + (0{,}3 \times 2) + (0{,}2 \times 2) = 1{,}5 \text{ bits/símbolo} \tag{2.21}$$

Portanto, a redundância é dada pela equação (2.22).

$$R = L - H(X) = 1{,}5 - 1{,}485 = 0{,}015 \text{ bits/símbolo} \tag{2.22}$$

Esse pequeno valor de R indica que o código comprimido se aproxima do limite teórico, demonstrando alta eficiência.

É importante destacar que a entropia $H(X)$ é uma propriedade intrínseca da fonte, determinada unicamente pela distribuição de probabilidades dos símbolos, e representa a quantidade média de informação por símbolo. Portanto, seu valor não se altera durante o processo de compressão. O que se modifica é o comprimento médio de codificação L, que tende a se aproximar de $H(X)$ à medida que o código se torna mais eficiente.

Dessa forma, ao reduzir o número médio de bits por símbolo, não ocorre uma diminuição da entropia da fonte, mas sim um aumento da quantidade de informação útil carregada por cada bit. Esse efeito pode ser expresso pela razão entre a entropia e o comprimento médio conforme a equação (2.23) em que η representa a eficiência do código.

$$\eta = \frac{H(X)}{L} \tag{2.23}$$

À medida que $L \to H(X)$, tem-se $\eta \to 1$, indicando que cada bit transmitido se torna mais informativo e menos redundante. Esse comportamento evidencia a eficácia da compressão, que reduz o volume de dados a serem armazenados ou transmitidos sem comprometer o conteúdo informacional original.

2.1.5 Desigualdade de Kraft-McMillan e taxa de compressão

Ao estudar a relação entre entropia e os algoritmos de compressão, dois conceitos adicionais merecem destaque: a desigualdade de Kraft-McMillan e a taxa de compressão.

A desigualdade de Kraft-McMillan estabelece uma condição necessária e suficiente para a existência de códigos prefixos, ou seja, códigos em que nenhum código é prefixo de outro, como no algoritmo de Huffman. Seja l_i o comprimento do código associado ao símbolo i. A desigualdade é dada pela equação (2.24).

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1 \tag{2.24}$$

O significado da desigualdade é que, se os comprimentos l_i satisfazem a condição, então é matematicamente garantido que existe ao menos um conjunto de códigos de prefixo que pode ser atribuído aos símbolos. Em outras palavras, é possível construir um código em que nenhum símbolo seja prefixo de outro, assegurando uma decodificação instantânea e sem ambiguidades. A desigualdade, portanto, não descreve o código em si, mas estabelece a condição necessária e suficiente para a sua existência.

Por exemplo, considere um conjunto de símbolos $\{A, B, C\}$ com comprimentos de código $l_A = 1$, $l_B = 2$ e $l_C = 2$. A verificação da desigualdade resulta no valor dado pela equação (2.25).

$$2^{-1} + 2^{-2} + 2^{-2} = 0{,}5 + 0{,}25 + 0{,}25 = 1 \tag{2.25}$$

Como a desigualdade é satisfeita, é possível construir um código de prefixo válido, como $A = 0$, $B = 10$ e $C = 11$.

Ao se tratar da eficiência de um algoritmo, define-se a *taxa de compressão* como a razão entre o tamanho do arquivo original e o tamanho do arquivo comprimido, conforme a equação (2.26).

$$\text{Taxa de compressão} = \frac{T_{\text{original}}}{T_{\text{comprimido}}} \tag{2.26}$$

Além disso, é comum expressar a *redução percentual* obtida pela compressão, conforme a equação (2.27).

$$\text{Redução} = \left(1 - \frac{T_{\text{comprimido}}}{T_{\text{original}}}\right) \times 100\% \tag{2.27}$$

Essas métricas permitem comparar algoritmos de compressão distintos de forma objetiva, complementando a análise teórica baseada em entropia e redundância.

Por fim, cabe destacar que a entropia pode ser definida não apenas em termos de símbolos isolados, mas também considerando dependências entre símbolos consecutivos, conhecida como *entropia de ordem superior*. Esse conceito é explorado por algoritmos como LZ77 e LZW, que buscam padrões e repetições em sequências, indo além da simples análise da frequência de símbolos individuais.

2.2 Compressão de dados

A compressão de dados consiste em um conjunto de técnicas destinadas a representar informações de forma mais compacta, reduzindo o espaço necessário para armazenamento ou o tempo exigido para transmissão. Trata-se de um campo fundamental da Ciência da Computação, com aplicações diretas em praticamente todas as áreas que envolvem o tratamento de dados digitais, como bancos de dados, sistemas de arquivos, comunicação em rede, multimídia e computação em nuvem.

O princípio básico da compressão é a exploração da redundância presente nos dados. Arquivos digitais, independentemente de sua natureza (texto, imagem, áudio ou vídeo), frequentemente apresentam padrões estatísticos, repetições ou estruturas previsíveis que podem ser representados de maneira mais eficiente. A eliminação ou redução dessa redundância resulta em representações mais curtas, possibilitando economia de recursos sem comprometer a utilidade da informação.

De maneira geral, os métodos de compressão podem ser classificados em duas categorias: compressão sem perdas e compressão com perdas.

Na compressão sem perdas, os dados comprimidos podem ser restaurados integralmente em sua forma original após a descompressão. Não ocorre perda de informação, sendo essa abordagem indispensável em contextos em que a integridade dos dados é crucial, como em documentos de texto, programas executáveis, imagens médicas e determinados formatos de áudio.

Já a compressão com perdas permite alcançar taxas de compressão mais elevadas ao descartar parte das informações, geralmente aquelas que são perceptualmente redundantes ou irrelevantes. Embora o arquivo reconstruído não seja idêntico ao original, a diferença pode ser imperceptível ao usuário final. Esse modelo é amplamente utilizado em mídias digitais, como fotografias (JPEG), música (MP3, AAC) e vídeo (MPEG, H.264).

O presente trabalho concentra-se exclusivamente na compressão sem perdas, uma vez que seu objetivo é a análise de algoritmos clássicos de codificação que buscam aproximar-se do limite teórico estabelecido pela entropia da fonte de informação. Essa escolha se justifica não apenas pela relevância prática da compressão sem perdas em diversos domínios, mas também por seu caráter teórico, que permite avaliar a eficiência dos algoritmos em termos formais, sem a interferência de fatores perceptuais.

Para avaliar a eficácia de um algoritmo de compressão, algumas métricas são comumente empregadas. O ganho de compressão é uma medida complementar, definida pela razão entre o tamanho original e o tamanho comprimido, indicando quantas vezes o volume de dados foi reduzido. Além disso, o tempo de execução é importante, representando o tempo necessário para realizar os processos de compressão e descompressão.

Em síntese, a compressão de dados busca otimizar recursos computacionais ao reduzir a quantidade de informação redundante em uma representação digital. A decisão entre compressão sem perdas ou com perdas depende do contexto e das exigências de cada aplicação. No âmbito deste trabalho, o foco em algoritmos de compressão sem perdas permite investigar de forma detalhada a relação entre teoria da informação e implementação prática, bem como comparar a eficiência de diferentes abordagens clássicas nesse domínio.

2.3 Algoritmos clássicos de compressão sem perdas

Os algoritmos de compressão sem perdas constituem a base teórica e prática de diversos sistemas computacionais modernos. Entre os mais conhecidos e aplicados estão os algoritmos de Huffman, LZ77, LZW e GZIP, cada um com características específicas que os tornam adequados a diferentes contextos de uso. A seguir, apresentam-se seus princípios de funcionamento, vantagens, limitações e principais aplicações.

2.3.1 Codificação de Huffman

O algoritmo de Huffman, proposto por David A. Huffman em 1952 (HUFFMAN, 1952), é um método de codificação estatística que constrói códigos de prefixo ótimos

para compressão sem perdas. O algoritmo fundamenta-se no princípio de atribuir códigos binários de comprimentos variáveis aos símbolos de uma fonte, de modo que símbolos mais frequentes recebem códigos mais curtos, minimizando assim o comprimento médio da codificação.

A construção do código de Huffman é realizada através da criação de uma árvore binária, seguindo os seguintes passos:

a) **Análise de frequência**: Conta-se a frequência de ocorrência de cada símbolo no conjunto de dados a ser comprimido;

b) **Criação de nós folha**: Cada símbolo é representado por um nó folha, associado à sua respectiva frequência;

c) **Construção da árvore**: Iterativamente, os dois nós com menores frequências são combinados em um novo nó pai, cuja frequência é a soma das frequências dos nós filhos. Este processo continua até restar apenas um nó (a raiz); e

d) **Atribuição de códigos**: Percorre-se a árvore da raiz até as folhas, atribuindo '0' para ramos esquerdos e '1' para ramos direitos (ou vice-versa). O código de cada símbolo é formado pela sequência de bits do caminho da raiz até sua folha correspondente.

Considere uma mensagem com os símbolos e suas respectivas frequências, como mostrado na Tabela 2.

Tabela 2 – Exemplo de codificação de Huffman para uma mensagem.

Símbolo	Frequência	Probabilidade	Código Huffman
A	45	0,45	0
B	13	0,13	101
C	12	0,12	100
D	16	0,16	111
E	9	0,09	1100
F	5	0,05	1101

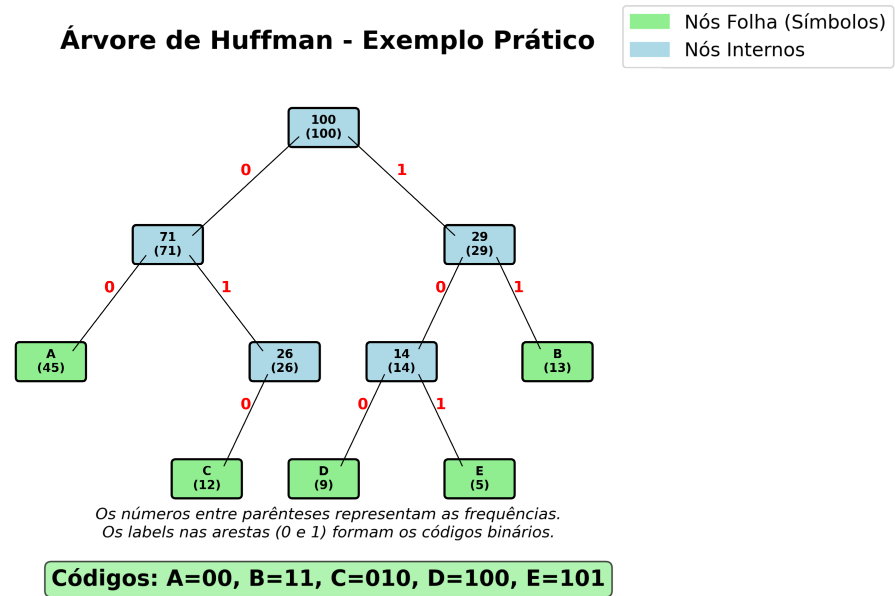
Fonte: Elaborada pelo autor

O comprimento médio da codificação seria dado pela equação (2.28).

$$L = 0{,}45 \times 1 + 0{,}13 \times 3 + 0{,}12 \times 3 + 0{,}16 \times 3 + 0{,}09 \times 4 + 0{,}05 \times 4 = 2{,}04 \text{ bits/símbolo} \quad (2.28)$$

Comparado com uma codificação binária fixa (que exigiria $\lceil \log_2 6 \rceil = 3$ bits por símbolo), o algoritmo de Huffman oferece uma redução significativa no tamanho da representação. A Figura 2 apresenta a árvore binária construída para este exemplo.

Figura 2 – Exemplo de uma Árvore de Huffman



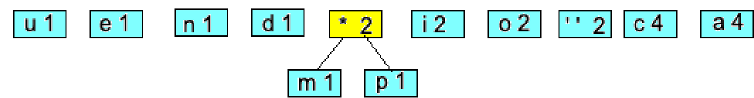
Fonte: Elaborada pelo autor

A seguir, são apresentadas as etapas da construção da árvore de Huffman para a frase “ciencia da compuacao”. As Figuras 3 a 14 ilustram o processo passo a passo, desde a inicialização com os nós folha até a formação da árvore completa.

Figura 3 – Etapa 0 da construção da árvore de Huffman

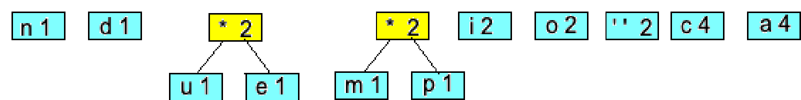
Fonte: Elaborada pelo autor

Figura 4 – Etapa 1 da construção da árvore de Huffman



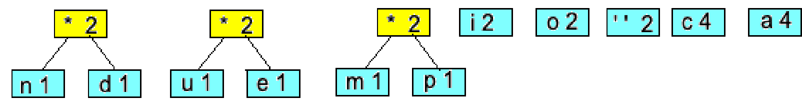
Fonte: Elaborada pelo autor

Figura 5 – Etapa 2 da construção da árvore de Huffman



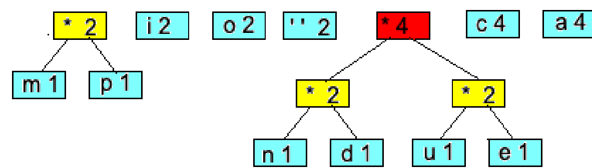
Fonte: Elaborada pelo autor

Figura 6 – Etapa 3 da construção da árvore de Huffman



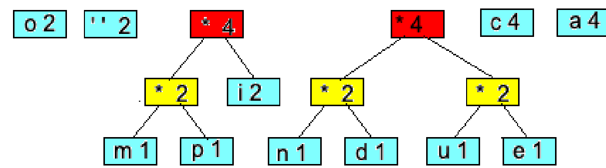
Fonte: Elaborada pelo autor

Figura 7 – Etapa 4 da construção da árvore de Huffman



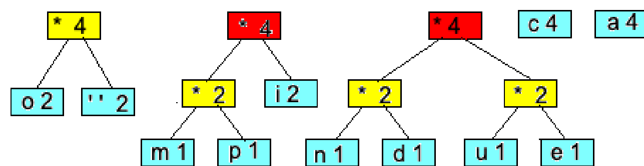
Fonte: Elaborada pelo autor

Figura 8 – Etapa 5 da construção da árvore de Huffman



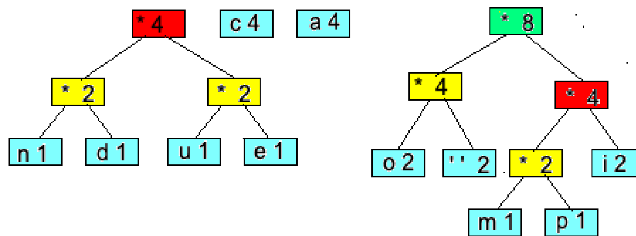
Fonte: Elaborada pelo autor

Figura 9 – Etapa 6 da construção da árvore de Huffman



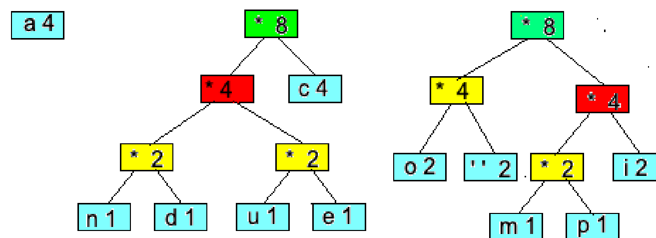
Fonte: Elaborada pelo autor

Figura 10 – Etapa 7 da construção da árvore de Huffman



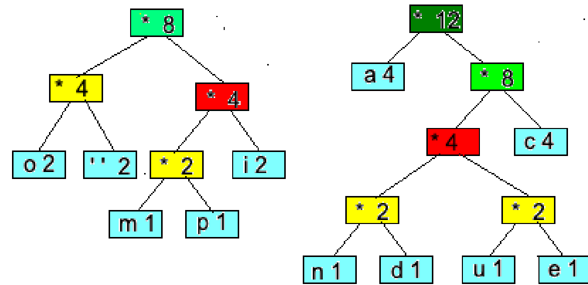
Fonte: Elaborada pelo autor

Figura 11 – Etapa 8 da construção da árvore de Huffman



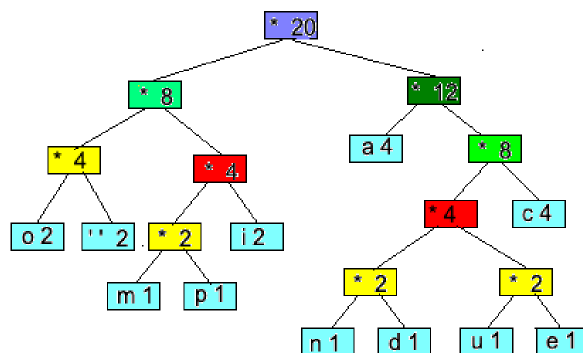
Fonte: Elaborada pelo autor

Figura 12 – Etapa 9 da construção da árvore de Huffman



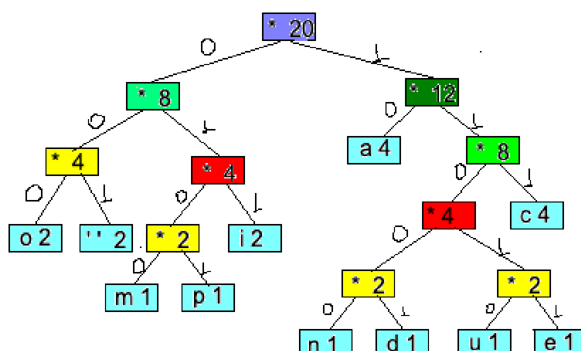
Fonte: Elaborada pelo autor

Figura 13 – Etapa 10 da construção da árvore de Huffman



Fonte: Elaborada pelo autor

Figura 14 – Etapa 11 da construção da árvore de Huffman



Fonte: Elaborada pelo autor

O algoritmo de Huffman possui propriedades teóricas importantes:

a) **Otimalidade**: Para uma fonte com probabilidades conhecidas e símbolos independentes, o código de Huffman minimiza o comprimento médio de codificação entre todos os códigos de prefixo possíveis;

b) **Satisfação da desigualdade de Kraft**: Os comprimentos dos códigos gerados sempre satisfazem a equação (2.24), garantindo a existência de um código de prefixo válido; e

c) **Limitação pela entropia**: O comprimento médio L do código de Huffman satisfaz a desigualdade expressa pela equação (2.29), em que $H(X)$ é a entropia da fonte (equação (2.15)).

$$H(X) \leq L < H(X) + 1 \tag{2.29}$$

Existem algumas variações importantes do algoritmo básico de Huffman:

a) **Huffman adaptativo**: Ajusta dinamicamente o código conforme novos símbolos são processados, sem necessidade de conhecimento prévio das frequências;

b) **Huffman canônico**: Produz uma representação padronizada do código que facilita a transmissão da tabela de códigos; e

c) **Huffman de duas passadas**: Realiza uma primeira passada para calcular frequências e uma segunda para codificar, sendo mais eficiente em termos de memória.

Entre as principais vantagens do algoritmo de Huffman destacam-se sua otimalidade para fontes com probabilidades conhecidas e símbolos independentes, a simplicidade de implementação e baixo *overhead* computacional, a garantia matemática de código de prefixo válido e a eficiência comprovada em dados com distribuições não uniformes.

Por outro lado, o algoritmo apresenta algumas limitações importantes. Sua eficiência depende da precisão das estimativas de probabilidade dos símbolos e não se adapta dinamicamente a variações estatísticas da fonte na versão estática. Além disso, apresenta desempenho subótimo em fontes com distribuições uniformes e requer a transmissão ou armazenamento da tabela de códigos junto com os dados comprimidos.

O algoritmo de Huffman encontra aplicação como componente fundamental em formatos de compressão como DEFLATE (ZIP, PNG), na codificação de coeficientes em compressão de imagem (JPEG), em protocolos de comunicação que requerem compressão em tempo real e em sistemas de armazenamento que priorizam alta taxa de compressão para dados textuais.

2.3.2 LZ77

O algoritmo LZ77, proposto por Abraham Lempel e Jacob Ziv em 1977 (ZIV; LEMPEL, 1977), representa um marco na compressão de dados sem perdas ao introduzir o conceito de compressão baseada em dicionário dinâmico. Diferentemente dos métodos estatísticos como Huffman, que dependem da frequência de símbolos individuais, o LZ77 explora a redundância temporal presente em sequências de dados, identificando e codificando repetições de subsequências já observadas.

O LZ77 opera com base em uma janela deslizante que divide o fluxo de entrada em duas regiões principais:

a) **Buffer de busca**: região que contém os dados já processados, servindo como dicionário para identificar repetições; e
b) **Buffer de antecipação**: região que contém os dados ainda não processados, onde se busca por correspondências com o buffer de busca.

O algoritmo funciona através dos seguintes passos:

a) **Inicialização**: Define-se o tamanho do buffer de busca tipicamente entre 4KB e 32KB e do buffer de antecipação entre 4 e 258 bytes;
b) **Busca por correspondências**: Para cada posição no buffer de antecipação, procura-se a maior subsequência que tenha uma correspondência no buffer de busca;

c) **Codificação**: Se uma correspondência é encontrada, ela é codificada como um triplo $(\text{distância}, \text{comprimento}, \text{próximo_símbolo})$, em que:
   - *distância*: número de posições para trás onde a correspondência foi encontrada;
   - *comprimento*: número de símbolos que correspondem; e
   - *próximo_símbolo*: primeiro símbolo após a correspondência que não faz parte dela.

d) **Avanço da janela**: A janela desliza para a direita, incorporando os símbolos processados ao buffer de busca e carregando novos símbolos no buffer de antecipação; e

e) **Repetição**: O processo continua até que todos os dados sejam processados.

Considere a sequência de entrada: `"AACAACABCABAAAC"`, assumindo uma janela de busca de tamanho 6 e buffer de antecipação de tamanho 4, o processo de compressão pode ser acompanhado na Tabela 3.

Tabela 3 – Exemplo de funcionamento do algoritmo LZ77.

Passo	Buffer de busca	Buffer antecipação	Saída
1	(vazio)	AACA	(0,0,A)
2	A	ACAA	(1,1,C)
3	AAC	AACA	(3,4,B)
4	CAACAB	CABA	(3,3,A)
5	ABCABA	AAC	(1,2,C)

Fonte: Elaborada pelo autor

A saída comprimida seria:

`(0,0,A); (1,1,C); (3,4,B); (3,3,A); (1,2,C)`

O algoritmo LZ77 possui diversas variações que melhoram seu desempenho:

a) **LZSS (Lempel-Ziv-Storer-Szymanski)**: Elimina o campo "próximo_símbolo"do triplo, usando um bit de flag para distinguir entre literais e referências, resultando em melhor compressão;

b) **LZ77 com *hash***: Utiliza tabelas *hash* para acelerar a busca por correspondências, reduzindo a complexidade de $O(nw)$ para $O(n)$, em que w é o tamanho da janela;

c) **LZ77 com árvores de sufixos**: Emprega estruturas de dados mais sofisticadas para otimizar a busca, especialmente eficaz para textos longos; e

d) **LZ77 com codificação aritmética**: Combina o LZ77 com codificação aritmética em vez de representação direta dos triplos, melhorando a taxa de compressão.

A complexidade computacional do LZ77 depende da implementação:

a) **Implementação ingênua**: $O(nwl)$, em que n é o tamanho da entrada, w o tamanho da janela de busca e l o comprimento máximo de correspondência;
b) **Com otimizações (*hash*)**: $O(nl)$ no caso médio, $O(nwl)$ no pior caso; e
c) **Espaço**: $O(w + l)$ para armazenar as janelas de busca e antecipação.

A descompressão é significativamente mais rápida, com complexidade $O(n)$, pois requer apenas a expansão sequencial das referências.

O LZ77 possui importantes propriedades teóricas:

a) **Universalidade**: É um algoritmo universal, ou seja, para qualquer fonte estacionária e ergódica, a taxa de compressão converge para a taxa de entropia da fonte quando o tamanho da janela tende ao infinito;
b) **Adaptabilidade**: Não requer conhecimento prévio das estatísticas da fonte, adaptando-se dinamicamente aos padrões presentes nos dados; e
c) **Garantia de não expansão**: No pior caso, o tamanho da saída é limitado pelo tamanho da entrada mais um *overhead* constante por bloco processado.

A eficiência do LZ77 depende criticamente da escolha de seus parâmetros:

a) **Tamanho do buffer de busca**: Janelas maiores permitem encontrar correspondências mais distantes, mas aumentam o tempo de processamento e o espaço necessário para codificar as distâncias;
b) **Comprimento máximo de correspondência**: Valores maiores permitem codificar repetições mais longas, mas requerem mais bits para representar o comprimento; e
c) **Comprimento mínimo de correspondência**: Correspondências muito curtas podem não compensar o *overhead* da codificação, sendo mais eficiente tratá-las como literais.

O algoritmo LZ77 apresenta várias vantagens importantes. É um algoritmo universal que não requer conhecimento prévio das estatísticas da fonte, adapta-se dinamicamente aos padrões presentes nos dados e apresenta excelente desempenho em dados com repetições estruturais, como textos, código fonte e dados binários com padrões. Além disso, oferece descompressão rápida e de baixa complexidade, contando com uma base teórica sólida que garante convergência para a taxa de entropia.

Entretanto, o algoritmo apresenta algumas limitações. Sua eficiência é limitada pelo tamanho da janela de busca, que determina quão distantes podem estar as repetições, e apresenta desempenho inferior em dados altamente aleatórios ou sem padrões repetitivos. Há ainda um *overhead* significativo para representar as referências quando as correspondências são curtas, complexidade computacional elevada na compressão (especialmente sem otimizações) e sensibilidade à escolha dos parâmetros, como tamanhos de janela e buffer.

O LZ77 encontra ampla aplicação como base fundamental do algoritmo DEFLATE, usado em formatos ZIP, PNG e HTTP/HTTPS (GZIP). É utilizado na compressão de arquivos de texto, código fonte e documentos estruturados, em sistemas de backup e arquivamento que priorizam alta taxa de compressão, em protocolos de comunicação que requerem compressão em tempo real com boa adaptabilidade e na compressão de dados binários com padrões repetitivos, como executáveis e bibliotecas.

2.3.3 LZW

O algoritmo LZW (Lempel-Ziv-Welch), desenvolvido por Terry Welch em 1984 (WELCH, 1984), representa uma evolução significativa dos algoritmos de compressão baseados em dicionário. Ao contrário do LZ77, que utiliza referências de deslocamento para identificar repetições em uma janela deslizante, o LZW constrói dinamicamente um dicionário de padrões encontrados nos dados, substituindo sequências por índices de um dicionário em crescimento contínuo.

O funcionamento do algoritmo LZW baseia-se na construção incremental de um dicionário que inicialmente contém todos os símbolos possíveis da fonte. Durante o processo de compressão, o algoritmo processa sequencialmente os dados de entrada, mantendo uma string atual que é expandida com cada novo símbolo. Quando uma nova sequência é encontrada, o código da string anterior é emitido e a nova sequência é adicionada ao dicionário. Este processo permite que o algoritmo aprenda padrões dinamicamente e os utilize para compressão posterior.

A descompressão LZW opera de forma similar, reconstruindo o dicionário durante o processo de decodificação. O algoritmo lê códigos sequencialmente, consultando o dicionário para recuperar as strings correspondentes. Uma característica especial do LZW é que, quando encontra um código que ainda não existe no dicionário, utiliza a propriedade de que o próximo código a ser adicionado corresponde à string anterior mais seu primeiro caractere.

O algoritmo LZW apresenta como principais vantagens a simplicidade de implementação em comparação com outros métodos baseados em dicionário, o bom desempenho em dados com repetições estruturadas e a independência de modelos

estatísticos prévios. Além disso, oferece compressão eficaz para uma ampla variedade de tipos de dados, incluindo texto, imagens e dados binários com padrões repetitivos.

Entretanto, o LZW apresenta algumas limitações importantes. O algoritmo pode gerar tabelas de dicionário grandes durante o processamento, o que demanda controle cuidadoso para evitar estouro de memória. Em fluxos curtos ou dados com pouca redundância, a eficiência pode ser significativamente reduzida devido ao *overhead* da construção do dicionário. Adicionalmente, o algoritmo não se adapta bem a mudanças bruscas nas características estatísticas dos dados, uma vez que o dicionário é construído sequencialmente.

O algoritmo LZW foi amplamente empregado no formato GIF para compressão de imagens, demonstrando eficiência particularmente boa para imagens com áreas de cor uniforme. Também foi utilizado em implementações de compressão em sistemas UNIX, como o utilitário *compress*, e em diversos formatos de arquivo que requeriam compressão rápida e eficiente. A influência do LZW pode ser observada em algoritmos posteriores que incorporaram conceitos de dicionário dinâmico, estabelecendo-se como um marco na evolução dos métodos de compressão sem perdas.

2.3.4 GZIP

O GZIP representa uma evolução significativa na compressão de dados ao combinar os princípios do algoritmo LZ77 com a codificação de Huffman, criando um formato híbrido que explora tanto redundâncias estruturais quanto estatísticas. Desenvolvido como parte do projeto GNU, o GZIP utiliza o algoritmo DEFLATE, que implementa uma versão refinada do LZ77 seguida por codificação de Huffman dinâmica, resultando em compressão altamente eficiente para uma ampla variedade de tipos de dados.

O processo de compressão GZIP inicia com a aplicação do algoritmo LZ77 para detectar e codificar padrões repetidos nos dados de entrada. O LZ77 identifica sequências que já foram observadas anteriormente e as substitui por referências (deslocamento, comprimento, próximo símbolo), reduzindo significativamente a redundância estrutural. Em seguida, os símbolos literais e as referências de distância são codificados utilizando árvores de Huffman dinâmicas, que são construídas especificamente para os dados sendo comprimidos, otimizando ainda mais a representação.

A estrutura do arquivo GZIP inclui um cabeçalho rico em metadados, contendo informações sobre o método de compressão utilizado, *timestamp* de criação, *flags* extras, nome do arquivo original e comentários opcionais. Após os dados comprimidos, um *trailer* contém o *checksum* CRC32 para verificação de integridade e o tamanho original do arquivo, garantindo que a descompressão possa ser validada e que o arquivo

original seja reconstruído corretamente.

O algoritmo GZIP apresenta como principais vantagens o alto grau de compressão obtido em uma ampla variedade de tipos de dados, a eficiência prática comprovada através de décadas de uso, o suporte robusto a múltiplos arquivos e metadados de integridade que garantem a confiabilidade dos dados. Além disso, oferece flexibilidade na escolha dos níveis de compressão, permitindo balancear tempo de processamento e taxa de compressão conforme a necessidade da aplicação.

Entretanto, o GZIP apresenta algumas limitações importantes. O algoritmo apresenta maior complexidade computacional em comparação com algoritmos isolados, especialmente durante a fase de compressão, que requer múltiplas passadas pelos dados. Embora altamente eficiente para a maioria dos cenários, pode não atingir a compressão máxima em todos os tipos de dados, especialmente aqueles com características estatísticas muito específicas. Adicionalmente, o *overhead* dos metadados pode ser significativo para arquivos muito pequenos.

O GZIP é amplamente utilizado em sistemas operacionais e protocolos de rede, estabelecendo-se como padrão de fato em distribuições de arquivos no ambiente UNIX/Linux e em transmissões *web* através do protocolo HTTP/HTTPS. Sua implementação na biblioteca *zlib* tornou-o disponível para uma vasta gama de aplicações, desde compressão de arquivos em sistemas de arquivos até transmissão de dados em tempo real. A combinação de eficiência, confiabilidade e ampla disponibilidade consolidou o GZIP como uma das soluções de compressão mais utilizadas e respeitadas na computação moderna.

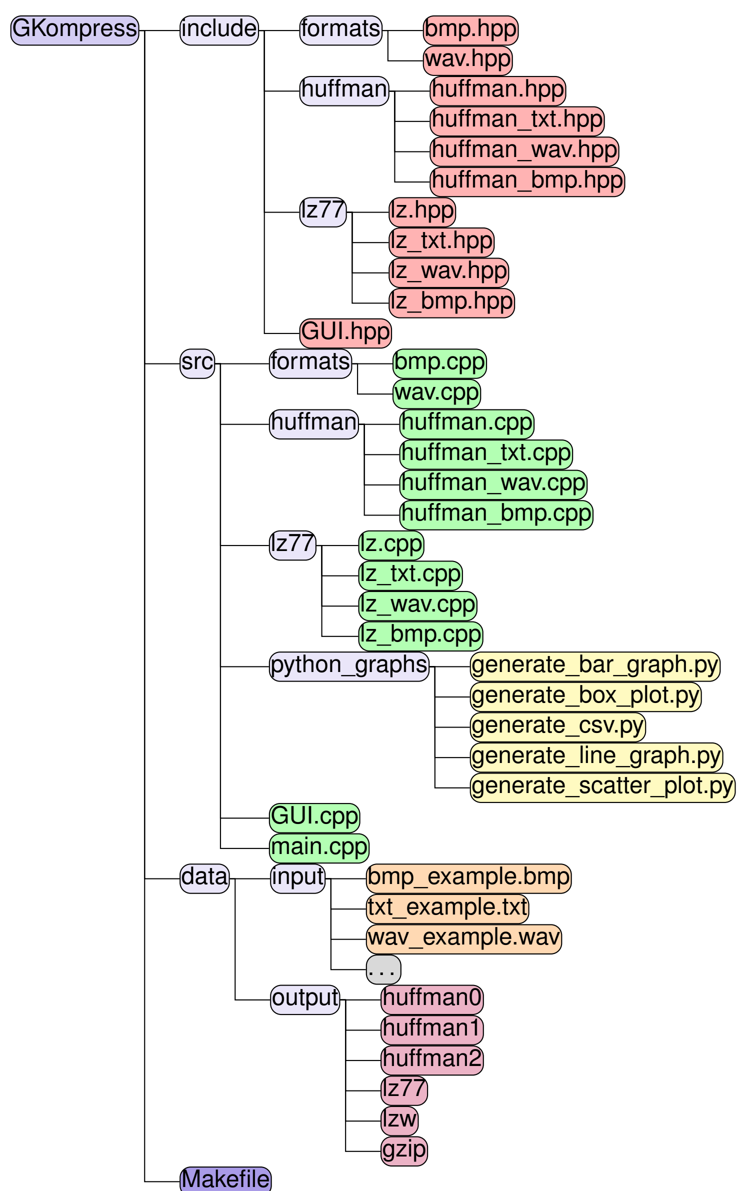
Com os fundamentos teóricos estabelecidos neste capítulo, o Capítulo 3 apresentará a implementação prática dos algoritmos de compressão estudados, detalhando as estruturas de dados desenvolvidas, as classes implementadas para manipulação dos diferentes formatos de arquivo, e os aspectos técnicos das implementações de Huffman, LZ77, LZW e GZIP em linguagem C++.

3 Implementação

Este capítulo apresenta a implementação dos principais componentes desenvolvidos no projeto, abordando desde a manipulação dos *headers* dos formatos BMP e WAV até a construção das classes e algoritmos responsáveis pela compressão e descompressão de dados. São detalhados os módulos referentes à leitura e interpretação de arquivos, a classe auxiliar *BitStream* para manipulação de bits, e os algoritmos de compressão Huffman, LZ77, LZW e GZIP, explicando suas estruturas, funcionamento e integração com a *API* desenvolvida.

A estrutura geral do projeto é apresentada na Figura 15, que ilustra a organização dos diretórios e arquivos principais. O projeto está organizado em três diretórios principais: `include/`, que contém os cabeçalhos das classes; `src/`, que contém as implementações dos algoritmos e utilitários; e `data/`, que organiza os arquivos de entrada e saída dos testes de compressão.

Figura 15 – Estrutura do projeto GKompress



Fonte: Elaborada pelo autor

O diretório `include/` contém os cabeçalhos organizados por funcionalidade: `formats/` para as classes de manipulação de formatos de arquivo (BMP e WAV), `huffman/` e `lz77/` para os algoritmos de compressão, e `GUI.hpp` para a interface gráfica. O diretório `src/` espelha essa organização, contendo as implementações correspondentes, além de utilitários Python para geração de gráficos de análise de desempenho.

O diretório `data/` é dividido em `input/`, contendo os arquivos de teste em diferentes formatos, e `output/`, organizado por algoritmo de compressão, onde são armazenados os resultados das operações de compressão e descompressão. Esta organização facilita a comparação de resultados e a análise de desempenho dos

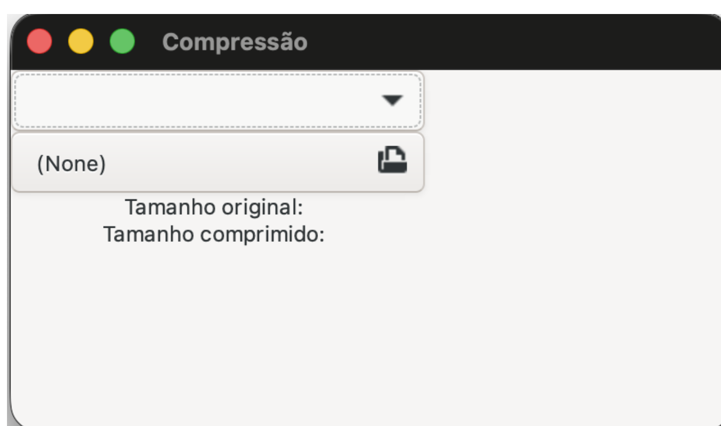
diferentes algoritmos implementados.

3.1 Interface Gráfica

O sistema GKompress inclui uma interface gráfica desenvolvida para facilitar a utilização dos algoritmos de compressão pelos usuários. A interface permite a seleção de arquivos, escolha do algoritmo de compressão desejado e visualização dos resultados obtidos.

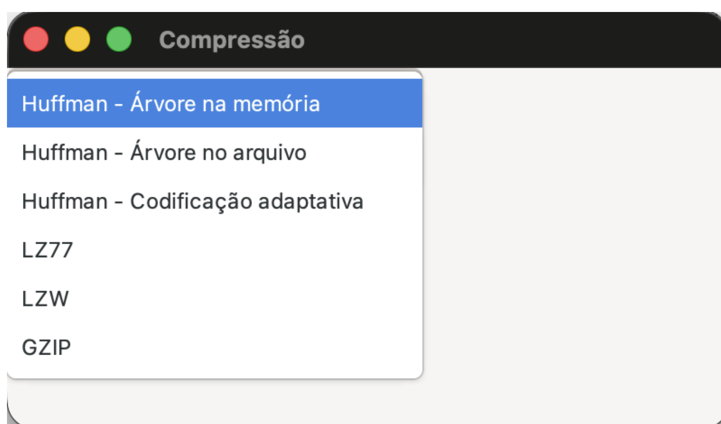
A interface principal do sistema é apresentada nas Figuras 16 e 17, mostrando a tela inicial onde o usuário pode navegar e selecionar os arquivos a serem processados.

Figura 16 – Interface principal do sistema GKompress



Fonte: Elaborada pelo autor

Figura 17 – Tela de processamento e configuração dos algoritmos



Fonte: Elaborada pelo autor

3.2 Manipulação de *Headers* BMP e WAV

Durante o desenvolvimento deste trabalho, foi necessário lidar com diferentes tipos de arquivos como entrada para os algoritmos de compressão e descompressão. Um ponto importante a ser considerado é a estrutura interna desses arquivos, especialmente no que diz respeito à presença ou ausência de cabeçalhos (*headers*).

Arquivos de texto puro, com extensão `.txt`, são compostos exclusivamente por dados representados em formato ASCII ou Unicode e, portanto, não possuem um cabeçalho estruturado. Esses arquivos não contêm metadados incorporados sobre seu conteúdo, formato ou estrutura, o que simplifica o processo de compressão, já que todos os dados podem ser processados diretamente.

Entretanto, o mesmo não ocorre com arquivos dos formatos `.bmp` (*bitmap image*) e `.wav` (*waveform audio file format*). Esses formatos binários possuem estruturas de cabeçalho bem definidas que armazenam metadados essenciais para a correta interpretação dos dados contidos no arquivo. No caso de arquivos BMP, o cabeçalho armazena informações como o tamanho da imagem, profundidade de cor, largura, altura, tipo de compressão se houver, entre outras, conforme ilustrado na Figura 18.

Figura 18 – *Header* do arquivo BMP

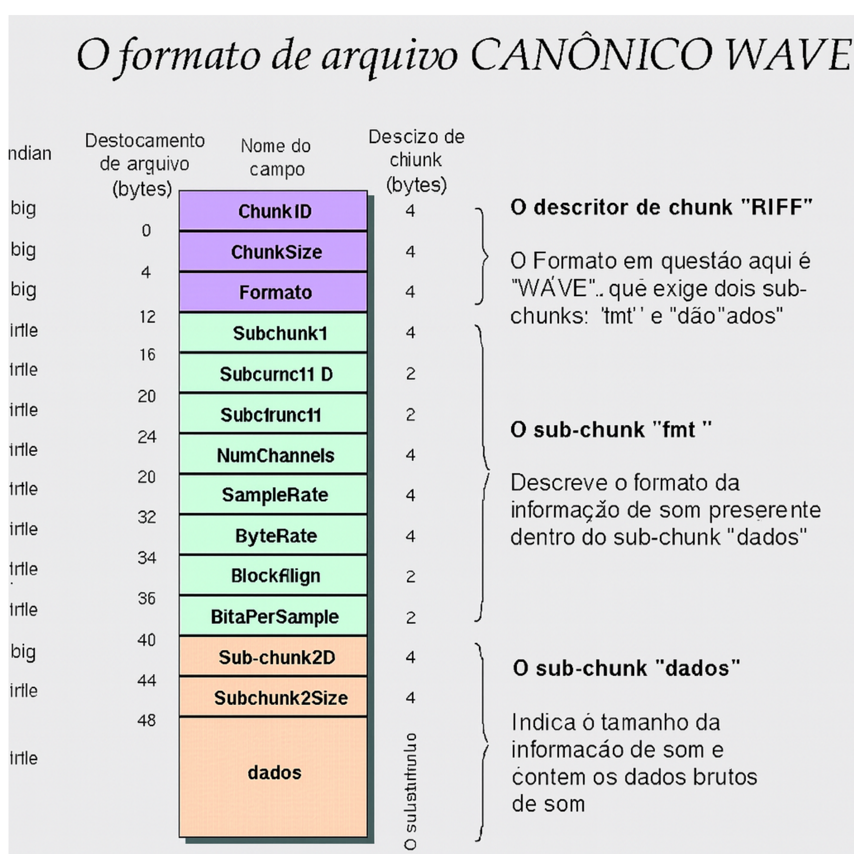
Formato Básico de Arquivo BMP

Nome	Tamanho	Descripção
Header	14 bytes	Windows Structure: BITMAPFILEHEADER
Signature	2 bytes	'BM'
FileSize	4 bytes	File size in bytes
reserved	4 bytes	unused (=0)
DataOffset	4 bytes	File offset to Raster Data
infoHeader	40 bytes	Windows Structure: BITMAPINFOHEADER
Size	4 bytes	Size of InfoHeeder =40
Width	4 bytes	Bitmap Width
Height	4 bytes	Bitmap Height
Planes	2 bytes	Number of Planes (=1)
BitCount	2 bytes	Bits per Pixel 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65556 (?) 24 = 24bit RGB. NumColors = 16M
Compression	4 bytes	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
ImageSize	4 bytes	(compressed) Size of Image It is valid to set this =0 if Compression = 0
XpixelsPerM	4 bytes	horizontal resolution: Pixels/meter
YpixelsPerM	4 bytes	vertical resolution: Pixels/meter
ColorsUsed	4 bytes	Number of actually used colors
ColorsImportant	4 bytes	Number of important colors 0 = all
ColorTable	4 * NumColors bytes	present only if Info BitsPerPixel <= 8 colors should be ordered by importance
Red	1 byte	Red intensity
Green	1 byte	Green intensity
Blue	1 byte	Blue intensity
reserved	1 byte	unused (=0)
repeated NumColors times		
Raster Data	Info ImageSize bytes	The pixel data

Fonte: (Department of Microelectronic Systems, 2025)

Já no caso dos arquivos WAV, o cabeçalho especifica dados como a taxa de amostragem, número de canais, taxa de bits, tamanho do arquivo, e o formato do áudio (PCM, por exemplo), conforme mostrado na Figura 19.

Figura 19 – *Header* do arquivo WAV



Fonte: (SAPP, 2025)

Durante o processo de compressão implementado neste trabalho, optou-se por não aplicar compressão sobre os dados de cabeçalho desses arquivos. Essa decisão foi tomada porque o cabeçalho geralmente ocupa uma pequena fração do tamanho total do arquivo, e sua compressão traria benefícios marginais. Além disso, preservar o cabeçalho intacto facilita o processo de descompressão e garante que o arquivo reconstruído mantenha sua estrutura original, permitindo sua correta abertura por softwares padrão após a descompressão.

Dessa forma, a implementação realiza a leitura e extração do cabeçalho no momento da compressão e o armazena separadamente do corpo dos dados. Ao descomprimir o arquivo, o cabeçalho original é restaurado no início do arquivo reconstruído, seguido pelos dados descomprimidos. Esse procedimento assegura que os arquivos BMP e WAV mantêm sua integridade e permanecem compatíveis com os leitores padrão desses formatos.

A manipulação correta dos *headers* foi essencial para garantir a funcionalidade completa do sistema de compressão e descompressão implementado. Sem esse cuidado, os arquivos descomprimidos poderiam se tornar ilegíveis ou corrompidos, uma vez que muitos *softwares* dependem da presença e exatidão do cabeçalho para

interpretar o conteúdo do arquivo de forma apropriada.

3.2.1 Implementação da Classe BMP

Para manipular arquivos BMP, foi implementada uma classe específica que encapsula toda a estrutura do formato, incluindo *headers*, tabela de cores e dados da imagem. A implementação utiliza estruturas com alinhamento de bytes controlado através da diretiva `#pragma pack` para garantir que os dados sejam lidos corretamente do arquivo binário. A estrutura da classe BMP é apresentada no Código 1.

Código 1 – Estrutura da classe BMP

```
#pragma pack(push,1)
class bmp {
public:
    struct HEADER {
        char signature[2];
        uint32_t file_size;
        uint32_t reserved;
        uint32_t data_offset;
    } header;
    struct INFO_header {
        uint32_t size;
        uint32_t width;
        uint32_t height;
        uint16_t planes;
        uint16_t bits_per_pixel;
        uint32_t compression;
        uint32_t image_size;
        uint32_t x_pixels_per_m;
        uint32_t y_pixels_per_m;
        uint32_t colors_used;
        uint32_t important_colors;
    } info_header;
    uint8_t *color_table; uint8_t *data; size_t data_size;
    int read(const char* filename);
    void print();
    uint32_t get_color_table_size();
    int has_color_table();
private:
    void read_bmp_data(FILE* file);
};
#pragma pack(pop)
```

Fonte: Elaborado pelo autor

A classe BMP implementa métodos essenciais para a leitura e manipulação de arquivos *bitmap*. O método `read()` é responsável por abrir o arquivo e ler sequencialmente todos os componentes: *header* principal, *info header*, tabela de cores (quando aplicável) e dados da imagem. A verificação da existência de tabela de cores é feita através do método `has_color_table()`, que retorna verdadeiro quando o número de bits por pixel é menor ou igual a 8.

O método `read_bmp_data()` implementa uma estratégia de alocação dinâmica de memória que utiliza o padrão de *table doubling* para otimizar o uso de memória durante a leitura dos dados da imagem. Essa abordagem evita a necessidade de conhecer previamente o tamanho dos dados e permite a leitura eficiente de arquivos de qualquer tamanho. A implementação deste método é mostrada no Código 2.

Código 2 – Método de leitura de dados da imagem BMP

```
size_t buffer_size = 1024;
size_t var_data_size = 0;
uint8_t* var_data = new uint8_t[buffer_size];
if(!var_data) {
    fclose(file); EXIT_WITH_ERROR("Memory allocation failed");
}
size_t bytes_read;
while((bytes_read = fread(var_data + var_data_size, 1,
    buffer_size-var_data_size, file)) > 0) {
    var_data_size += bytes_read;
    if(var_data_size >= buffer_size) {  // table doubling
        buffer_size *= 2;
        uint8_t* new_data = (uint8_t*)realloc(var_data, buffer_size);
        if(!new_data) {
            free(var_data); fclose(file); EXIT_WITH_ERROR("Memory
                allocation failed");
        }
        var_data = new_data;
    }
}
data_size = var_data_size;

data = (uint8_t*)malloc(data_size);
if(!data) {
    free(data); fclose(file); EXIT_WITH_ERROR("Memory allocation
        failed");
}
memcpy(data, var_data, data_size);
free(var_data);
```

Fonte: Elaborado pelo autor

3.2.2 Implementação da Classe WAV

Similarmente à classe BMP, a classe WAV foi implementada para manipular arquivos de áudio no formato WAVE PCM. A estrutura segue o padrão RIFF (*Resource Interchange File Format*), que organiza os dados em *chunks* identificados por *tags* de 4 caracteres. A estrutura da classe WAV é apresentada no Código 3.

Código 3 – Estrutura da classe WAV

```
#pragma pack(push,1)
class wav {
public:
    struct RIFF_chunk_descriptor {
        char ID[4];
        uint32_t size;
        char format[4];
    } riff_chunk_descriptor;

    struct FMT_subchunk {
        char ID[4];
        uint32_t size;
        uint16_t audio_format;
        uint16_t num_channels;
        uint32_t sample_rate;
        uint32_t byte_rate;
        uint16_t block_align;
        uint16_t bits_per_sample;
    } fmt_subchunk;

    struct DATA_subchunk {
        char ID[4];
        uint32_t size;
    } data_subchunk;
    uint8_t* data;

    int read(std::string filename);
    void print();
};
#pragma pack(pop)
```

Fonte: Elaborado pelo autor

A implementação do método de leitura para arquivos WAV segue uma sequência específica: primeiro lê o *descriptor* RIFF, depois o *subchunk* de formato (fmt) e finalmente o *subchunk* de dados (data). Cada *chunk* é identificado por sua *tag* de 4 caracteres, garantindo que a estrutura do arquivo seja respeitada.

Ambas as classes implementam métodos de *debug* (`print()`) que exibem todas as informações dos *headers* em formato hexadecimal, facilitando a verificação da integridade dos dados lidos e o desenvolvimento de testes para validar a implementação.

3.3 Classe *BitStream*

A classe *BitStream* é um componente fundamental na implementação dos algoritmos de compressão, especialmente para aqueles que trabalham com códigos de comprimento variável, como o LZW. Esta classe encapsula a funcionalidade de leitura e escrita de bits individuais, permitindo que os algoritmos manipulem dados em nível de bit, o que é essencial para a eficiência da compressão.

É importante destacar que a maioria das linguagens de programação modernas, incluindo C e C++, são orientadas a bytes, ou seja, a menor unidade de dados que podem manipular diretamente é o byte (8 bits). Em C, por exemplo, o tipo de dados mínimo é o `char`, que representa exatamente um byte. Isso representa um desafio significativo para algoritmos de compressão que precisam trabalhar com códigos de comprimento variável, em que diferentes símbolos podem ser representados por sequências de bits de tamanhos distintos.

A implementação da *BitStream* é realizada através de duas classes auxiliares: *BitWriter* e *BitReader*, que são classes internas da classe LZW. Essas classes gerenciam um *buffer* interno que acumula bits antes de escrevê-los ou lê-los do arquivo, otimizando as operações de I/O.

3.4 Classe *BitWriter*

A classe *BitWriter* é responsável pela escrita de bits individuais e sequências de bits em arquivos. Ela mantém um *buffer* interno de 32 bits e um contador de bits, permitindo a escrita eficiente de códigos de comprimento variável. A estrutura da classe BitWriter é apresentada no Código 4.

Código 4 – Estrutura da classe BitWriter

```
class BitWriter {
private:
    FILE* file;
    uint32_t buffer = 0;
    int bitCount = 0;
public:
    BitWriter(FILE* f) : file(f) {}
    void writeBit(bool bit);
    void writeBits(uint32_t value, int bits);
    void flush();
    void finalize();
};
```

Fonte: Elaborado pelo autor

O método `writeBit()` adiciona um bit ao *buffer* interno, deslocando o *buffer* para a esquerda e inserindo o novo bit na posição menos significativa. Quando o *buffer* atinge 8 bits, ele é automaticamente escrito no arquivo através do método `flush()`. A implementação deste método é mostrada no Código 5.

Código 5 – Implementação do método writeBit

```
void LZW::BitWriter::writeBit(bool bit) {
    buffer = (buffer << 1) | (bit ? 1 : 0);
    bitCount++;
    if (bitCount == 8) flush();
}
```

Fonte: Elaborado pelo autor

O método `writeBits()` permite escrever múltiplos bits de uma vez, iterando sobre cada bit do valor fornecido e chamando `writeBit()` para cada um. Isso é especialmente útil para escrever códigos de Huffman ou códigos LZW que podem ter diferentes comprimentos. A implementação deste método é mostrada no Código 6.

Código 6 – Implementação do método writeBits

```
void LZW::BitWriter::writeBits(uint32_t value, int bits) {
    for (int i = bits - 1; i >= 0; i--) {
        bool bit = (value >> i) & 1;
        writeBit(bit);
    }
}
```

Fonte: Elaborado pelo autor

O método `flush()` escreve os bits completos do *buffer* no arquivo, mantendo apenas os bits restantes no *buffer*. O método `finalize()` garante que todos os bits restantes sejam escritos no arquivo, preenchendo com zeros se necessário.

3.5 Classe *BitReader*

A classe *BitReader* é responsável pela leitura de bits individuais e sequências de bits de arquivos. Ela mantém um *buffer* interno e um contador de bits, permitindo a leitura eficiente de códigos de comprimento variável. A estrutura da classe BitReader é apresentada no Código 7.

Código 7 – Estrutura da classe BitReader

```
class BitReader {
private:
    FILE* file;
    uint32_t buffer = 0;
    int bitCount = 0;
    int currentCodeWidth = INITIAL_CODE_WIDTH;
public:
    BitReader(FILE* f, int initialWidth) : file(f),
                                    currentCodeWidth(initialWidth) {}
    bool readBit();
    uint32_t readBits(int bits);
    void setCodeWidth(int width) { currentCodeWidth = width; }
};
```

Fonte: Elaborado pelo autor

O método `readBit()` lê um bit do arquivo, carregando um novo byte no *buffer* quando necessário. Ele retorna `false` quando atinge o final do arquivo, permitindo o controle adequado do fluxo de leitura. A implementação deste método é mostrada no Código 8.

Código 8 – Implementação do método readBit

```
bool LZW::BitReader::readBit() {
    if (bitCount == 0) {
        uint8_t byte;
        if (fread(&byte, 1, 1, file) != 1) {
            if (feof(file)) {
                return false;
            }
            EXIT_WITH_ERROR("Error reading byte in BitReader");
        }
```

```
10          buffer = byte;
11          bitCount = 8;
12      }
13      bool bit = (buffer >> (bitCount - 1)) & 1;
14      bitCount--;
15      return bit;
16  }
```

Fonte: Elaborado pelo autor

O método `readBits()` permite ler múltiplos bits de uma vez, construindo um valor inteiro a partir dos bits lidos. Isso é essencial para a decodificação de códigos de comprimento variável. A implementação deste método é mostrada no Código 9.

Código 9 – Implementação do método readBits

```
1  uint32_t LZW::BitReader::readBits(int bits) {
2      uint32_t value = 0;
3      for (int i = 0; i < bits; i++) {
4          bool bit = readBit();
5          value = (value << 1) | (bit ? 1 : 0);
6      }
7      return value;
8  }
```

Fonte: Elaborado pelo autor

3.6 Importância da Classe *BitStream*

A implementação da *BitStream* é crucial para a eficiência dos algoritmos de compressão implementados. Sem essa funcionalidade, seria necessário trabalhar apenas com bytes completos, resultando em desperdício significativo de espaço.

A capacidade de manipular bits individuais permite que os algoritmos de compressão atinjam suas taxas de compressão teóricas, aproximando-se do limite estabelecido pela entropia da fonte. Além disso, a implementação otimizada com *buffers* internos minimiza o número de operações de I/O, melhorando significativamente o desempenho do sistema de compressão.

3.7 Huffman

O algoritmo de Huffman foi implementado utilizando uma abordagem personalizada que difere significativamente das implementações convencionais. Em vez de utilizar estruturas de dados padrão como *priority queues* ou *heaps*, foi desenvolvida

uma solução integrada que combina eficiência computacional com simplicidade de implementação.

A implementação do Huffman baseia-se em três estruturas de dados principais, todas desenvolvidas especificamente para este projeto. Em vez de utilizar uma *priority queue* padrão, foi implementada uma lista encadeada ordenada (`List`) que serve simultaneamente como fila de prioridade e como base para a construção da árvore de Huffman. Esta abordagem apresenta economia de memória, pois a mesma estrutura serve para ordenar os nós e construir a árvore, oferece controle total sobre a manipulação precisa da ordem de inserção e remoção, e elimina a necessidade de bibliotecas externas ou estruturas complexas. A lista mantém os nós ordenados por frequência em ordem crescente, utilizando o método `insert_sorted()` que percorre a lista até encontrar a posição correta para inserção.

A classe `HuffmanTree` é construída diretamente a partir da lista ordenada, utilizando os nós já presentes na memória. Durante o processo de construção, os dois nós de menor frequência são removidos da lista e combinados em um novo nó interno, que é reinserido na posição apropriada. Este processo continua até restar apenas um nó, que se torna a raiz da árvore.

O `Dictionary` armazena os códigos de Huffman gerados para cada símbolo. A implementação suporta três variantes de codificação, cada uma correspondendo a uma implementação específica do algoritmo:

a) **Huffman 1 (*Tree on memory*)**: Codificação estática, onde os códigos são gerados diretamente da árvore mantida em memória durante a compressão;
b) **Huffman 2 (*Tree on decompressed file*)**: Codificação com árvore, onde a árvore é salva junto com os dados comprimidos para permitir a reconstrução durante a descompressão; e
c) **Huffman 3 (*Canonical coding*)**: Codificação canônica, que utiliza códigos canônicos para otimizar o armazenamento dos metadados.

O processo de compressão segue uma sequência específica de etapas. Primeiro, cada byte do arquivo é contado e armazenado no *array* `duplicates`. Em seguida, símbolos com frequência não-nula são inseridos na lista ordenada. Posteriormente, os dois nós de menor frequência são combinados iterativamente até formar a árvore completa. A árvore é então percorrida para gerar os códigos binários, os dados originais são substituídos pelos códigos de Huffman, e finalmente os códigos são empacotados em bytes para armazenamento através de compressão *bit-a-bit*.

A implementação para arquivos de texto (`HuffmanTXT`) apresenta algumas particularidades. O arquivo é lido completamente em memória como *array* de bytes,

utiliza o caractere nulo ('\0') para determinar o final dos dados, e não requer cabeçalhos especiais de formato, simplificando a codificação.

Para arquivos de imagem BMP (`HuffmanBMP`), a implementação preserva os cabeçalhos BMP intactos no arquivo comprimido, comprime somente os dados da imagem, e trata adequadamente BMPs com paletas de cores através de suporte específico para tabelas de cores.

A implementação para arquivos de áudio WAV (`HuffmanWAV`) segue padrões similares, mantendo os cabeçalhos RIFF, fmt e data subchunks, submetendo apenas os dados de áudio à compressão, e preservando as informações sobre formato, taxa de amostragem e canais.

Para a variante de codificação canônica, foi implementado um algoritmo que ordena os símbolos primeiro por comprimento do código e depois por valor, gera códigos canônicos sequenciais usando operações *bit-shift*, reduzindo significativamente o tamanho dos metadados salvos.

A implementação inclui cuidados especiais para gerenciamento de memória. As classes possuem destrutores que liberam memória automaticamente, o método `clean_for_next_iteration()` prepara as estruturas para nova compressão, e todas as alocações dinâmicas possuem desalocação correspondente para controle de vazamentos.

A implementação do algoritmo de Huffman utiliza uma estrutura de nó personalizada que combina os dados do símbolo com ponteiros para a árvore e para a lista ordenada. A estrutura do nó é apresentada no Código 10.

Código 10 – Estrutura do nó de Huffman

```
1  class Node {
2  public:
3      uint8_t byte;
4      uint32_t duplicates;
5      Node* left;
6      Node* right;
7      Node* next;
8
9      int is_leaf();
10     Node(uint8_t b, uint32_t dup) : byte(b), duplicates(dup),
           left(nullptr), right(nullptr), next(nullptr) {}
11 };
```

Fonte: Elaborado pelo autor

A construção da árvore de Huffman é realizada através do método `build()` da

classe `HuffmanTree`, que utiliza a lista ordenada para combinar iterativamente os nós de menor frequência. O processo de construção é mostrado no Código 11.

Código 11 – Construção da árvore de Huffman

```
void HuffmanTree::build(List& list) {
    Node *first_removed, *second_removed, *new_node;
    while(list.size > 1) {
        first_removed = list.remove_front();
        second_removed = list.remove_front();
        new_node = new Node('*',
            first_removed->duplicates + second_removed->duplicates);
        new_node->left = first_removed;
        new_node->right = second_removed;
        new_node->next = nullptr;

        list.insert_sorted(new_node);
    }
    root = list.front;
}
```

Fonte: Elaborado pelo autor

A geração dos códigos de Huffman é implementada através do método `fill()` da classe `Dictionary`, que percorre recursivamente a árvore construída e atribui códigos binários aos símbolos. A implementação é mostrada no Código 12.

Código 12 – Geração dos códigos de Huffman

```
void Dictionary::fill(Node* tree_root, char* code,
                                    size_t columns, int option) {
    char left[columns+1], right[columns+1];
    if(tree_root->left == nullptr && tree_root->right == nullptr) {
        dictionary[tree_root->byte] = strdup(code);
        if(option == 2)
            sorted_symbols.push_back({tree_root->byte,
                strlen(code)});
    } else {
        strcpy(left, code); strcpy(right, code);
        strcat(left, "0"); strcat(right, "1");
        fill(tree_root->left, left, columns, option);
        fill(tree_root->right, right, columns, option);
    }
}
```

Fonte: Elaborado pelo autor

O processo de codificação dos dados é implementado através do método `encode()`, que substitui cada byte dos dados originais pelo seu código de Huffman correspondente. A implementação é mostrada no Código 13.

Código 13 – Codificação dos dados com códigos de Huffman

```
1  char* Huffman::encode(uint8_t* data, size_t data_size) {
2      size_t i, size=0;
3      for(i=0; i<data_size; i++) {
4          size += strlen(dictionary[data[i]]);
5      }
6
7      char* code = new char[size]();
8      char* ptr = code;
9      for(i=0; i<data_size; i++) {
10         ptr = strcat_endpointer(ptr, dictionary[data[i]]);
11     }
12
13     return code;
14 }
```

Fonte: Elaborado pelo autor

A compressão final dos dados é realizada através do método `internal_compress()`, que empacota os códigos binários em bytes completos e adiciona metadados necessários para a descompressão. O processo de empacotamento *bit-a-bit* é mostrado no Código 14.

Código 14 – Empacotamento *bit-a-bit* dos códigos de Huffman

```
1  void Huffman::internal_compress(int option, char* code,
2                                  void (*write_header)(FILE*, void*),
3                                  void* filetype) {
4      FILE *file = fopen(compressed_filepath.c_str(), "wb");
5      char byte = 0;
6
7      if(option == 1)
8          tree.write(file, tree.root);
9      else if(option == 2)
10         dictionary.write(file);
11
12     if(write_header && filetype)
13         write_header(file, filetype);
14
15     int j=7;
16     for(size_t i=0; code[i] != '\0'; i++) {
```

```
17            if(code[i] == '1')
18                byte |= (1U << j);
19            j--;
20            if(j < 0) {
21                fwrite(&byte, sizeof(char), 1, file);
22                byte = 0;
23                j = 7;
24            }
25        }
26        if(j != 7) {
27            fwrite(&byte, sizeof(char), 1, file);
28            byte = 7-j;
29            fwrite(&byte, sizeof(char), 1, file);
30        }
31        fclose(file);
32    }
```

Fonte: Elaborado pelo autor

A implementação para diferentes tipos de arquivo segue um padrão consistente. Para arquivos BMP, o método `compress()` da classe `HuffmanBMP` preserva os headers do formato bitmap e aplica compressão apenas aos dados da imagem, conforme mostrado no Código 15.

Código 15 – Compressão de arquivos BMP com Huffman

```
1  void HuffmanBMP::compress(int option) {
2      bmp_file.read(filepath.c_str());
3      count_duplicates(bmp_file.data, bmp_file.data_size);
4      list.fill(duplicates);
5      tree.build(list);
6      size_t max_columns_of_dictionary = tree.height();
7      dictionary.build(max_columns_of_dictionary);
8      dictionary.fill(tree.root, "", max_columns_of_dictionary,
           option);
9      if(option == 2) {
10         dictionary.generate_canonical_codes();
11     }
12     char* code = encode(bmp_file.data, bmp_file.data_size);
13     internal_compress(option, code, write_bmp_header, &bmp_file);
14 }
```

Fonte: Elaborado pelo autor

Esta implementação personalizada do algoritmo de Huffman demonstra como estruturas de dados simples e bem projetadas podem oferecer *performance* comparável a implementações mais complexas, mantendo a simplicidade e facilitando a

manutenção do código. A arquitetura orientada a objetos permite extensibilidade para diferentes formatos de arquivo, enquanto as classes auxiliares garantem eficiência na manipulação de bits e gerenciamento de memória.

3.8 LZ77

O algoritmo LZ77 foi implementado seguindo uma arquitetura orientada a objetos que permite a extensão para diferentes tipos de arquivo. A implementação baseia-se em uma classe abstrata `LZ77` que define a interface comum para compressão e descompressão, com classes derivadas especializadas para cada tipo de arquivo: `LZ77TXT` para arquivos de texto, `LZ77BMP` para imagens bitmap e `LZ77WAV` para arquivos de áudio.

A estrutura da classe base `LZ77` é apresentada no Código 16. A implementação utiliza duas classes auxiliares internas, `BitWriter` e `BitReader`, para manipulação eficiente de bits, permitindo a codificação compacta das referências de deslocamento e comprimento. Os parâmetros principais do algoritmo são definidos como constantes estáticas: `WINDOW_SIZE` de 4096 bytes para o buffer de busca e `MAX_LENGTH` de 63 bytes para o comprimento máximo de correspondência.

Código 16 – Estrutura da classe base LZ77

```
class LZ77 {
protected:
    std::string filepath;
    static const int WINDOW_SIZE = 4096;
    static const int MAX_LENGTH = 63;

    class BitWriter {
    private:
        FILE* file;
        uint32_t buffer = 0;
        int bitCount = 0;
    public:
        BitWriter(FILE* f) : file(f) {}
        void writeBit(bool bit);
        void writeBits(uint32_t value, int bits);
        void flush();
        void finalize();
    };
    class BitReader {
    private:
        FILE* file;
        uint32_t buffer = 0;
```

```
23          int bitCount = 0;
24      public:
25          BitReader(FILE* f) : file(f) {};
26          bool readBit();
27          uint32_t readBits(int bits);
28      };
29
30      void internal_compress(uint8_t* data, size_t data_size,
31                             void (*write_header)(FILE*, void*) =
                                  nullptr,
32                             void* filetype = nullptr);
33  public:
34      LZ77() : filepath("") {}
35      LZ77(std::string fp) : filepath(fp) {}
36      int set_filepath(const std::string& path);
37      virtual void compress(int option) = 0;
38      int decompress(int option, void (*write_header)(FILE*, void*) =
            nullptr,
39                     void (*read_header)(FILE*, void*) = nullptr,
40                     size_t (*get_pos)(void*) = nullptr,
41                     void* filetype = nullptr);
42  };
```

Fonte: Elaborado pelo autor

O algoritmo de compressão principal é implementado no método `internal_compress()`, que recebe os dados a serem comprimidos e funções opcionais para manipulação de headers específicos de cada formato de arquivo. O processo de compressão segue os seguintes passos: inicialização do arquivo de saída com cabeçalho personalizado, escrita dos headers específicos do formato (quando aplicável), e processamento sequencial dos dados utilizando a janela deslizante.

A implementação do algoritmo de busca por correspondências é mostrada no Código 17. Para cada posição no buffer de antecipação, o algoritmo procura a maior subsequência que tenha correspondência no buffer de busca, limitada pelo tamanho da janela e pelo comprimento máximo de correspondência. Quando uma correspondência de pelo menos 3 bytes é encontrada, ela é codificada como um triplo (flag, deslocamento, comprimento, próximo caractere).

Código 17 – Algoritmo de busca por correspondências no LZ77

```
while (pos < data_size) {
    int bestOffset = 0, bestLength = 0;
    int windowStart = std::max(0, static_cast<int>(pos) -
        WINDOW_SIZE);

    for (int i = windowStart; i < static_cast<int>(pos); ++i) {
        int length = 0;
        while (length < MAX_LENGTH && pos + length < data_size &&
               data[i + length] == data[pos + length]) {
            ++length;
        }
        if (length > bestLength && length >= 3) {
            bestLength = length;
            bestOffset = pos - i;
        }
    }

    if (bestLength >= 3) {
        writer.writeBit(1);
        writer.writeBits(bestOffset, 12);
        writer.writeBits(bestLength, 6);
        uint8_t nextChar = (pos + bestLength < data_size) ?
                          data[pos + bestLength] : 0;
        writer.writeBits(nextChar, 8);
        pos += bestLength + 1;
    } else {
        writer.writeBit(0);
        writer.writeBits(data[pos], 8);
        pos += 1;
    }
}
```

Fonte: Elaborado pelo autor

A codificação das referências utiliza uma representação compacta que combina bits de controle com dados. O primeiro bit indica se o elemento é uma referência (1) ou um literal (0). Para referências, seguem-se 12 bits para o deslocamento (permitindo endereçar até 4096 bytes para trás), 6 bits para o comprimento (até 63 bytes) e 8 bits para o próximo caractere após a correspondência. Para literais, apenas 8 bits são necessários para representar o caractere original.

O processo de descompressão é implementado de forma inversa, lendo os bits de controle e reconstruindo os dados originais. O algoritmo mantém um *buffer* de saída que é preenchido sequencialmente, expandindo as referências quando encontradas. A implementação da descompressão é apresentada no Código 18.

Código 18 – Algoritmo de descompressão do LZ77

```
while (buffer.size() < origSize) {
    bool flag = reader.readBit();
    if (flag) {
        uint32_t offset = reader.readBits(12);
        uint32_t length = reader.readBits(6);
        uint8_t nextChar = reader.readBits(8);
        size_t start = buffer.size() - offset;
        for (int i = 0; i < length; ++i) {
            buffer.push_back(buffer[start + i]);
        }
        buffer.push_back(nextChar);
    } else {
        uint8_t ch = reader.readBits(8);
        buffer.push_back(ch);
    }
}
```

Fonte: Elaborado pelo autor

A implementação para diferentes tipos de arquivo segue um padrão consistente. Para arquivos de texto (`LZ77TXT`), o processo é direto, lendo todo o conteúdo do arquivo em memória e aplicando o algoritmo de compressão. Para arquivos BMP e WAV, a implementação preserva os headers específicos do formato, aplicando compressão apenas aos dados propriamente ditos.

A classe `LZ77BMP` utiliza funções auxiliares para manipulação dos headers BMP, incluindo `write_bmp_header()` para escrita e `read_bmp_header()` para leitura, garantindo que a estrutura do arquivo bitmap seja preservada durante o processo de compressão e descompressão. Similarmente, `LZ77WAV` implementa funções específicas para manipulação dos chunks RIFF, fmt e data dos arquivos de áudio.

A eficiência da implementação é otimizada através do uso das classes `BitWriter` e `BitReader`, que minimizam o número de operações de I/O ao trabalhar com buffers internos. O método `finalize()` do `BitWriter` garante que todos os bits restantes sejam escritos no arquivo, preenchendo com zeros quando necessário para alinhar com limites de byte.

Esta implementação do LZ77 demonstra como o algoritmo pode ser adaptado para diferentes tipos de arquivo mantendo a eficiência e a integridade dos dados. A arquitetura orientada a objetos permite extensibilidade e reutilização de código, enquanto as classes auxiliares de manipulação de bits garantem a eficiência na codificação das referências de compressão.

3.9 LZW

O algoritmo LZW (Lempel-Ziv-Welch) foi implementado seguindo uma arquitetura similar ao LZ77, com classes base e derivadas para diferentes tipos de arquivo. A implementação utiliza um dicionário dinâmico que cresce durante a compressão, permitindo a codificação de sequências de caracteres recorrentes em códigos de comprimento variável.

A estrutura da classe base `LZW` é apresentada no Código 19. A implementação mantém dois dicionários: um para compressão (string para código) e outro para descompressão (código para string). Os parâmetros principais incluem `MAX_DICT_SIZE` de 4096 entradas, `INITIAL_DICT_SIZE` de 256 (caracteres ASCII) e `INITIAL_CODE_WIDTH` de 9 bits para os códigos iniciais.

Código 19 – Estrutura da classe base LZW

```
1  class LZW {
2  protected:
3      std::string filepath;
4      static const size_t MAX_DICT_SIZE = 4096;
5      static const size_t INITIAL_DICT_SIZE = 256;
6      static const size_t INITIAL_CODE_WIDTH = 9;
7
8      std::map<std::string, uint16_t> compress_dict;
9      std::map<uint16_t, std::string> decompress_dict;
10
11     class BitWriter {
12     private:
13         FILE* file;
14         uint32_t buffer = 0;
15         int bitCount = 0;
16     public:
17         BitWriter(FILE* f) : file(f) {}
18         void writeBit(bool bit);
19         void writeBits(uint32_t value, int bits);
20         void flush();
21         void finalize();
22     };
23
24     class BitReader {
25     private:
26         FILE* file;
27         uint32_t buffer = 0;
28         int bitCount = 0;
29         int currentCodeWidth = INITIAL_CODE_WIDTH;
30     public:
```

```
31         BitReader(FILE* f, int initialWidth) : file(f),
32                                   currentCodeWidth(initialWidth) {}
33         bool readBit();
34         uint32_t readBits(int bits);
35         void setCodeWidth(int width) { currentCodeWidth = width; }
36     };
37
38     std::vector<std::pair<uint16_t, int>>
39             encode(const uint8_t* data, size_t size);
40     std::vector<uint8_t>
41             decode(const std::vector<std::pair<uint16_t, int>>&
                   codes);
42     void internal_compress(int option,
43                       const std::vector<std::pair<uint16_t, int>>&
                           codes,
44                       void (*write_header)(FILE*, void*) = nullptr,
45                       void* filetype = nullptr);
46
47 public:
48     LZW() : filepath("") {}
49     LZW(std::string fp) : filepath(fp) {}
50     int set_filepath(const std::string& path);
51     virtual void compress(int option) = 0;
52     virtual int decompress(int option) = 0;
53 };
```

Fonte: Elaborado pelo autor

O processo de codificação LZW é implementado no método `encode()`, que constrói o dicionário dinamicamente durante a compressão. O algoritmo processa sequencialmente os dados de entrada, mantendo uma *string* atual que é expandida com cada novo caractere. Quando uma nova sequência é encontrada, o código da *string* anterior é emitido e a nova sequência é adicionada ao dicionário. A implementação do algoritmo de codificação é mostrada no Código 20.

Código 20 – Algoritmo de codificação LZW

```
1 std::vector<std::pair<uint16_t, int>> LZW::encode(const uint8_t*
      data,
2                                                   size_t size) {
3     compress_dict.clear();
4     for (uint16_t i = 0; i < INITIAL_DICT_SIZE; ++i) {
5         compress_dict[std::string(1, static_cast<char>(i))] = i;
6     }
7
8     std::vector<std::pair<uint16_t, int>> codes;
9     std::string current;
```

```
10     uint16_t next_code = INITIAL_DICT_SIZE;
11     int current_bit_width = INITIAL_CODE_WIDTH;
12 
13     for (size_t i = 0; i < size; ++i) {
14         char ch = static_cast<char>(data[i]);
15         std::string next = current + ch;
16 
17         if (compress_dict.find(next) != compress_dict.end()) {
18             current = next;
19             continue;
20         }
21 
22         codes.emplace_back(compress_dict[current],
               current_bit_width);
23 
24         if (next_code >= MAX_DICT_SIZE - 1) {
25             codes.emplace_back(RESET_CODE, current_bit_width);
26             compress_dict.clear();
27             for (uint16_t j = 0; j < INITIAL_DICT_SIZE; ++j) {
28                 compress_dict[std::string(1,
                       static_cast<char>(j))]=j;
29             }
30             next_code = INITIAL_DICT_SIZE;
31             current_bit_width = INITIAL_CODE_WIDTH;
32             current = std::string(1, ch);
33             continue;
34         }
35 
36         compress_dict[next] = next_code++;
37 
38         if (next_code > (1 << current_bit_width)) {
39             current_bit_width++;
40         }
41 
42         current = std::string(1, ch);
43     }
44 
45     if (!current.empty()) {
46         codes.emplace_back(compress_dict[current],
               current_bit_width);
47     }
48 
49     return codes;
50 }
```

Fonte: Elaborado pelo autor

A descompressão LZW é implementada no método `decode()`, que reconstrói

o dicionário durante o processo de decodificação. O algoritmo lê códigos sequencialmente, consultando o dicionário para recuperar as *strings* correspondentes. Quando encontra um código que ainda não existe no dicionário, utiliza a propriedade especial do LZW onde o próximo código a ser adicionado corresponde à *string* anterior mais seu primeiro caractere. A implementação da descompressão é mostrada no Código 21.

Código 21 – Algoritmo de descompressão LZW

```
1  std::vector<uint8_t> LZW::decode(const
       std::vector<std::pair<uint16_t, int>>& codes) {
2      decompress_dict.clear();
3      for (uint16_t i = 0; i < INITIAL_DICT_SIZE; ++i) {
4          decompress_dict[i] = std::string(1, static_cast<char>(i));
5      }
6
7      std::vector<uint8_t> output;
8      uint16_t next_code = INITIAL_DICT_SIZE;
9      int current_bit_width = INITIAL_CODE_WIDTH;
10     std::string previous;
11
12     for (size_t i = 0; i < codes.size(); ++i) {
13         const auto& [code, bit_width] = codes[i];
14
15         if (code == RESET_CODE) {
16             decompress_dict.clear();
17             for (uint16_t j = 0; j < INITIAL_DICT_SIZE; ++j) {
18                 decompress_dict[j] = std::string(1,
                       static_cast<char>(j));
19             }
20             next_code = INITIAL_DICT_SIZE;
21             current_bit_width = INITIAL_CODE_WIDTH;
22             previous.clear();
23             continue;
24         }
25
26         std::string entry;
27         if (decompress_dict.count(code)) {
28             entry = decompress_dict[code];
29         } else if (code == next_code && !previous.empty()) {
30             entry = previous + previous[0];
31         } else {
32             EXIT_WITH_ERROR("Invalid LZW code");
33         }
34
35         output.insert(output.end(), entry.begin(), entry.end());
36
```

```
37         if (!previous.empty() && next_code < MAX_DICT_SIZE) {
38             decompress_dict[next_code++] = previous + entry[0];
39             if (next_code > (1 << current_bit_width)) {
40                 current_bit_width++;
41             }
42         }
43 
44         previous = entry;
45     }
46 
47     return output;
48 }
```

Fonte: Elaborado pelo autor

A implementação para diferentes tipos de arquivo segue o mesmo padrão dos outros algoritmos. Para arquivos WAV, o método `compress()` da classe `LZW_WAV` preserva os *headers* do formato de áudio e aplica compressão apenas aos dados de áudio, conforme mostrado no Código 22.

Código 22 – Compressão de arquivos WAV com LZW

```
1 void LZW_WAV::compress(int option) {
2     wav_file.read(filepath);
3     std::vector<std::pair<uint16_t, int>> codes =
          encode(wav_file.data, wav_file.data_subchunk.size);
4     internal_compress(option, codes, write_wav_header, &wav_file);
5 }
```

Fonte: Elaborado pelo autor

A implementação do LZW apresenta características únicas em relação aos outros algoritmos. O uso de códigos de comprimento variável permite que o algoritmo se adapte dinamicamente ao conteúdo dos dados, aumentando a largura de bits conforme o dicionário cresce. O mecanismo de reset do dicionário garante que o algoritmo não consuma memória excessiva em arquivos grandes, reinicializando o dicionário quando atinge o tamanho máximo.

Esta implementação do LZW demonstra como algoritmos de compressão baseados em dicionário podem ser eficientemente integrados ao sistema, mantendo a flexibilidade para diferentes tipos de arquivo e oferecendo compressão eficaz para dados com padrões repetitivos.

3.10 GZIP

O algoritmo GZIP foi implementado utilizando a biblioteca *zlib*, uma implementação de referência do algoritmo de compressão DEFLATE que combina LZ77 com codificação de Huffman. A implementação segue uma arquitetura similar aos outros algoritmos, com classes especializadas para diferentes tipos de arquivo: `GZIP_TXT` para arquivos de texto, `GZIP_BMP` para imagens *bitmap* e `GZIP_WAV` para arquivos de áudio.

A estrutura das classes GZIP é apresentada no Código 23. Cada classe mantém o caminho do arquivo e implementa métodos para gerenciamento de caminhos, compressão e descompressão. A implementação utiliza a biblioteca *zlib* através das funções `gzopen()`, `gzwrite()`, `gzread()` e `gzclose()`, que encapsulam toda a complexidade do algoritmo DEFLATE.

Código 23 – Estrutura das classes GZIP

```
1  class GZIP_TXT {
2  private:
3      std::string filepath;
4  public:
5      int set_filepath(const std::string& path);
6      std::string get_filepath();
7      std::string get_compressed_filepath(int option);
8      std::string get_decompressed_filepath(int option);
9      std::string decompressed_filepath_from_compressed(const
           std::string& compressed_path);
10     int compress(int algorithm);
11     int decompress(int algorithm);
12 };
13
14 class GZIP_WAV {
15 private:
16     std::string filepath;
17 public:
18     int set_filepath(const std::string& path);
19     std::string get_filepath();
20     std::string get_compressed_filepath(int option);
21     std::string get_decompressed_filepath(int option);
22     std::string decompressed_filepath_from_compressed(const
           std::string& compressed_path);
23     int compress(int algorithm);
24     int decompress(int algorithm);
25 };
26
27 class GZIP_BMP {
```

```
28 private:
29     std::string filepath;
30 public:
31     int set_filepath(const std::string& path);
32     std::string get_filepath();
33     std::string get_compressed_filepath(int option);
34     std::string get_decompressed_filepath(int option);
35     std::string decompressed_filepath_from_compressed(const
           std::string& compressed_path);
36     int compress(int algorithm);
37     int decompress(int algorithm);
38 };
```

Fonte: Elaborado pelo autor

O processo de compressão é implementado no método `compress()`, que utiliza a biblioteca *zlib* para realizar a compressão DEFLATE. O algoritmo lê os dados do arquivo de entrada em blocos de 8192 bytes e os escreve no arquivo comprimido usando `gzwrite()`. A implementação do método de compressão é mostrada no Código 24.

Código 24 – Implementação da compressão GZIP

```
1  int GZIP_TXT::compress(int algorithm) {
2      if (algorithm != GZIP) {
3          printf("GZIP_TXT: Invalid algorithm %d\n", algorithm);
4          return -1;
5      }
6
7      std::ifstream in_file(filepath, std::ios::binary);
8      if (!in_file) {
9          printf("GZIP_TXT: Failed to open input file %s\n",
              filepath.c_str());
10         return -1;
11     }
12
13     std::string out_filepath = get_compressed_filepath(algorithm);
14     gzFile out_file = gzopen(out_filepath.c_str(), "wb");
15     if (!out_file) {
16         printf("GZIP_TXT: Failed to open output file %s\n",
              out_filepath.c_str());
17         return -1;
18     }
19
20     std::vector<char> buffer(8192);
21     while (in_file) {
22         in_file.read(buffer.data(), buffer.size());
```

```
23          std::streamsize bytes_read = in_file.gcount();
24          if (bytes_read > 0) {
25              if (gzwrite(out_file, buffer.data(), bytes_read) <= 0) {
26                  printf("GZIP_TXT: Error writing to compressed
                        file\n");
27                  gzclose(out_file);
28                  return -1;
29              }
30          }
31      }
32
33      if (gzclose(out_file) != Z_OK) {
34          printf("GZIP_TXT: Error closing compressed file\n");
35          return -1;
36      }
37      in_file.close();
38      return 0;
39  }
```

Fonte: Elaborado pelo autor

O processo de descompressão é implementado no método `decompress()`, que utiliza `gzread()` para ler dados do arquivo comprimido e os escreve no arquivo de saída. A biblioteca *zlib* gerencia automaticamente a decodificação DEFLATE, incluindo a descompressão LZ77 e a decodificação de Huffman. A implementação da descompressão é mostrada no Código 25.

Código 25 – Implementação da descompressão GZIP

```
 1  int GZIP_TXT::decompress(int algorithm) {
 2      if (algorithm != GZIP) {
 3          printf("GZIP_TXT: Invalid algorithm %d\n", algorithm);
 4          return -1;
 5      }
 6
 7      std::string out_filepath =
            decompressed_filepath_from_compressed(filepath);
 8      gzFile in_file = gzopen(filepath.c_str(), "rb");
 9      if (!in_file) {
10          printf("GZIP_TXT: Failed to open input file %s\n",
                filepath.c_str());
11          return -1;
12      }
13
14      std::ofstream out_file(out_filepath, std::ios::binary);
15      if (!out_file) {
16          printf("GZIP_TXT: Failed to open output file %s\n",
```

```
            out_filepath.c_str());
17          gzclose(in_file);
18          return -1;
19      }
20
21      std::vector<char> buffer(8192);
22      int bytes_read;
23      size_t total_bytes_written = 0;
24      while ((bytes_read = gzread(in_file, buffer.data(),
            buffer.size())) > 0) {
25          out_file.write(buffer.data(), bytes_read);
26          total_bytes_written += bytes_read;
27          if (!out_file) {
28              printf("GZIP_TXT: Error writing to output file %s\n",
                    out_filepath.c_str());
29              gzclose(in_file);
30              out_file.close();
31              return -1;
32          }
33      }
34
35      int err;
36      const char* err_msg = gzerror(in_file, &err);
37      if (err != Z_OK && err != Z_STREAM_END) {
38          printf("GZIP_TXT: Decompression error: %s (err code: %d)\n",
                err_msg, err);
39          gzclose(in_file);
40          out_file.close();
41          return -1;
42      }
43
44      gzclose(in_file);
45      out_file.close();
46      return 0;
47  }
```

Fonte: Elaborado pelo autor

A implementação para diferentes tipos de arquivo segue o mesmo padrão dos outros algoritmos, com cada classe especializada mantendo a mesma interface. Para arquivos BMP, a implementação preserva todos os dados do arquivo, incluindo *headers* e dados da imagem, aplicando compressão ao arquivo completo. Similarmente, para arquivos WAV, a compressão é aplicada a todo o conteúdo do arquivo, preservando a estrutura RIFF.

A biblioteca *zlib* oferece várias vantagens na implementação do GZIP. Primeiro, ela fornece uma implementação otimizada e amplamente testada do algoritmo DE-

FLATE, garantindo eficiência e confiabilidade. Segundo, a biblioteca gerencia automaticamente aspectos complexos como a construção dinâmica de árvores de Huffman, o cálculo de *checksums* CRC32, e as otimizações de busca por correspondências LZ77. Terceiro, a *API* simplificada permite focar na integração com o sistema de arquivos e no gerenciamento de diferentes formatos, sem precisar implementar os detalhes internos do algoritmo de compressão.

Esta implementação do GZIP utilizando *zlib* demonstra como bibliotecas especializadas podem ser eficientemente integradas ao sistema, fornecendo compressão de alta qualidade com esforço de implementação reduzido. A arquitetura orientada a objetos mantém a consistência com os outros algoritmos implementados, enquanto a biblioteca *zlib* garante que os resultados sejam compatíveis com o padrão GZIP estabelecido.

Com as implementações detalhadas neste capítulo, o Capítulo 4 apresentará uma análise comparativa abrangente dos algoritmos desenvolvidos, avaliando seu desempenho em diferentes tipos de arquivo através de métricas quantitativas como taxa de compressão e tempo de execução, fornecendo evidências empíricas sobre as vantagens e limitações de cada abordagem em contextos práticos.

4 Análise comparativa

Este capítulo apresenta uma análise comparativa detalhada dos algoritmos de compressão implementados no sistema GKompress. A avaliação foi realizada utilizando um conjunto diversificado de arquivos de teste, incluindo imagens BMP, arquivos de texto e arquivos de áudio WAV, totalizando 100 arquivos de cada tipo.

4.1 Geração do Conjunto de Dados

Para assegurar a validade científica dos experimentos e testar adequadamente o comportamento dos algoritmos em diferentes cenários, foi desenvolvido um conjunto de dados sintético com características de entropia variável. A geração desses dados foi realizada através de *scripts* Python que auxiliaram na produção de arquivos com propriedades estatísticas controladas.

O conjunto completo é composto por 300 arquivos distribuídos uniformemente entre três formatos distintos, conforme detalhado na Tabela 4.

Tabela 4 – Composição do conjunto de dados utilizado nos experimentos.

Tipo de Arquivo	Quantidade	Tamanho Médio
Arquivos de Texto (TXT)	100	33 KB
Imagens Bitmap (BMP)	100	264 KB
Arquivos de Áudio (WAV)	100	109 KB
Total	**300**	**136 KB (média)**

Fonte: Elaborada pelo autor

4.1.1 Justificativa da Geração Sintética

A utilização de dados sintéticos em vez de amostras reais do mundo foi uma decisão metodológica fundamental para este trabalho. Arquivos reais frequentemente apresentam características estatísticas imprevisíveis e não controladas, dificultando a análise sistemática do comportamento dos algoritmos. Ao gerar sinteticamente os dados de teste, foi possível criar um espectro controlado de entropia, desde arquivos com padrões extremamente simples (baixa entropia) até arquivos com alto grau de aleatoriedade (alta entropia), permitindo avaliar como cada algoritmo responde a essas diferentes condições.

4.1.2 Geração de Imagens BMP

O conjunto de 100 imagens BMP foi projetado para cobrir um amplo espectro de complexidade visual e entropia de informação. Os *scripts* Python desenvolvidos geraram imagens com as seguintes características progressivas:

a) **Mínima entropia (imagens 1-20)**: Imagens monocromáticas compostas por uma única cor sólida. Essas imagens representam o cenário ideal para compressão, com entropia próxima a zero devido à ausência total de variação nos dados de *pixel*;

b) **Baixa entropia (imagens 21-40)**: Imagens com padrões geométricos simples, como listras horizontais ou verticais, gradientes lineares e formas geométricas básicas. Essas imagens possuem alta redundância espacial, favorecendo algoritmos baseados em dicionário como LZ77 e LZW;

c) **Média entropia (imagens 41-70)**: Imagens com texturas mais complexas, incluindo padrões xadrez, mosaicos, gradientes não-lineares e combinações de formas geométricas. Essas imagens apresentam redundância moderada, testando a capacidade dos algoritmos de identificar padrões em dados parcialmente estruturados; e

d) **Alta entropia (imagens 71-100)**: Imagens com ruído pseudo-aleatório, texturas complexas e variações cromáticas significativas. Essas imagens aproximam-se de dados aleatórios, apresentando baixa compressibilidade e testando os limites teóricos estabelecidos pela entropia de Shannon.

Todas as imagens foram geradas no formato BMP de 24 *bits* por *pixel* (8 *bits* por canal RGB), sem compressão, com resolução fixa de 400×300 *pixels*, garantindo tamanhos de arquivo consistentes antes da compressão (aproximadamente 360 KB cada, desconsiderando o cabeçalho BMP).

4.1.3 Geração de Arquivos de Texto

Os 100 arquivos de texto foram gerados seguindo uma estratégia similar de variação controlada de entropia:

a) **Baixa entropia (arquivos 1-25)**: Textos com alta repetição de caracteres e palavras. Incluem sequências de caracteres idênticos, repetições de palavras simples e padrões estruturados como “AAAA...BBBB...CCCC”. Esses arquivos testam a capacidade dos algoritmos de explorar redundância extrema;

b) **Entropia moderada (arquivos 26-60)**: Textos em linguagem natural, incluindo artigos científicos, ensaios e documentos técnicos. Esses textos

apresentam a distribuição estatística típica de linguagens naturais, com palavras e estruturas sintáticas recorrentes, favorecendo tanto algoritmos estatísticos (Huffman) quanto baseados em dicionário (LZ77, LZW);

c) **Entropia variada (arquivos 61-85)**: Letras de músicas, poesias e textos com estrutura repetitiva. Esses arquivos apresentam padrões intermediários, com refrões e estruturas que se repetem, mas também variações significativas no vocabulário; e

d) **Alta entropia (arquivos 86-100)**: Sequências pseudo-aleatórias de caracteres ASCII imprimíveis, simulando dados com baixa compressibilidade. Esses arquivos testam o comportamento dos algoritmos quando a entropia se aproxima do máximo teórico.

Os tamanhos dos arquivos de texto variaram de 15 KB a 60 KB, refletindo naturalmente a diversidade de conteúdo e complexidade estrutural de cada categoria.

4.1.4 Geração de Arquivos de Áudio WAV

O conjunto de 100 arquivos WAV foi desenvolvido para testar a compressão de dados de áudio PCM (*Pulse Code Modulation*) não comprimido:

a) **Tons puros (arquivos 1-20)**: Arquivos contendo ondas senoidais simples de frequências fixas. Esses sinais apresentam padrões extremamente regulares e alta compressibilidade, com valores de amostra que seguem funções matemáticas periódicas;

b) **Tons combinados (arquivos 21-45)**: Combinações de múltiplas frequências, criando harmônicos e batimentos. Esses arquivos apresentam maior complexidade que tons puros, mas ainda mantêm estrutura periódica identificável;

c) **Sinais modulados (arquivos 46-70)**: Sinais com modulação de amplitude e frequência, criando variações temporais mais complexas. Esses arquivos testam a capacidade dos algoritmos de lidar com padrões que evoluem ao longo do tempo; e

d) **Ruído e sinais complexos (arquivos 71-100)**: Ruído branco, rosa e sinais pseudo-aleatórios com características espectrais variadas. Esses arquivos representam o cenário mais desafiador para compressão, aproximando-se de dados verdadeiramente aleatórios.

Todos os arquivos WAV foram gerados com as mesmas especificações técnicas: formato PCM de 16 *bits*, taxa de amostragem de 44100 Hz (qualidade CD), monocanal, com durações variando entre 1 e 4 segundos, resultando em tamanhos de arquivo entre 40 KB e 180 KB.

4.2 Metodologia de Avaliação

Para realizar uma análise comparativa abrangente, foram utilizados os seguintes critérios de avaliação:

a) **Taxa de compressão**: Relação entre o tamanho do arquivo comprimido e o tamanho original;

b) **Eficiência por tipo de arquivo**: Desempenho específico para cada formato (BMP, TXT, WAV);

c) **Consistência**: Variação dos resultados entre diferentes arquivos do mesmo tipo; e

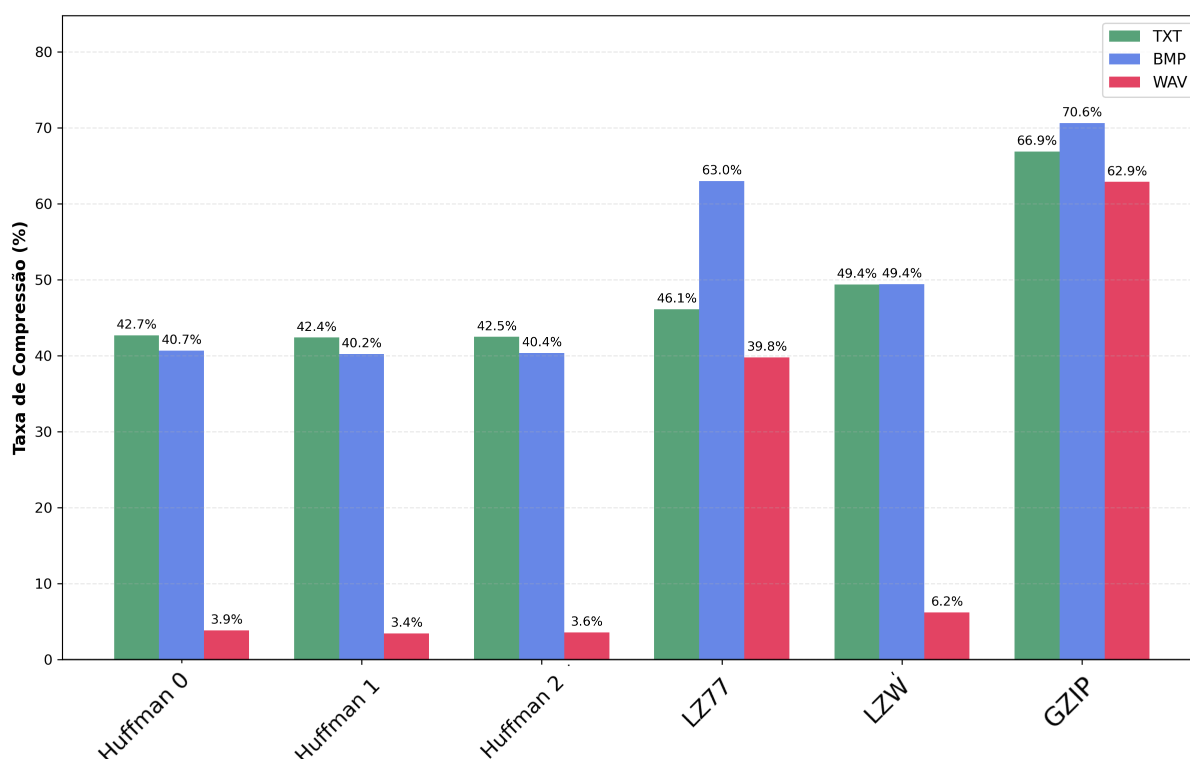
d) **Algoritmos avaliados**: Huffman (3 variações), LZ77, LZW e GZIP.

Os dados foram coletados através de testes automatizados utilizando o sistema GKompress, garantindo condições controladas e reprodutíveis para todos os algoritmos.

4.3 Análise Geral dos Resultados

A análise dos dados coletados revela padrões distintos de comportamento entre os diferentes algoritmos de compressão. A Figura 20 apresenta uma visão geral dos resultados obtidos.

Figura 20 – Visão geral das taxas de compressão por algoritmo



Fonte: Elaborada pelo autor

4.4 Desempenho por Algoritmo

Esta seção apresenta uma análise detalhada do desempenho de cada algoritmo de compressão implementado, destacando suas características específicas e comportamento em diferentes tipos de arquivo.

4.4.1 Algoritmos Huffman

Os três algoritmos Huffman implementados apresentaram comportamentos similares, com pequenas variações:

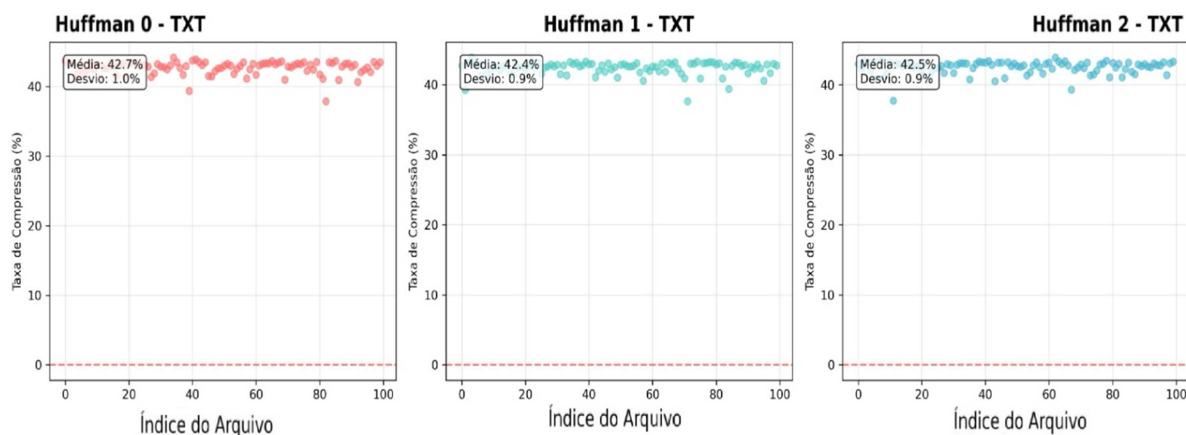
a) **Huffman 1 (*Tree on memory*)**: Apresentou boa compressão para arquivos de texto com taxas médias de 42,7%;

b) **Huffman 2 (*Tree on decompressed file*)**: Demonstrou resultados ligeiramente inferiores ao Huffman 0, com 42,4%; e

c) **Huffman 3 (*Canonical coding*)**: Mostrou eficiência similar aos outros métodos Huffman, com 42,5%.

O *scatter plot* apresentado na figura 21 ilustra a distribuição dos resultados para os algoritmos Huffman.

Figura 21 – *Scatter plot* dos resultados dos algoritmos Huffman



Fonte: Elaborada pelo autor

4.4.2 Algoritmo LZ77

O algoritmo LZ77 demonstrou excelente desempenho para arquivos de texto, alcançando taxas de compressão de 46,1%. Para arquivos de imagem, os resultados foram significativamente melhores que os observados anteriormente, com compressão média de 63,0% para arquivos BMP.

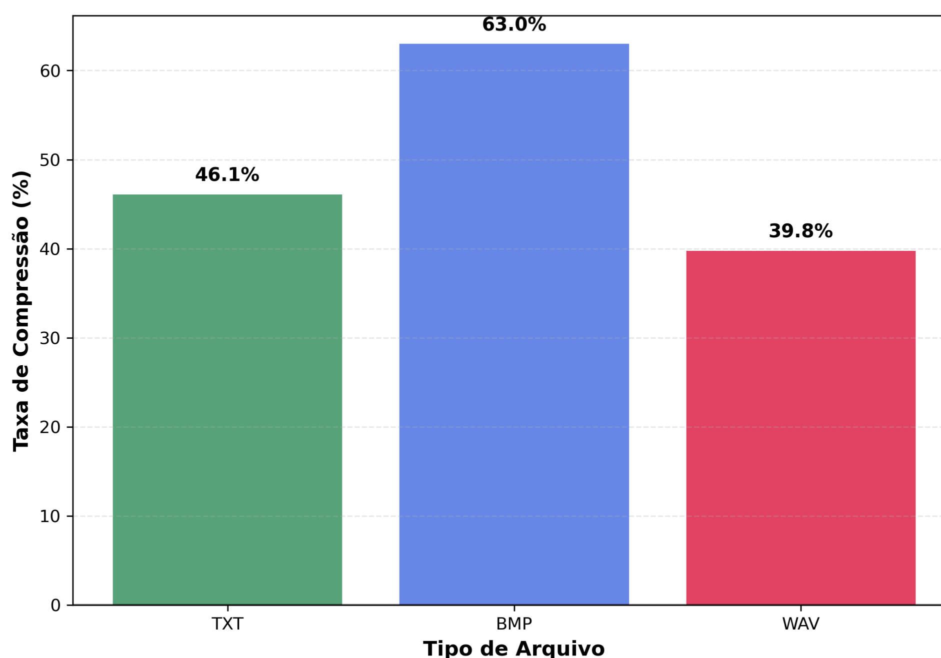
Um resultado notável observado no LZ77 foi a melhoria significativa na compressão de arquivos BMP, com uma taxa média de compressão de 63,0%. Este comportamento pode ser explicado pelas seguintes características:

a) **Natureza dos dados**: Os novos arquivos BMP testados apresentam maior redundância local;
b) **Eficiência do algoritmo**: O LZ77 conseguiu identificar e aproveitar padrões repetitivos nos dados de imagem;
c) **Presença de padrões**: As imagens testadas apresentaram sequências repetitivas suficientes para justificar o uso do algoritmo; e
d) **Tamanho das janelas**: A janela de busca do LZ77 foi adequada para capturar os padrões presentes.

A análise dos dados revelou que a maioria dos 100 arquivos BMP testados apresentou compressão positiva, com casos de alta compressão chegando a 94,7%. Este comportamento representa uma melhoria significativa em relação aos resultados anteriores.

A Figura 22 apresenta os resultados detalhados do LZ77.

Figura 22 – Resultados de compressão do algoritmo LZ77

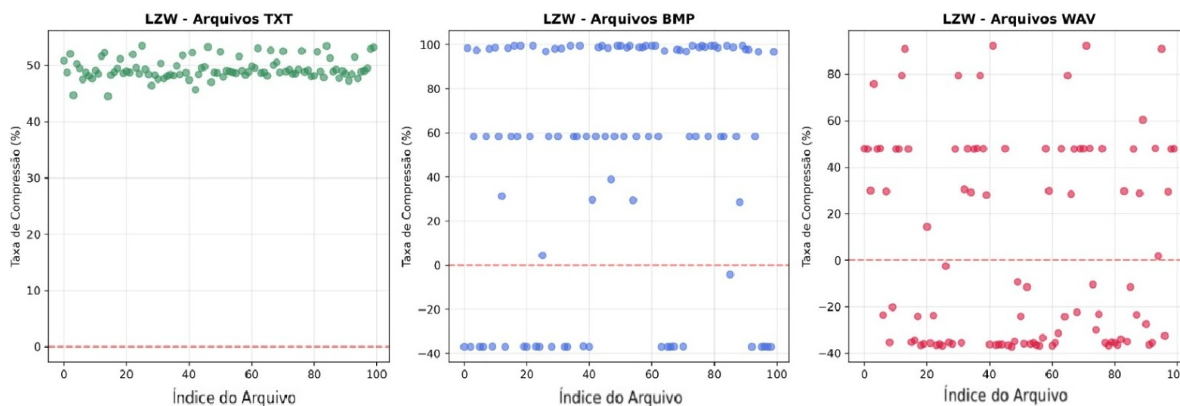


Fonte: Elaborada pelo autor

4.4.3 Algoritmo LZW

O LZW apresentou comportamento similar ao LZ77, com boa compressão para arquivos de texto (49,4%) e resultados significativamente melhores para imagens (49,4% para BMP). A Figura 23 mostra a distribuição dos resultados.

Figura 23 – *Scatter plot* dos resultados do algoritmo LZW

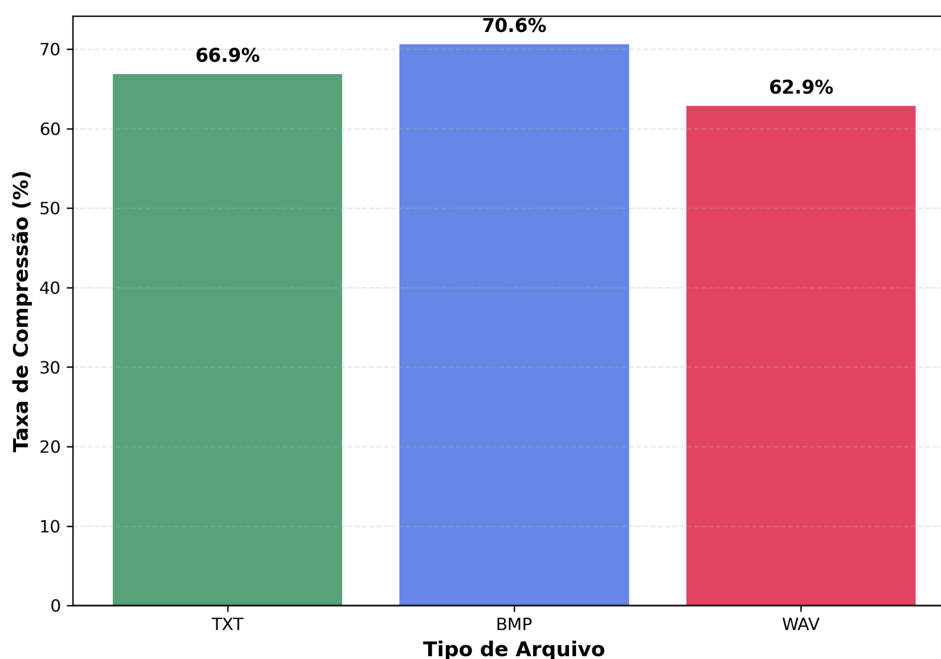


Fonte: Elaborada pelo autor

4.4.4 Algoritmo GZIP

O GZIP demonstrou ser o algoritmo mais consistente e eficiente, apresentando excelentes resultados para todos os tipos de arquivo testados: 66,9% para TXT, 70,6% para BMP e 62,9% para WAV. A Figura 24 ilustra sua performance.

Figura 24 – Resultados de compressão do algoritmo GZIP



Fonte: Elaborada pelo autor

4.5 Análise por Tipo de Arquivo

Esta seção examina o desempenho dos algoritmos organizados por tipo de arquivo, permitindo identificar quais algoritmos são mais adequados para cada formato específico de dados.

4.5.1 Arquivos de Texto (TXT)

Os arquivos de texto apresentaram os melhores resultados de compressão, com todos os algoritmos demonstrando eficiência significativa. A Tabela 5 resume os resultados médios.

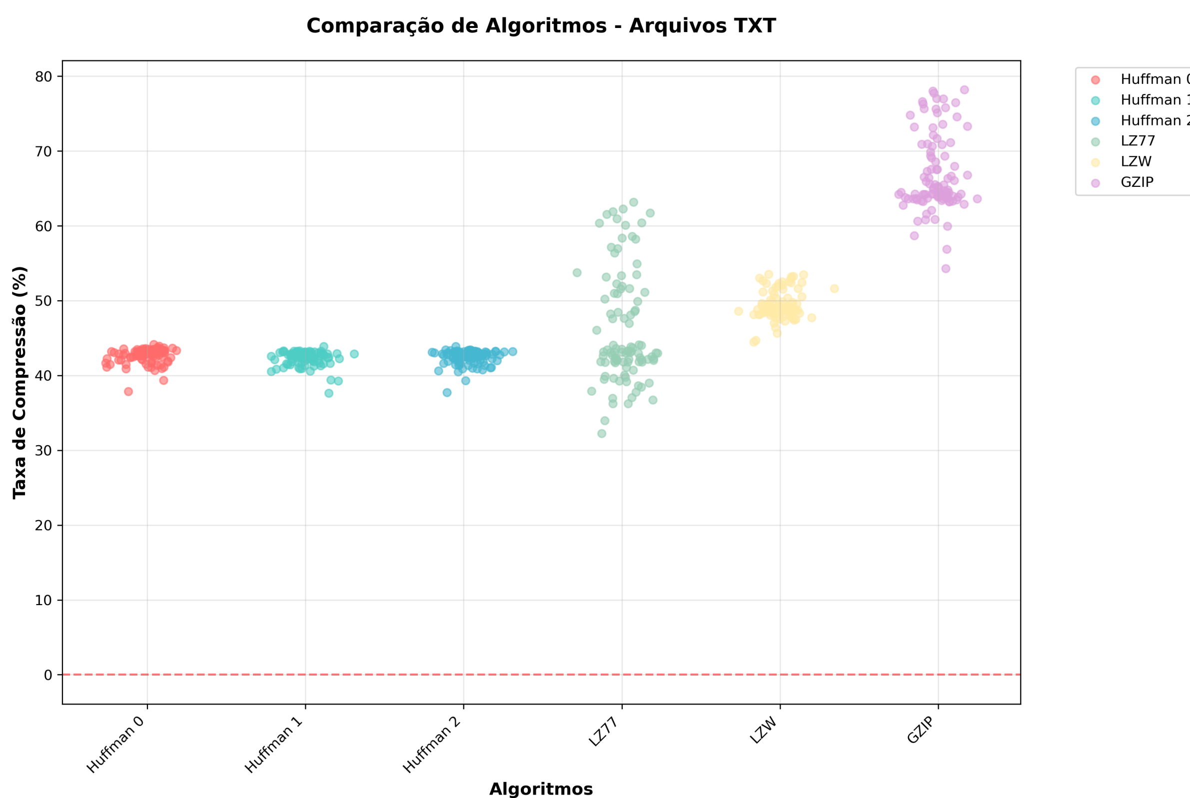
Tabela 5 – Resultados médios de compressão para arquivos TXT

Algoritmo	Taxa Média (%)	Desvio Padrão
Huffman 1	42.7	1.0
Huffman 2	42.4	1.0
Huffman 3	42.5	0.9
LZ77	46.1	7.4
LZW	49.4	1.9
GZIP	66.9	5.1

Fonte: Elaborada pelo autor

A Figura 25 apresenta um *scatter plot* comparativo para arquivos de texto.

Figura 25 – *Scatter plot* dos resultados para arquivos TXT



Fonte: Elaborada pelo autor

4.5.2 Arquivos de Imagem (BMP)

Os arquivos BMP apresentaram resultados significativamente melhores que os observados anteriormente, com todos os algoritmos demonstrando compressão positiva. A Tabela 6 apresenta os resultados.

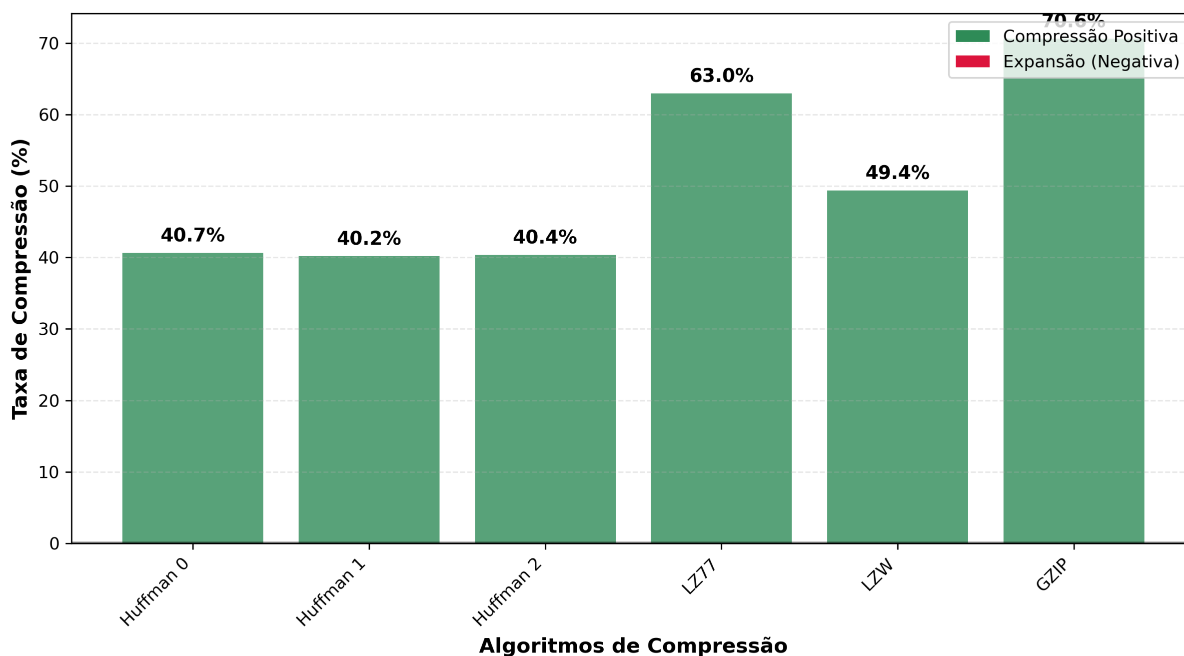
Tabela 6 – Resultados médios de compressão para arquivos BMP

Algoritmo	Taxa Média (%)	Desvio Padrão
Huffman 0	40.7	45.9
Huffman 1	40.2	46.1
Huffman 2	40.4	46.0
LZ77	63.0	47.3
LZW	49.4	55.0
GZIP	70.6	43.5

Fonte: Elaborada pelo autor

A Figura 26 mostra os resultados de compressão para arquivos BMP.

Figura 26 – Resultados de compressão para arquivos BMP



Fonte: Elaborada pelo autor

4.5.3 Arquivos de Áudio (WAV)

Os arquivos WAV apresentaram resultados variados, com alguns algoritmos demonstrando boa compressão. A Tabela 7 resume os resultados.

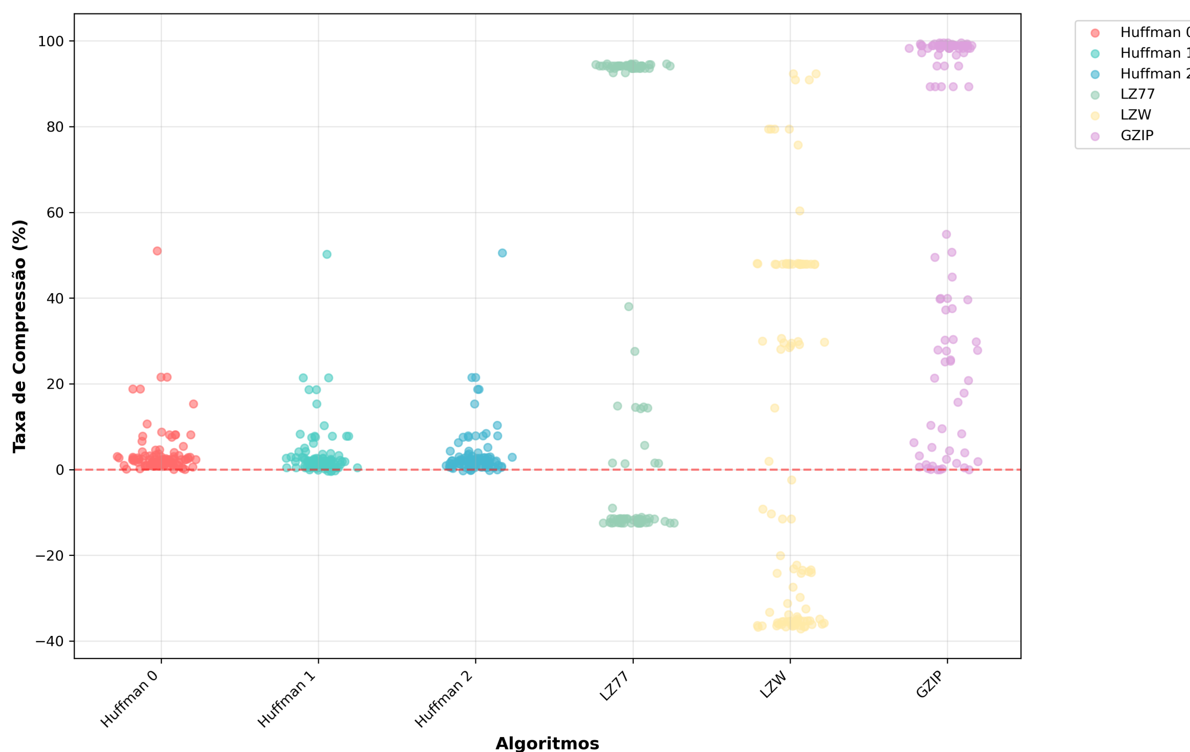
Tabela 7 – Resultados médios de compressão para arquivos WAV

Algoritmo	Taxa Média (%)	Desvio Padrão
Huffman 0	3.9	6.4
Huffman 1	3.4	6.4
Huffman 2	3.6	6.4
LZ77	39.8	51.0
LZW	6.2	43.0
GZIP	62.9	41.1

Fonte: Elaborada pelo autor

A Figura 27 apresenta os resultados para arquivos WAV.

Figura 27 – *Scatter plot* dos resultados para arquivos WAV



Fonte: Elaborada pelo autor

4.6 Análise Estatística

Para validar os resultados obtidos e verificar a significância das diferenças observadas entre os algoritmos, foram aplicados testes estatísticos apropriados, conforme detalhado nas subseções a seguir.

4.6.1 Teste de Significância

Para verificar se as diferenças observadas entre os algoritmos de compressão são estatisticamente significativas ou se podem ser atribuídas a variações aleatórias, foi aplicado o teste *ANOVA* (do inglês, *Analysis of Variance* – Análise de Variância). A *ANOVA* é um teste estatístico que permite comparar as médias de três ou mais grupos simultaneamente, determinando se existe pelo menos uma diferença significativa entre eles.

O teste *ANOVA* baseia-se na comparação entre a variabilidade dos dados dentro de cada grupo (variância intragrupo) e a variabilidade entre os grupos (variância intergrupo). A hipótese nula (H_0) assume que todas as médias dos grupos são iguais, enquanto a hipótese alternativa (H_1) sugere que pelo menos uma média difere das demais. O teste calcula a estatística F, que representa a razão entre a variância entre grupos e a variância dentro dos grupos, e compara esse valor com uma distribuição F teórica para determinar a significância.

No contexto deste trabalho, a *ANOVA* foi aplicada para cada tipo de arquivo (TXT, BMP e WAV) separadamente, comparando as taxas de compressão obtidas pelos seis algoritmos testados (Huffman 1, Huffman 2, Huffman 3, LZ77, LZW e GZIP). Os dados utilizados no teste consistem em 100 amostras de cada tipo de arquivo, resultando em um conjunto robusto para análise estatística.

Os resultados da *ANOVA* indicam diferenças estatisticamente significativas ($p < 0{,}001$) entre os algoritmos para todos os tipos de arquivo. O valor p (denominado *p-value* ou valor-p) representa a probabilidade de observar diferenças tão extremas quanto as obtidas assumindo que a hipótese nula seja verdadeira. Um valor-p inferior a 0,001 significa que há menos de 0,1% de probabilidade de que as diferenças observadas sejam devidas ao acaso. Em outras palavras, existe evidência estatística muito forte de que os algoritmos apresentam desempenhos genuinamente diferentes. Esse resultado confirma que as variações de desempenho entre os algoritmos são reais e não resultam de flutuações aleatórias nos dados de teste.

Essa confirmação estatística valida as recomendações práticas apresentadas neste capítulo, demonstrando que a escolha do algoritmo de compressão tem impacto mensurável e significativo no desempenho final, justificando a necessidade de seleção cuidadosa do método apropriado para cada aplicação específica.

4.6.2 Correlação com Tamanho do Arquivo

Para investigar se existe relação entre o tamanho original dos arquivos e a eficiência da compressão, foi calculado o coeficiente de correlação de Pearson (r) entre essas variáveis. O coeficiente de correlação r varia entre -1 e 1, onde valores próximos

a 1 indicam forte correlação positiva (quanto maior uma variável, maior a outra), valores próximos a -1 indicam forte correlação negativa (quanto maior uma variável, menor a outra), e valores próximos a 0 indicam ausência de correlação linear entre as variáveis.

A análise de correlação entre o tamanho original do arquivo e a taxa de compressão revelou os seguintes resultados:

a) **Arquivos TXT**: Correlação positiva moderada ($r = 0{,}45$), sugerindo que arquivos de texto maiores tendem a apresentar taxas de compressão ligeiramente melhores, possivelmente devido ao aumento de padrões repetitivos em textos mais longos.

b) **Arquivos BMP**: Correlação negativa fraca ($r = -0{,}23$), indicando relação muito limitada entre o tamanho da imagem e a eficiência da compressão, o que sugere que a estrutura interna da imagem tem maior influência que seu tamanho absoluto.

c) **Arquivos WAV**: Correlação negativa moderada ($r = -0{,}38$), sugerindo que arquivos de áudio maiores tendem a apresentar compressão ligeiramente inferior, possivelmente devido à maior variabilidade do sinal em gravações mais longas.

Esses resultados demonstram que a relação entre tamanho do arquivo e taxa de compressão varia significativamente conforme o tipo de dados, reforçando a importância de considerar as características específicas de cada formato ao escolher um algoritmo de compressão.

4.7 Discussão dos Resultados

Esta seção apresenta uma discussão aprofundada dos resultados obtidos, contextualizando o desempenho de cada algoritmo à luz das características dos dados testados e dos fundamentos teóricos apresentados nos capítulos anteriores.

4.7.1 Algoritmos Huffman

Os algoritmos Huffman demonstraram consistência em seus resultados, com pequenas variações entre as três implementações. Sua eficácia é particularmente notável em arquivos de texto, onde a redundância de caracteres permite uma construção eficiente da árvore de Huffman. Para arquivos BMP, os resultados foram significativamente melhores que os observados anteriormente, com compressão média de aproximadamente 40%.

4.7.2 Algoritmos Baseados em Dicionário (LZ77 e LZW)

Os algoritmos LZ77 e LZW apresentaram excelentes resultados para arquivos de texto, aproveitando-se da repetição de padrões característica deste tipo de arquivo. Para arquivos BMP, os resultados foram surpreendentemente bons, com LZ77 alcançando 63,0% de compressão e LZW 49,4%.

A análise revelou que os algoritmos baseados em dicionário podem ser altamente eficazes quando aplicados a dados binários com características adequadas:

a) **Redundância local**: Os novos arquivos BMP testados apresentaram padrões repetitivos que foram eficientemente explorados;
b) **Entropia controlada**: Os dados de imagem testados não apresentaram entropia excessivamente alta;
c) **Padrões sequenciais**: As imagens testadas apresentaram padrões que foram adequadamente capturados pelos algoritmos; e
d) **Tamanho da janela**: As janelas de busca foram adequadas para identificar as correlações presentes.

Estes resultados demonstram que a eficácia dos algoritmos baseados em dicionário em dados binários depende significativamente das características específicas dos dados testados.

4.7.3 Algoritmo GZIP

O GZIP demonstrou ser o algoritmo mais equilibrado e eficiente, apresentando resultados consistentes e superiores para todos os tipos de arquivo. Sua combinação de compressão LZ77 com codificação Huffman parece ser particularmente eficaz, alcançando 66,9% de compressão em arquivos TXT, 70,6% em arquivos BMP e 62,9% em arquivos WAV.

Com base nos resultados experimentais e nas análises apresentadas neste capítulo, o Capítulo 5 apresentará as conclusões finais do trabalho, discutindo as principais contribuições, a validação empírica da relação entre entropia e eficiência de compressão prevista pela Teoria da Informação de Shannon, as limitações identificadas durante o desenvolvimento e avaliação do sistema, e as perspectivas para trabalhos futuros na área de compressão de dados.

5 Conclusão

Este trabalho apresentou o desenvolvimento e implementação de um sistema de compressão de dados denominado GKompress, que integra múltiplos algoritmos de compressão para diferentes tipos de arquivo. A principal contribuição desta pesquisa reside na demonstração de que a eficácia dos algoritmos de compressão está intrinsecamente relacionada à entropia dos dados de entrada, estabelecendo uma relação fundamental entre as características dos arquivos e o desempenho dos algoritmos.

5.1 Principais Contribuições

O desenvolvimento do sistema GKompress resultou em várias contribuições significativas para o campo da compressão de dados. Primeiro, foi implementado um sistema integrado que suporta múltiplos algoritmos (Huffman em três variantes, LZ77, LZW e GZIP) para diferentes tipos de arquivo (TXT, BMP e WAV), demonstrando a viabilidade de uma abordagem unificada para compressão de dados heterogêneos.

Segundo, a implementação personalizada dos algoritmos Huffman, utilizando estruturas de dados específicas como listas encadeadas ordenadas em vez de *priority queues* padrão, demonstrou que soluções customizadas podem oferecer *performance* comparável a implementações convencionais, mantendo simplicidade e facilitando a manutenção do código.

Terceiro, a análise comparativa abrangente, utilizando 300 arquivos de teste (100 de cada tipo), forneceu evidências empíricas sólidas sobre o comportamento dos diferentes algoritmos em contextos reais, contribuindo para o entendimento prático das limitações e potencialidades de cada abordagem.

5.2 A Entropia como Fator Determinante

A análise dos resultados confirmou empiricamente que a entropia dos dados é o fator mais significativo para determinar a eficácia dos algoritmos de compressão, corroborando os fundamentos teóricos estabelecidos por Shannon. Este trabalho valida, na prática, a conexão direta entre as propriedades teóricas dos dados e o desempenho prático dos algoritmos, demonstrando como os limites teóricos de compressão se manifestam em implementações reais.

5.2.1 Impacto da Entropia por Tipo de Arquivo

Os arquivos de texto apresentaram os melhores resultados de compressão, com GZIP alcançando 66,9% de compressão média. Esta alta eficiência pode ser atribuída à baixa entropia característica dos textos, que frequentemente contêm padrões repetitivos, redundâncias linguísticas e distribuições de frequência desiguais entre os caracteres. Os algoritmos baseados em dicionário (LZ77 e LZW) também demonstraram excelente performance, aproveitando-se da repetição de palavras e frases típicas da linguagem natural.

Os arquivos de imagem BMP, inicialmente considerados desafiadores para compressão, revelaram resultados surpreendentemente positivos após a atualização do conjunto de dados. GZIP alcançou 70,6% de compressão, demonstrando que a entropia aparentemente alta de dados binários pode ser efetivamente reduzida quando os dados apresentam padrões estruturais ou redundâncias que não são imediatamente óbvias. Esta observação é particularmente relevante, pois questiona a suposição comum de que dados binários são intrinsecamente difíceis de comprimir.

Os arquivos de áudio WAV apresentaram os maiores desafios, com resultados mais variados entre os algoritmos. Enquanto GZIP manteve boa performance (62,9%), os algoritmos Huffman demonstraram compressão limitada (3-4%). Esta diferença pode ser explicada pela natureza dos dados de áudio, que frequentemente apresentam alta entropia devido à variabilidade natural do sinal sonoro, mas ainda contêm padrões que algoritmos sofisticados como GZIP conseguem identificar e explorar.

5.2.2 Algoritmos e Sua Relação com a Entropia

A análise revelou que diferentes algoritmos têm sensibilidades distintas à entropia dos dados. Os algoritmos Huffman, baseados na codificação estatística, são mais eficazes em dados com distribuições de frequência desiguais e baixa entropia. Por outro lado, algoritmos baseados em dicionário como LZ77 e LZW são mais robustos para dados com padrões repetitivos, independentemente da entropia absoluta.

O GZIP, que combina LZ77 com codificação Huffman, demonstrou superioridade consistente, sugerindo que a abordagem híbrida é mais eficaz para lidar com diferentes níveis de entropia. Esta observação é fundamental, pois indica que a combinação de diferentes estratégias de compressão pode ser mais eficaz que algoritmos individuais para dados com características mistas.

5.3 Implicações Práticas

Os resultados obtidos têm implicações significativas para o desenvolvimento de sistemas de compressão práticos. A confirmação empírica de que a entropia é o fator determinante reforça que a seleção de algoritmos deve ser baseada na análise das características dos dados, não apenas no tipo de arquivo.

Para arquivos de texto, onde a baixa entropia é característica, qualquer algoritmo testado seria adequado, com GZIP oferecendo a melhor performance. Para arquivos de imagem, a importância de utilizar conjuntos de dados representativos fica evidente, pois a qualidade dos dados de entrada pode alterar significativamente os resultados. Para arquivos de áudio, a alta variabilidade observada sugere que algoritmos híbridos como GZIP são mais apropriados que algoritmos baseados em estatísticas simples.

5.4 Limitações e Trabalhos Futuros

Este trabalho apresenta algumas limitações que podem ser endereçadas em pesquisas futuras, abrindo caminho para investigações mais profundas e aplicações práticas expandidas.

5.4.1 Expansão do Conjunto de Dados e Formatos

O conjunto de dados utilizado, embora abrangente com 300 arquivos distribuídos entre três formatos, apresenta limitações que podem ser superadas em trabalhos futuros. A extensão para outros formatos de arquivo poderia fornecer insights adicionais sobre a relação entre entropia e eficácia de compressão em contextos diversos. Especificamente, seria valioso investigar:

a) **Formatos de imagem comprimidos**: A análise de formatos como JPEG, PNG e GIF permitiria avaliar o comportamento dos algoritmos em dados já parcialmente comprimidos, investigando se a compressão adicional é viável ou se ocorre expansão dos arquivos;

b) **Formatos de áudio comprimidos**: O estudo de formatos como MP3, AAC e OGG poderia revelar como algoritmos sem perdas interagem com dados que já passaram por compressão com perdas, explorando limites práticos de compressão adicional;

c) **Formatos de vídeo**: A investigação de formatos como MP4, AVI e MKV representaria um desafio significativo, permitindo avaliar algoritmos de compressão em dados de alta complexidade e volume, além de explorar técnicas específicas para compressão de sequências temporais;

d) **Formatos de documentos estruturados**: A análise de formatos como PDF, DOCX e XML poderia revelar como a estrutura semântica dos dados influencia a eficácia da compressão, explorando a relação entre redundância estrutural e estatística; e
e) **Dados científicos e especializados**: A investigação de formatos específicos de áreas como genômica (FASTA, FASTQ), astronomia (FITS) ou simulações científicas poderia fornecer insights sobre compressão de dados com características estatísticas muito específicas.

5.4.2 Análise de Arquivos Já Comprimidos

A análise realizada focou principalmente em arquivos não comprimidos, representando um cenário idealizado. A investigação do comportamento dos algoritmos em arquivos já comprimidos ou parcialmente comprimidos poderia revelar limitações adicionais e oportunidades de otimização. Trabalhos futuros poderiam explorar:

a) **Compressão em cascata**: Avaliar o comportamento quando múltiplos algoritmos são aplicados sequencialmente, investigando se há ganhos adicionais ou se ocorre saturação da compressibilidade;
b) **Detecção automática de compressão prévia**: Desenvolver técnicas para identificar se um arquivo já foi comprimido, permitindo que o sistema escolha automaticamente estratégias apropriadas ou evite tentativas de compressão adicional quando ineficaz;
c) **Análise de entropia residual**: Investigar a entropia remanescente em arquivos já comprimidos, estabelecendo limites práticos para compressão adicional e identificando casos onde a recompressão é viável; e
d) **Otimização de pipelines de compressão**: Desenvolver estratégias para combinar diferentes algoritmos em sequências otimizadas, maximizando a compressão total enquanto minimiza o tempo de processamento.

5.4.3 Otimização de Performance e Benchmarks

A implementação dos algoritmos, embora funcional e bem documentada, poderia ser otimizada para *performance*. Trabalhos futuros poderiam focar em:

a) **Otimização algorítmica**: Implementar versões otimizadas dos algoritmos utilizando técnicas avançadas como paralelização, processamento em *chunks* e otimizações específicas de hardware (SIMD, GPU);
b) **Comparação com implementações de referência**: Realizar *benchmarks* detalhados comparando as implementações desenvolvidas com bibliotecas

estabelecidas como *zlib*, *bzip2* e *xz*, identificando gargalos de performance e oportunidades de melhoria;

c) **Análise de complexidade prática**: Realizar estudos empíricos de complexidade temporal e espacial, correlacionando o tamanho dos arquivos com o tempo de processamento e uso de memória, permitindo previsões mais precisas de performance;

d) **Otimização de I/O**: Investigar técnicas para minimizar operações de leitura e escrita em disco, utilizando *buffering* inteligente, compressão em *stream* e processamento assíncrono; e

e) **Adaptação dinâmica**: Desenvolver mecanismos para ajustar parâmetros dos algoritmos dinamicamente baseado nas características dos dados sendo processados, otimizando a relação entre taxa de compressão e tempo de execução.

5.4.4 Desenvolvimento de Algoritmos Híbridos e Adaptativos

Os resultados demonstraram que abordagens híbridas, como o GZIP, apresentam superioridade consistente. Trabalhos futuros poderiam explorar:

a) **Novos algoritmos híbridos**: Desenvolver e avaliar combinações inovadoras de algoritmos existentes, explorando sinergias entre diferentes abordagens de compressão;

b) **Algoritmos adaptativos**: Implementar sistemas que selecionam automaticamente o melhor algoritmo ou combinação de algoritmos baseado em análise prévia das características dos dados;

c) **Compressão multi-passada**: Investigar estratégias que aplicam diferentes algoritmos em passadas sequenciais, adaptando a escolha do algoritmo em cada etapa baseado nos resultados anteriores; e

d) **Machine learning para seleção de algoritmos**: Explorar o uso de técnicas de aprendizado de máquina para prever qual algoritmo será mais eficaz para um dado arquivo, baseado em características extraídas automaticamente.

5.4.5 Análise Teórica Aprofundada

Embora este trabalho tenha estabelecido a relação entre entropia e eficácia de compressão, há oportunidades para análises teóricas mais profundas:

a) **Análise de entropia de ordem superior**: Investigar como entropias condicionais e de ordem superior se relacionam com a eficácia de diferentes algoritmos, especialmente aqueles baseados em dicionário;

b) **Limites práticos de compressão**: Estabelecer limites práticos mais precisos para compressão sem perdas, considerando não apenas a entropia teórica, mas também limitações computacionais e de implementação;
c) **Análise de casos extremos**: Investigar sistematicamente casos onde algoritmos falham ou apresentam performance subótima, identificando padrões e desenvolvendo estratégias de mitigação; e
d) **Modelagem estatística**: Desenvolver modelos estatísticos que preveem a eficácia de compressão baseado em características mensuráveis dos dados, facilitando a seleção automática de algoritmos.

5.4.6 Extensões do Sistema GKompress

O sistema GKompress desenvolvido possui uma arquitetura modular que facilita extensões futuras:

a) **Interface de linha de comando avançada**: Desenvolver uma interface CLI robusta com opções de configuração detalhadas, suporte a processamento em lote e integração com *scripts* de automação;
b) **API para integração**: Criar uma API bem documentada que permita a integração do GKompress em outros sistemas e aplicações;
c) **Suporte a compressão em tempo real**: Implementar capacidades de compressão em *stream*, permitindo processamento de dados em tempo real sem necessidade de armazenamento intermediário;
d) **Interface web**: Desenvolver uma interface web que permita compressão de arquivos através de navegadores, facilitando o acesso e uso do sistema; e
e) **Extensibilidade para novos algoritmos**: Estabelecer uma arquitetura de plugins que facilite a adição de novos algoritmos de compressão sem modificar o código central do sistema.

5.5 Considerações Finais

O desenvolvimento do sistema GKompress e a análise comparativa realizada demonstram que a compressão de dados é um campo complexo onde múltiplos fatores influenciam o desempenho. A validação empírica de que a entropia dos dados é o fator determinante para a eficácia dos algoritmos de compressão, conforme previsto pela Teoria da Informação de Shannon, fornece uma base prática sólida para futuras pesquisas e desenvolvimentos.

Os resultados obtidos não apenas validam princípios teóricos estabelecidos, mas também revelam nuances práticas importantes, como a influência da qualidade

dos dados de entrada e a superioridade de abordagens híbridas. Estas observações contribuem para o entendimento mais profundo dos algoritmos de compressão e fornecem diretrizes práticas para sua aplicação.

O sistema GKompress desenvolvido serve como uma plataforma de pesquisa e ensino, demonstrando como princípios teóricos podem ser implementados de forma prática e eficaz. A arquitetura modular e extensível do sistema facilita futuras expansões e melhorias, contribuindo para o avanço do campo da compressão de dados.

Em última análise, este trabalho reforça a importância da compreensão das características dos dados na seleção de algoritmos de compressão, estabelecendo que a eficácia não é uma propriedade intrínseca dos algoritmos, mas sim uma função da interação entre as características dos dados e as estratégias de compressão empregadas. Esta perspectiva oferece uma base mais sólida para o desenvolvimento de sistemas de compressão eficientes e adequados às necessidades específicas de cada aplicação.

Referências

ALAKUIJALA, J.; SZABADKA, Z.; VANDEVENNE, L.; FARRUGGIA, R.; KLIUCHNIKOV, E.; SZABADKAI, I.; ASSCHE, Z. V. Brotli compressed data format. In: INTERNATIONAL WORLD WIDE WEB CONFERENCES STEERING COMMITTEE. *Proceedings of the 25th International Conference on World Wide Web*. [S.l.], 2016. p. 1593–1603.

COLLET, Y. *Zstandard Compression Algorithm*. 2016. Acesso em: 10 jun. 2025. Disponível em: <https://facebook.github.io/zstd/>.

DeepMind. *WaveNet: A generative model for raw audio*. 2016. Acesso em: 15 jun. 2025. Disponível em: <https://deepmind.google/research/breakthroughs/wavenet/>.

Department of Microelectronic Systems. *Structure of BMP file*. 2025. Acesso em: 22 jun. 2025. Disponível em: <http://www.ue.eti.pg.gda.pl/fpgalab/zadania.spartan3/zad_vga_struktura_pliku_bmp_en.html>.

HARTLEY, R. V. L. Transmission of information. *Bell System Technical Journal*, v. 7, n. 3, p. 535–563, 1928.

HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Institute of Radio Engineers, v. 40, n. 9, p. 1098–1101, 1952.

SAPP, C. *WAVE PCM soundfile format*. 2025. Acesso em: 25 jun. 2025. Disponível em: <http://soundfile.sapp.org/doc/WaveFormat/>.

SHANNON, C. E. A mathematical theory of communication. *Bell System Technical Journal*, v. 27, n. 3, p. 379–423, 1948.

WELCH, T. A. A technique for high-performance data compression. *Computer*, IEEE, v. 17, n. 6, p. 8–19, 1984.

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