

Compressão e descompressão de dados com LZW

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

Lucas Vitor da Silva Ramos
Hélio Martins de Araújo Costa Neto

Introdução

No cenário atual, onde a transferência e o armazenamento de dados desempenham um papel crucial em aplicações tecnológicas, a compressão de dados emerge como uma ferramenta indispensável. A compressão visa reduzir o tamanho dos arquivos sem perda significativa de informações, otimizando espaço em disco e tempo de transmissão em redes. O algoritmo Lempel-Ziv-Welch (LZW) é uma abordagem amplamente utilizada para compressão de dados devido à sua eficiência e simplicidade na implementação. Este trabalho prático aborda a implementação do algoritmo LZW, demonstrando seu funcionamento tanto para compressão quanto para descompressão.

Método

A implementação do algoritmo LZW foi desenvolvida em C++, a estrutura de dados escolhida para o algoritmo foi a *Trie*, utilizada para armazenar sequências de caracteres e seus respectivos códigos. Cada nó da *Trie* contém um mapa de filhos, implementado como `unordered_map<char, TrieNode*>`, que permite armazenar transições entre caracteres, além de um índice inteiro (`int index`) que indica a posição da sequência correspondente no dicionário.

A classe *Trie* foi projetada com um construtor que inicializa a raiz da árvore e define o índice inicial do dicionário como 256, cobrindo todos os caracteres ASCII. Entre seus métodos principais, destaca-se o `insert(const string& prefix, int index)`, que insere novas sequências associadas a um índice, o `search(const string& prefix)`, responsável por buscar uma sequência e retornar seu índice, e o `getNextIndex()`, que gera o próximo índice disponível para inserção. A classe também utiliza um mapa adicional (`unordered_map<int, string>`) para permitir conversões rápidas durante a descompressão.

O processo de compressão, implementado na função `lzwCompress`, inicializa a *Trie* com todos os caracteres ASCII. O algoritmo percorre os dados de entrada, construindo um prefixo crescente. Quando o prefixo atual não está presente na *Trie*, o índice do prefixo anterior é adicionado à saída comprimida, o novo prefixo é inserido na *Trie* com um índice gerado, e o prefixo atual é redefinido para o caractere não encontrado.

A descompressão, realizada pela função `lzwDecompress`, começa com o primeiro código comprimido, que é garantido estar no dicionário inicial. Cada código subsequente é processado para reconstruir sua sequência correspondente, buscando o código no mapa de dicionário. Durante o processo, novas sequências são adicionadas ao dicionário de acordo com a lógica do LZW, e a sequência atual é atualizada para acumular o resultado final.

Além disso, foram desenvolvidas funções auxiliares para manipulação de arquivos. Entre elas, destacam-se as funções para escrever a saída comprimida em um arquivo binário (`writeCompressedToFile`), ler arquivos comprimidos para descompressão

(*readCompressedFromFile*) e salvar o conteúdo descomprimido em um arquivo (*writeDecompressedToFile*). A implementação também suporta manipulação de múltiplos arquivos e tamanhos de dicionário configuráveis por meio de opções de linha de comando.

Análise de Complexidade

Na compressão, a busca na *Trie* para cada prefixo tem uma complexidade proporcional ao comprimento do prefixo, resultando em uma complexidade total de $O(n \cdot m)$, onde n é o comprimento do texto e m o tamanho médio dos prefixos. Na descompressão, o acesso ao mapa de dicionário ocorre, em média, em tempo $O(1)$, tornando a complexidade dominada pelo número de códigos na entrada comprimida, ou seja, $O(c)$. Em termos de espaço, a *Trie* consome memória proporcional ao tamanho do dicionário, com um limite superior de $O(2^b)$, onde b é o número máximo de bits por código. Além disso, são necessários espaços adicionais de $O(n)$ e $O(c)$ para armazenar, respectivamente, a entrada e a saída.

Análise Experimental

Para avaliar o desempenho do algoritmo implementado, realizamos uma série de testes abrangentes utilizando arquivos de diferentes formatos e características. Foram selecionados dois tipos principais de arquivos: .txt e .bmp, permitindo analisar o comportamento do algoritmo em dados textuais e binários. Para cada tipo, exploramos variações como alta e baixa repetitividade, tamanhos longos e curtos, totalizando 7 arquivos de teste.

Cada arquivo foi submetido a quatro execuções, variando o limite máximo de bits por código na *Trie* (*maxBits*) entre 9, 12, 16 e 20. Em cada execução, o arquivo foi comprimido, gerando métricas como taxa de compressão, tempo de execução e tamanho final do dicionário. Em seguida, o arquivo comprimido foi descomprimido para verificar a integridade do processo e coletar as mesmas métricas para a descompressão. Os resultados foram armazenados em um arquivo JSON contendo 56 linhas e 5 colunas: *maxBits*, tempo (em segundos), taxa (de compressão), tipo (compressão ou descompressão) e tamanho do dicionário [Figura 1]. A análise detalhada dos dados foi conduzida com o auxílio de ferramentas no Jupyter Notebook, o que nos permitiu observar tendências claras e consistentes:

A screenshot of a Jupyter Notebook interface showing a table with performance data. The table has 6 columns: an index column, maxBits, tamanhoDicionario, taxa, tempo, and tipo. The data is organized into pairs of rows for each maxBits value (9, 12, 16, 20), with the first row of each pair representing compression and the second representing decompression. The table is displayed in a dark-themed environment.

	maxBits	tamanhoDicionario	taxa	tempo	tipo
0	9	185364	62.282113	430	Compressão
1	9	185364	165.126196	158	Descompressão
2	9	2871	95.582010	138	Compressão
3	9	2871	2163.472647	8	Descompressão
4	9	1464993	77.659806	7811	Compressão

Figura 1: Aparência da tabela dos dados de desempenho

Taxa de compressão: Observamos que a taxa de compressão aumenta à medida que diminuimos o valor de maxBits. Isso ocorre porque, com limites menores para os códigos, o algoritmo tende a reaproveitar mais prefixos já existentes no dicionário, resultando em representações mais compactas para arquivos com dados altamente repetitivos [Figura 2].

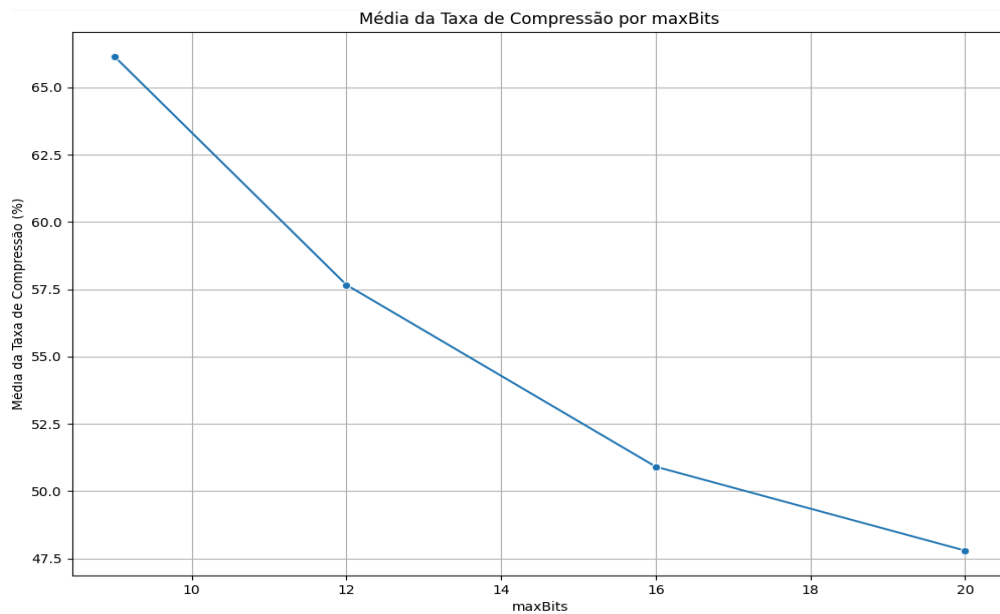


Figura 2: Gráfico de linha entre a media da taxa de compressão por maxBits

Tempo de execução: O tempo médio de execução, tanto na compressão quanto na descompressão, cresce com a redução de maxBits. Isso pode ser atribuído ao fato de que dicionários menores exigem buscas mais frequentes e recalibração das sequências, aumentando o esforço computacional [Figura 3].

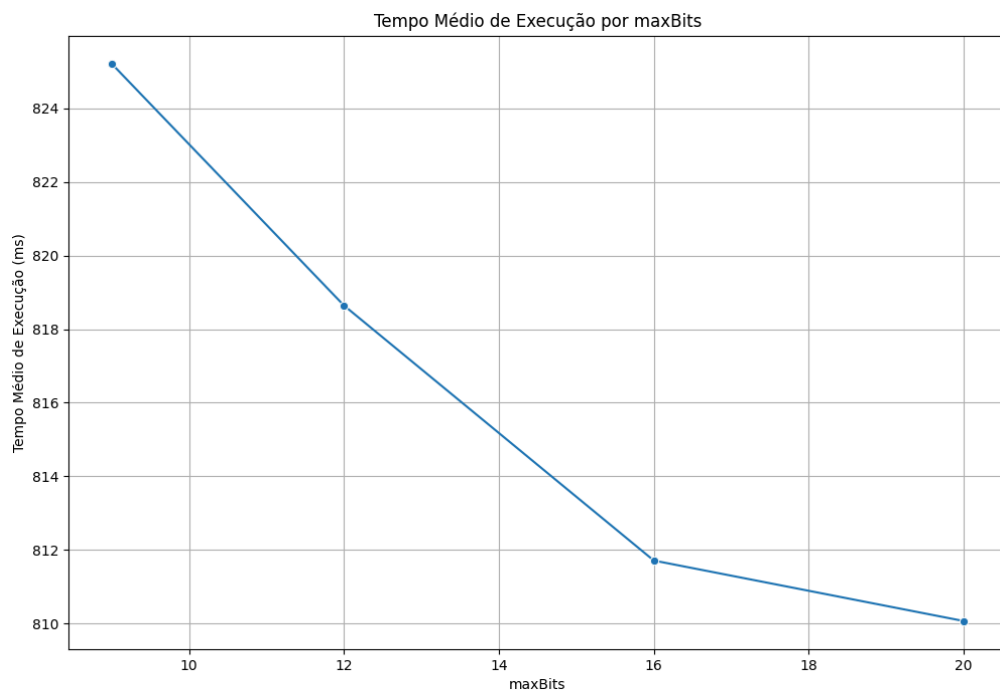


Figura 3: Gráfico linha do tempo médio de execução por maxBits

Tamanho do dicionário: Curiosamente, o tamanho do dicionário não apresentou variações significativas com mudanças em maxBits. Esse comportamento reflete o fato de que o dicionário se adapta às características dos dados e não depende diretamente do limite de bits, que apenas define o tamanho máximo de cada entrada, mas não o número total de entradas [Figura 4].

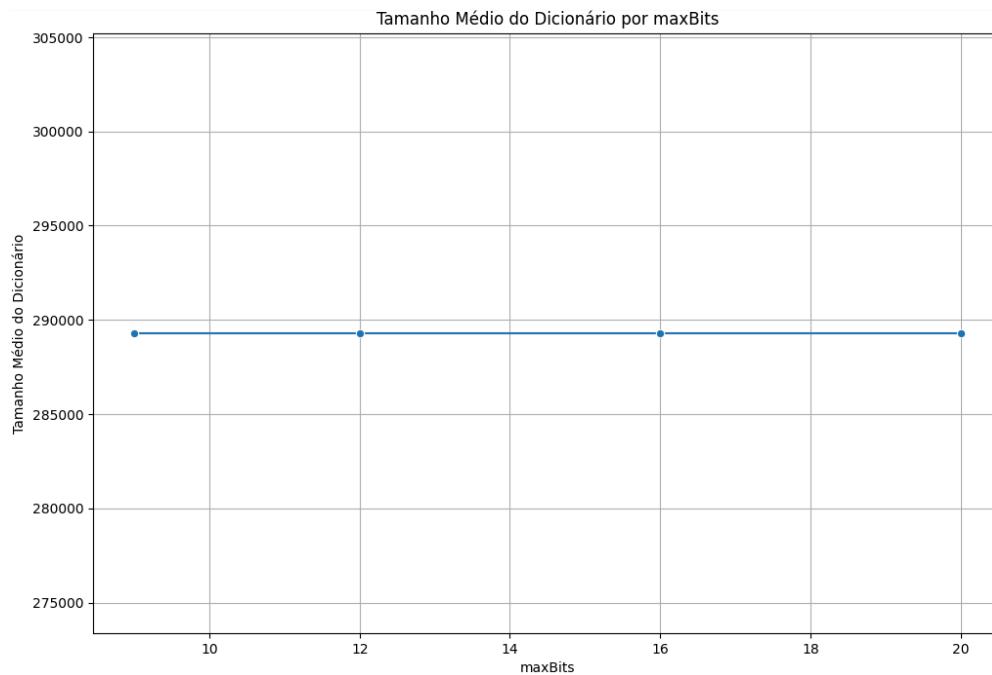


Figura 4: Gráfico de linha do Tamanho médio por maxBots

Comparação entre compressão e descompressão: A compressão se mostrou consistentemente mais demorada do que a descompressão, independentemente de maxBits ou do tamanho dos dados. Isso se explica pela natureza do processo de compressão, que requer inserções e buscas constantes na Trie, enquanto a descompressão opera de forma mais linear, apenas traduzindo códigos em sequência [Figura 5].

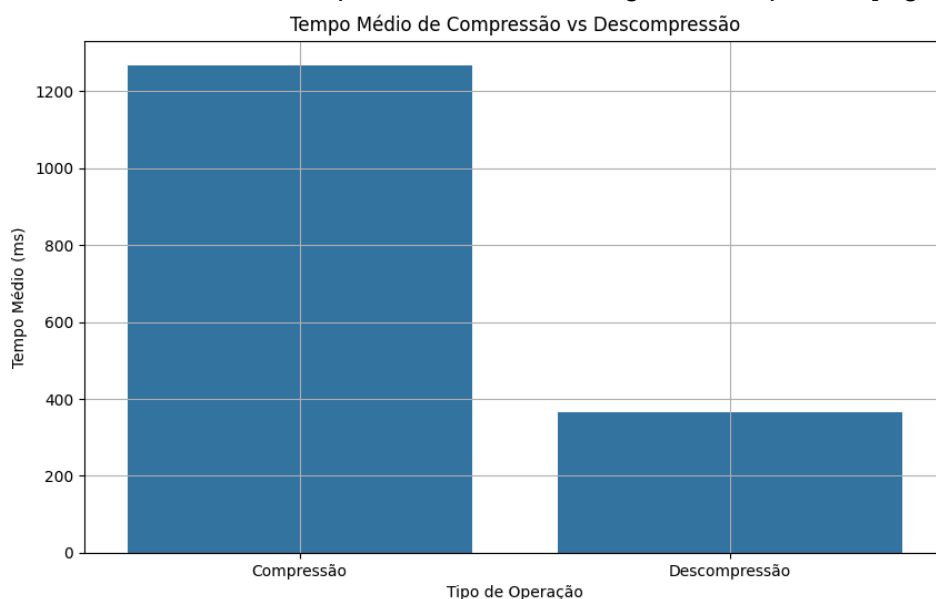


Figura 5: Gráfico de barras do tempo médio de execução de compressão de descompressão

Impacto do tamanho do dicionário: Com o aumento do tamanho do dicionário, foi observado um aumento considerável no tempo de execução, especialmente na compressão. Além disso, a diferença de tempo entre compressão e descompressão se torna mais evidente. Isso indica que o gerenciamento de um dicionário maior adiciona complexidade ao processo de compressão, enquanto a descompressão permanece relativamente eficiente [Figura 6].

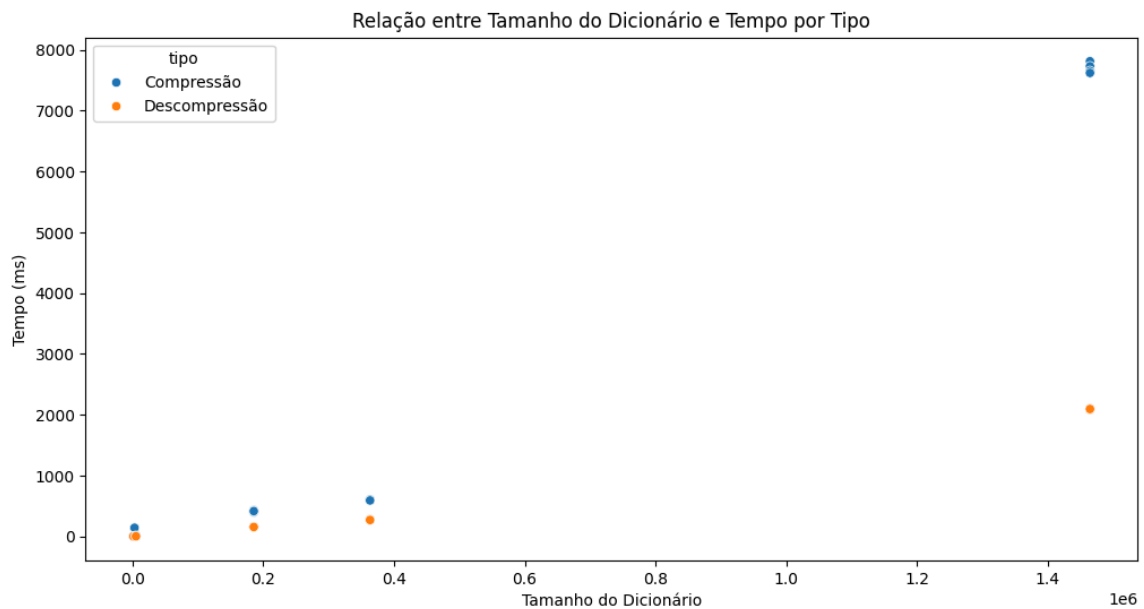


Figura 6: Gráfico de pontos

Esses resultados não apenas validam o funcionamento do algoritmo, mas também destacam as trocas entre eficiência e desempenho relacionadas ao parâmetro maxBits e às características dos dados de entrada. Essa análise pode ser usada para ajustar o algoritmo a diferentes cenários, dependendo das prioridades (compressão máxima ou menor tempo de execução).

Conclusão

Este trabalho abordou a implementação do algoritmo LZW para compressão e descompressão de dados, destacando seu funcionamento, eficiência e limitações por meio de uma análise experimental detalhada. Ao longo do desenvolvimento e da análise, aprendemos que o algoritmo apresenta um equilíbrio interessante entre compactação e desempenho. Observou-se que a redução do parâmetro maxBits melhora a taxa de compressão, mas ao custo de um aumento significativo no tempo de execução, tanto para compressão quanto para descompressão. Essa relação reflete a natureza intrínseca do LZW, onde dicionários menores exigem mais operações para gerenciar as sequências de prefixos.

Outra descoberta importante foi que o tamanho do dicionário, apesar de ser configurável, não variou significativamente com diferentes valores de maxBits, indicando que sua eficiência depende mais das características dos dados de entrada do que do tamanho máximo permitido para os códigos. Além disso, a compressão foi consistentemente mais demorada que a descompressão, evidenciando a complexidade adicional do processo de inserção na estrutura de dados.

Por fim, os testes reforçaram a importância de ajustar os parâmetros do algoritmo conforme o contexto de uso. Para cenários onde espaço em disco é crítico, vale priorizar configurações que maximizem a compressão, mesmo que o tempo de execução aumente. Já em aplicações onde a velocidade é mais relevante, parâmetros mais permissivos podem ser a melhor escolha. Este estudo, portanto, não apenas implementou e validou o algoritmo, mas também forneceu insights práticos para sua aplicação em diferentes contextos.

Referências

<https://web.mit.edu/6.02/www/s2012/handouts/3.pdf>

<https://www.davidsalomon.name/DC4advertis/DComp4Ad.html>

[https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Signal_Processing_and_Modeling/Information_and_Entropy_\(Penfield\)/03%3A_Compression/3.07%3A_Detail-_LZW_Compression](https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Signal_Processing_and_Modeling/Information_and_Entropy_(Penfield)/03%3A_Compression/3.07%3A_Detail-_LZW_Compression)

<https://iq.opengenus.org/lempel-ziv-welch-compression-and-decompression/>