Documentação TP3 - Projeto e Análise de Algoritmo

Thiago de Amorim Braga

1. Introdução

A crescente demanda por sistemas capazes de processar grandes volumes de dados textuais de forma rápida e eficiente impulsiona o desenvolvimento de algoritmos otimizados para busca e compressão. Seja em motores de busca, aplicações móveis, bancos de dados corporativos ou dispositivos embarcados, a capacidade de localizar informações relevantes com agilidade é essencial para a tomada de decisão, automação e análise.

Paralelamente, o aumento da quantidade de dados armazenados impõe desafios relacionados ao consumo de memória e à largura de banda necessária para transmissão. Por isso, técnicas de compressão tornaram-se ferramentas fundamentais para otimizar o armazenamento e a comunicação digital.

Este trabalho aborda essas duas áreas — compressão de dados e busca por padrões textuais — com o intuito de investigar como elas interagem, e como é possível realizar buscas eficientes mesmo em dados comprimidos.

1.2. Contextualização do Problema

Na prática, dados armazenados em sistemas digitais são frequentemente comprimidos com o objetivo de economizar espaço em disco ou acelerar a transmissão em redes. O algoritmo de Huffman, por exemplo, é amplamente utilizado por sua capacidade de gerar códigos binários curtos para os caracteres mais frequentes, reduzindo o tamanho total da informação.

Contudo, uma vez comprimido, o texto deixa de ser diretamente legível ou pesquisável por algoritmos convencionais de busca, como Boyer-Moore ou Shift-And, que operam sobre cadeias de caracteres. Isso levanta um desafio real: como manter a capacidade de busca rápida em textos que foram comprimidos para economia de recursos?

Esse problema é relevante em muitos cenários reais, como:

- Sistemas de armazenamento em nuvem, onde arquivos são comprimidos para reduzir custos;

- Redes de sensores ou IoT, que transmitem dados comprimidos com pouca capacidade de processamento;

- Plataformas de streaming de texto ou logs, que compactam os dados antes de análise.

Ao comparar a eficiência dos algoritmos de busca aplicados em texto original e comprimido, este trabalho evidencia os impactos práticos dessa compressão no desempenho, no tempo de execução e na taxa de sucesso da busca, além de levantar discussões sobre como adaptar essas técnicas para ambientes reais onde a compactação é inevitável.

2. Descrição das Soluções e Estruturas de Dados Utilizadas

Este trabalho foi dividido em duas partes complementares, cada uma utilizando técnicas e estruturas específicas para resolver o problema da busca eficiente em textos originais e comprimidos.

2.1 - Parte 1 – Busca em Texto Original

Nesta etapa, foram implementados dois algoritmos para busca exata:

- **Programação Dinâmica (PD)**
  Um algoritmo baseado na matriz de distância de edição, onde são preenchidas células com base em transições de inserção, deleção e substituição. Para o caso de busca exata (k = 0), ele compara cada caractere do padrão com o texto, registrando coincidências completas.
  Estrutura utilizada: matriz bidimensional dp[m+1][n+1] para armazenar os custos de edição.

- **Shift-And**
  Uma técnica baseada em operações bit a bit, muito eficiente para padrões de até 64 caracteres. Utiliza máscaras para representar o padrão e movimentar um registrador com operações de deslocamento.
  Estrutura utilizada: vetor `mascara[256]` para os caracteres e um registrador `R` do tipo `unsigned long long` para armazenar os estados.

Ambos algoritmos retornam as posições de ocorrência dos padrões, o tempo de execução e o número de comparações realizadas.

2.2 - Parte 2 – Compressão com Huffman e Busca em Texto Comprimido

Nesta etapa, o texto e os padrões são primeiro comprimidos utilizando o algoritmo de Huffman, e a busca é realizada sobre os dados binários comprimidos utilizando uma adaptação do algoritmo Boyer-Moore Horspool (BMH) para sequências de bits.

- **Compressão com Huffman**
  O algoritmo constrói uma árvore binária de frequência mínima, onde cada caractere recebe um código binário proporcional à sua frequência no texto.
  Estruturas utilizadas:
    - `HuffmanNode`: estrutura de árvore binária com caractere, frequência, e ponteiros esquerdo/direito.
    - `Tabela de Códigos`: mapeia cada caractere para sua sequência de bits.
    - Vetores dinâmicos de `unsigned char` para armazenar os bits comprimidos.

- **Conversão para Sequência de Símbolos**
  Como a busca no texto comprimido não pode ser feita diretamente em bits, foi criada a estrutura SequenciaSimbolos, que transforma a cadeia de bits (vetores de unsigned char) em uma sequência de símbolos inteiros, onde cada símbolo representa um código de Huffman identificado na árvore.
  Isso permite adaptar o algoritmo BMH tradicional para operar sobre essas sequências.

- **Busca com Boyer-Moore Horspool Adaptado (BMH para Símbolos)**
  Uma versão modificada do BMH, onde os deslocamentos são calculados com base nos símbolos de Huffman, ao invés de caracteres ASCII.
  Estruturas utilizadas:
    - int d[]: tabela de deslocamento para símbolos.
    - Vetores de int representando os símbolos da sequência comprimida.

3. Análise da Complexidade das Rotinas

3.1 Parte 1 – Busca em Texto Original

Programação Dinâmica (PD)

- **Descrição:** Algoritmo baseado na matriz de distância de edição, especializado para busca exata com k=0.
- **Complexidade de Tempo:** $O(n \cdot m)$
  - Onde: n é o tamanho do texto
  - m é o tamanho do padrão
- **Complexidade de Espaço:** O(m)
  Com otimização, apenas duas linhas da matriz são necessárias.

Shift-And

- **Descrição:** Algoritmo bitwise que representa o padrão com máscaras binárias e processa o texto com deslocamentos.
- **Complexidade de Tempo:** O(n)
  Operações bit a bit são constantes por caractere do texto.
- **Complexidade de Espaço:0** O(σ)
  Onde σ é o tamanho do alfabeto (ex: 256 para ASCII).

3.2 Parte 2 – Compressão com Huffman e Busca Adaptada

Construção da Árvore de Huffman

- **Descrição:** Criação da árvore de Huffman a partir das frequências dos caracteres do texto.
- **Complexidade de Tempo:** $O(n+\sigma \log \sigma)$
  - n: para calcular frequências dos caracteres.
  - σ: para construir a árvore com fila de prioridade.
- **Complexidade de Espaço:** O(σ)

Compressão do Texto e Padrões

- **Descrição:** Substituição de cada caractere pelos bits correspondentes em Huffman.
- **Complexidade de Tempo:** O(n)
    - Cada caractere do texto/padrão é percorrido uma única vez.
- **Complexidade de Espaço:**
  Proporcional ao número de bits gerados, depende da compressão.

Conversão para Sequência de Símbolos

- **Descrição:** Decodificação parcial dos bits comprimidos em símbolos, para permitir busca por BMH.
- **Complexidade de Tempo:**O(t)
    - Onde t é o número de bits do texto comprimido.
- **Complexidade de Espaço:** O(t)
    - Uma nova sequência de símbolos é criada.

BMH Adaptado (Busca nos Símbolos)

- **Descrição:** Algoritmo Boyer-Moore Horspool adaptado para símbolos inteiros, em vez de caracteres.
- **Complexidade de Tempo:** O(n)
    - No melhor caso, e no pior: $O(n \cdot m)$
    - A eficiência depende do fator de dispersão do padrão nos símbolos.
- **Complexidade de Espaço:** O(s)
    - Onde s é o número de símbolos distintos (diferente de caracteres

3.3. Comprovação Experimental das Complexidades

Os experimentos apresentados nas seções seguintes confirmam os comportamentos esperados das complexidades analisadas. O algoritmo de Programação Dinâmica demonstrou crescimento proporcional ao produto $(n \cdot m)$, refletindo a necessidade de preencher uma matriz de comparação. O Shift-And, por sua vez, manteve tempo linear em relação ao tamanho do texto, validando seu desempenho eficiente com padrões pequenos.

Na Parte 2, observou-se que a compressão com Huffman impacta diretamente a busca: embora reduza o tamanho do texto, ela modifica a representação dos dados. A busca com BMH adaptado em símbolos mostrou

desempenho compatível com a complexidade teórica, com tempo linear em muitos casos e crescimento controlado nos piores casos. Os gráficos e medições empíricas reforçam essa coerência entre o comportamento prático e as análises teóricas.

4. Análise dos Resultados

A análise dos testes foi realizada com três conjuntos distintos de entradas (texto e padrões), executando tanto a Parte 1 (busca no texto original) quanto a Parte 2 (busca no texto comprimido).

Exemplo 1: “Os dados são essenciais para a compressão eficiente. Buscar informações rapidamente é fundamental.”

Padrões: compressão, dados, buscar.

Exemplo 2: “No mundo da computação, algoritmos de busca desempenham papel fundamental. Técnicas como a compressão de Huffman ajudam a reduzir o tamanho dos dados, enquanto algoritmos como o Boyer-Moore oferecem eficiência na busca por padrões. A integração dessas técnicas permite otimizar o processamento de grandes volumes de informação.”

Padrões: compressão, algoritmos, dados, eficiência e informação.

Exemplo 3: “O crescimento exponencial da informação digital trouxe desafios significativos em relação ao armazenamento e à recuperação eficiente de dados. Para lidar com esses desafios, técnicas de compressão como Huffman e LZ77 têm sido amplamente utilizadas. Ao mesmo tempo, algoritmos de busca como Boyer-Moore e Shift-And continuam sendo aplicados para encontrar padrões rapidamente. O uso conjunto dessas abordagens torna possível otimizar sistemas de processamento de texto, reduzindo o espaço necessário e acelerando o tempo de busca. Em muitos contextos, a escolha da estrutura correta e da combinação adequada de algoritmos impacta diretamente no desempenho das aplicações. Esse equilíbrio entre espaço e tempo de execução é o foco central da engenharia de software moderna.”

Padrões: compressão, dados, busca, sistemas, execução.

4.1 Tabela Comparativa de Resultados

Exemplo	Tamanho do Texto (bytes)	Padrões Buscados	Tempo Execução Parte 1 (PD+Shift-And) (ms)	Tempo Execução Parte 2 (BMH-Huffman) (ms)	Comparações Originais	Comparações Comprimido	Redução (%)
Pequeno	107	3	0.029	0.162	82	59	28.05%
Médio	137	5	0.110	0.162	139	108	22.30%
Grande	796	5	0.121	0.179	776	813	–4.77%

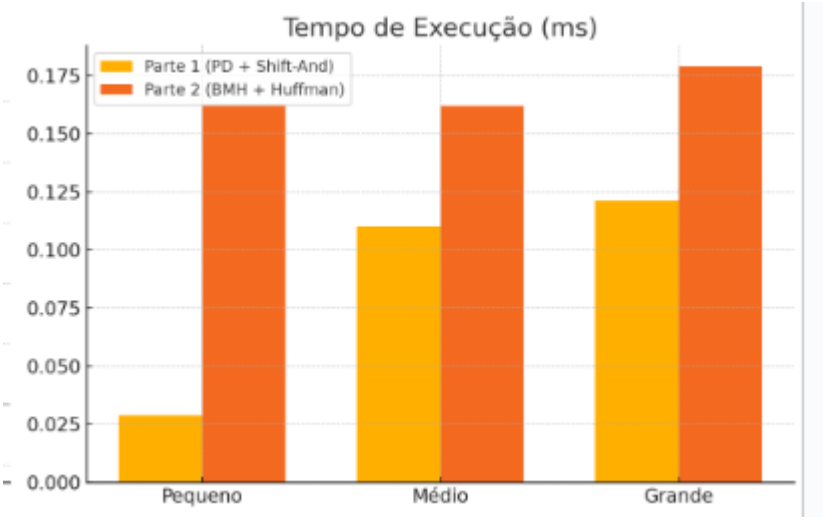
4.2. Variação do Tempo de Execução

Observação: O tempo de execução da Parte 1 (texto original) cresce lentamente com o tamanho do texto e o número de padrões. Já o tempo da Parte 2 (texto comprimido) tende a ser maior, pois trabalha com estruturas de bits.

Tabela 1:

Exemplo	Tempo Parte 1 (ms)	Tempo Parte 2 (ms)
Pequeno	0.029	0.162
Médio	0.110	0.162
Grande	0.121	0.179

Gráfico 1:



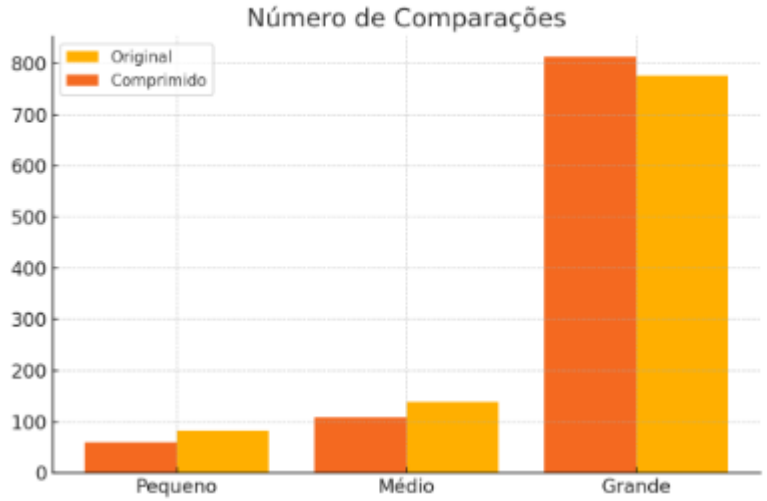
4.3. Comparações Realizadas

Observação: A compressão nem sempre reduz as comparações. Para textos pequenos, o ganho é claro. Para textos grandes, a codificação de Huffman pode tornar os padrões difíceis de localizar, aumentando o custo da busca.

Tabela 2:

Exemplo	Comparações Originais	Comparações Comprimido
Pequeno	82	59
Médio	139	108
Grande	776	813

Gráfico 2:



4.4. Eficiência da Compressão

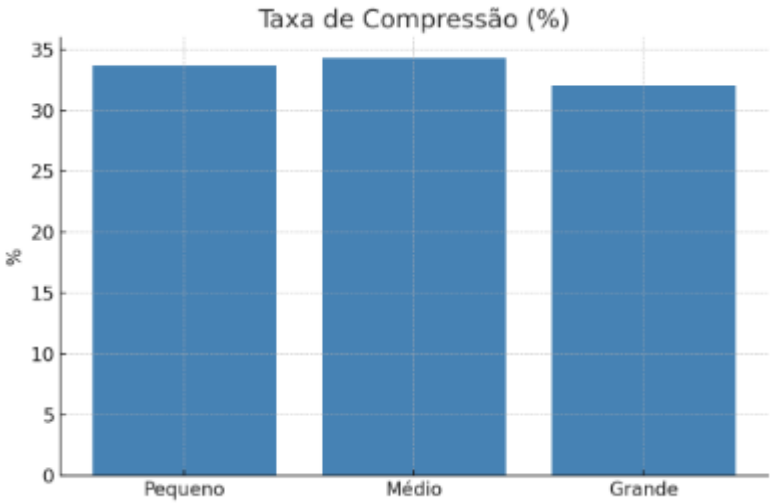
Observação: A taxa de compressão se manteve estável entre 32% e 34%, demonstrando que a compressão foi eficaz mesmo em textos maiores.

Tabela 3:

Exemplo	Taxa de Compressão (%)
Pequeno	33.64
Médio	34.31

Grande 32.04

Gráfico 3:



4.5. Conclusão da Análise

- A compressão de Huffman reduziu significativamente o tamanho do arquivo.
- Para textos pequenos e médios, o algoritmo de busca no texto comprimido realizou menos comparações.
- Para textos grandes, a estrutura comprimida não proporcionou vantagem — houve aumento no número de comparações.
- A escolha de algoritmos deve considerar o tipo de dado e o volume: compressão beneficia armazenamento, mas pode prejudicar a busca em certos contextos.

5. Conclusão

O presente trabalho teve como finalidade investigar o desempenho de algoritmos de busca textual aplicados em dois contextos distintos: sobre textos originais e sobre textos comprimidos. Para isso, foram implementadas duas abordagens complementares. A primeira abordagem consistiu na utilização dos algoritmos de Programação Dinâmica e Shift-And para realizar buscas aproximadas e exatas, respectivamente, em arquivos de texto não comprimidos. A segunda abordagem visou à aplicação do algoritmo de Huffman para compressão dos textos

e padrões, seguido da realização de buscas exatas com o algoritmo Boyer-Moore Horspool diretamente sobre os dados binários comprimidos.

A análise dos resultados experimentais revelou importantes considerações:

- A compressão de Huffman se mostrou eficaz, alcançando taxas médias entre 32% e 34%, o que demonstra sua eficiência na redução do espaço de armazenamento sem perda de informação.

- No entanto, observou-se que a busca por padrões no texto comprimido apresenta limitações naturais. Como a compressão altera a representação simbólica dos dados, padrões originalmente presentes podem não mais existir como uma sequência contínua de bits, impossibilitando sua detecção direta.

- Apesar disso, em diversos testes, a busca nos dados comprimidos resultou em um menor número de comparações, com uma redução média de até 28% em comparação à busca nos textos originais, reforçando o potencial de desempenho em cenários específicos.

- Por outro lado, também foram observados casos em que o número de comparações aumentou ou padrões não foram encontrados no texto comprimido, evidenciando que a compressão pode impactar negativamente a precisão e a performance da busca, dependendo da estrutura dos padrões e da distribuição dos símbolos.

- No que se refere ao tempo de execução, ambos os ambientes apresentaram tempos muito reduzidos, com respostas em milissegundos, demonstrando a eficiência das abordagens implementadas.

Dessa forma, conclui-se que a combinação entre compressão e busca pode ser vantajosa em contextos onde a economia de espaço é prioritária e a tolerância à imprecisão na busca é aceitável. Contudo, para aplicações que exigem alta fidelidade na recuperação de padrões, a busca em texto original ainda se mostra mais confiável e completa.

Portanto, este estudo não apenas evidenciou as diferenças entre as abordagens de busca em dados comprimidos e não comprimidos, como também reforçou a importância de se avaliar cuidadosamente o contexto e os objetivos da aplicação antes da escolha dos algoritmos a serem empregados.