Universidade Federal
de São João del-Rei

Trabalho Prático 3

Disciplina: Projeto e Análise de Algoritmos
Integrantes: Gabriel da Silva Souza
José Vitor Santos Alves
Professor: Leonardo Chaves Dutra da Rocha

São João del-Rei - Junho, 2025

Sumário

1. O Problema

O problema central do trabalho consiste na busca de padrões textuais em arquivos, considerando tanto o caso de arquivos não comprimidos quanto arquivos comprimidos por meio do algoritmo de Huffman com marcação. Além disso, o problema é abordado em dois cenários distintos: a busca aproximada, que admite certo número de erros entre o padrão e o texto, e a busca exata, que exige a correspondência perfeita do padrão.

Na Parte 1 do trabalho, o objetivo é localizar ocorrências de padrões em arquivos de texto comuns, permitindo até k erros de substituição, inserção ou remoção. Para isso, foram implementados dois algoritmos clássicos de casamento aproximado de padrões: (i) um baseado em programação dinâmica, que computa uma matriz de distâncias de edição; e (ii) o algoritmo Shift-And, adaptado para tolerar erros. O desempenho desses algoritmos foi avaliado com diferentes valores de k (de 0 a 3), considerando tanto o número de comparações realizadas quanto o tempo de execução.

Na Parte 2, o objetivo é localizar a busca exata de padrões em arquivos comprimidos com o algoritmo de Huffman com marcação, conforme descrito por Ziviani (2004). Essa técnica permite que o padrão também seja comprimido e buscado diretamente no arquivo comprimido, evitando a necessidade de descompressão prévia. Para isso, foi utilizado o algoritmo de Boyer-Moore Horspool (BMH), conhecido por sua eficiência prática em buscas exatas, tanto nos arquivos originais quanto nos comprimidos. A comparação de desempenho entre esses dois contextos permite avaliar o impacto da compressão no processo de busca.

2. Módulos do Projeto

2.1. Módulo io.h/io.c

Responsável pela Leitura e Escrita de Arquivos

- char* lerArquivo(const char* nomeArquivo)
    - Lê todo o conteúdo de um arquivo de texto e retorna como uma string;
- void escreverArquivo(const char *nomeArquivo, const char* conteudo)
    - Escreve o conteúdo no arquivo de saída.

2.2. Módulo alg_parte1.h/alg_parte1.c

Contém os algoritmos de busca aproximada

- ResultadoParte1 ProgramacaoDinamica(char *texto, int tam_texto, char *padrao, int tam_padrao, int k_erros);
    - Algoritmo de programação dinâmica para casamento aproximado.
- ResultadoParte1 ShiftAnd(char *texto, int tam_texto, char *padrao, int tam_padrao, int k_erros);
    - Algoritmo Shift-And para até k erros.

2.3. Módulo alg_parte2.h/alg_parte2.c

Contém os algoritmos de compressão e busca

- void ComprimirHuffman(const unsigned char *dados, size_t tamanho, unsigned char **dados_comprimidos, size_t *tamanho_comprimido);

    - Comprime um vetor de dados com base na frequência dos caracteres.
- long long buscar_comprimido(const unsigned char *texto, size_t tam_texto, const unsigned char *padrao, size_t tam_padrao);
    - Busca exata de bytes no texto comprimido.
- ResultadoParte2 BoyerMooreHorspool(char *texto, int tam_texto, char *padrao, int tam_padrao)
    - Busca em texto não comprimido.

3. Algoritmos em Detalhe

3.1. Parte 1 - Busca Aproximada

Pseudocódigo:

Algoritmo 1: Busca Aproximada (Programação Dinâmica ou Shift-And)

Entrada: algoritmo (1=PD, 2=SA), arquivoTexto, arquivoPadroes;
Saída: Ocorrências dos padrões nos textos (com até 3 erros);
texto = LerArquivo(arquivoTexto);
padroes = LerArquivo(arquivoPadroes);
i = 0;
while padrao em padroes **do**
    for k = 0 até 3 **do**
        if algoritmo = 1 **then**
            resultado = ProgramacaoDinamica(texto, padrao, k);
        else if algoritmo = 2 **then**
            resultado = ShiftAnd(texto, padrao, k);
        Imprimir(tempo, comparações);
        if k = 3 **then**
            EscreverArquivo(saida.txt, resultado.ocorrencias);
    i = i + 1;

3.2. Parte 2 - Busca em Texto Comprimido

Pseudocódigo:

Algoritmo 2: Busca Exata em Texto Comprimido

Entrada: arquivoTexto, arquivoPadroes;
Saída: Ocorrências dos padrões nos textos (não comprimido e comprimido);
texto = LerArquivo(arquivoTexto);
padroes = LerArquivo(arquivoPadroes);
(textoComprimido, tamComprimido) = ComprimirHuffman(texto);
i = 0;
while padrao em padroes **do**
    resultado = BoyerMooreHorspool(texto, padrao);
    (padraoComprimido, tamPadraoComprimido) = ComprimirHuffman(padrao);
    compsComprimido = BuscarComprimido(textoComprimido, padraoComprimido);

Imprimir(tempo não comprimido, comps não comprimido, tempo comprimido, compsComprimido);
EscreverArquivo(saida.txt, resultado.ocorrencias);
i = i + 1;

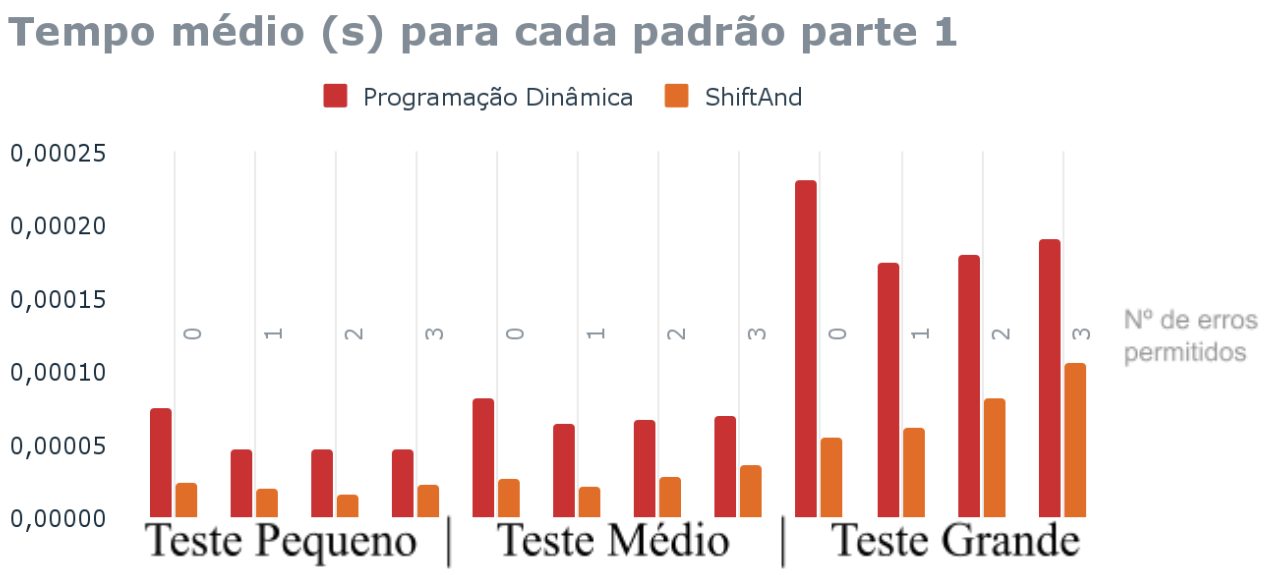
---

4. Testes e Resultados

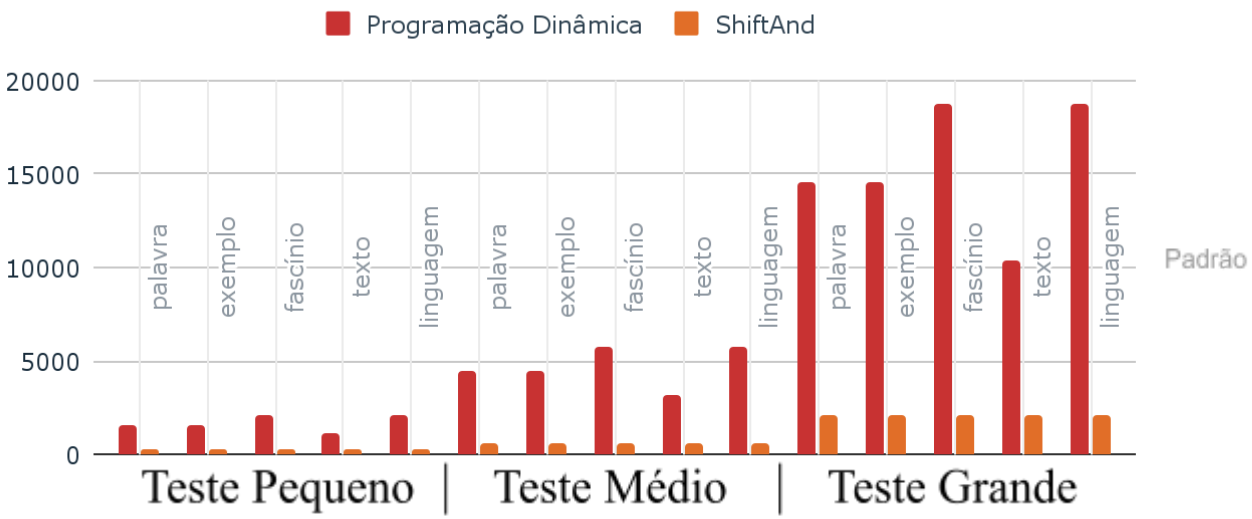
Ambiente de teste:
Sistema Operacional: Ubuntu 22.04 LTS
- CPU: Intel Core i5-8250U @ 1.60GHz × 8
- Memória: 8 GB RAM

Padrões:
palavra | exemplo | fascínio | texto | linguagem | comunicação

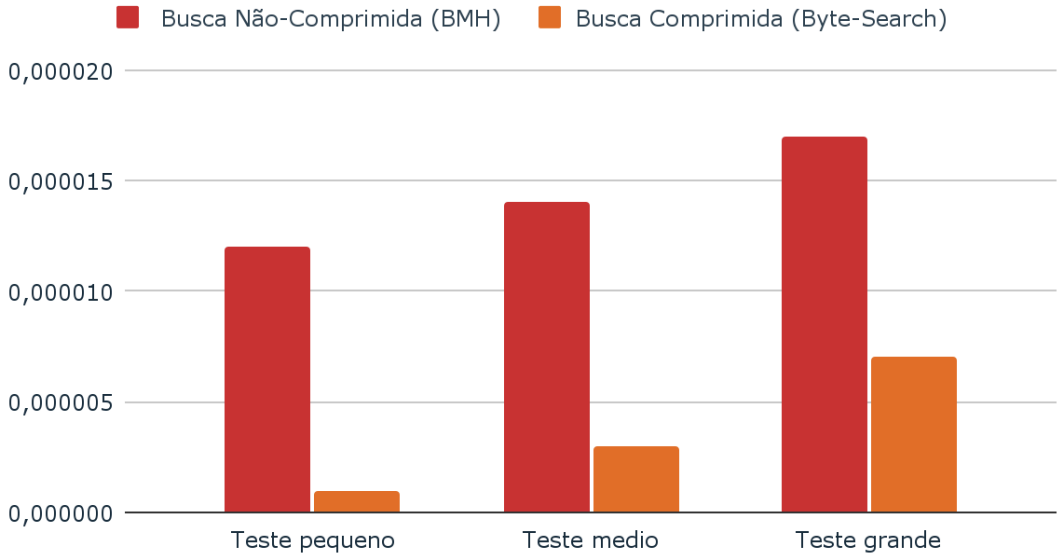
Testes:
Teste pequeno : 227 caracteres
Teste médio: 621 caracteres
Teste Grande: 2011 caracteres

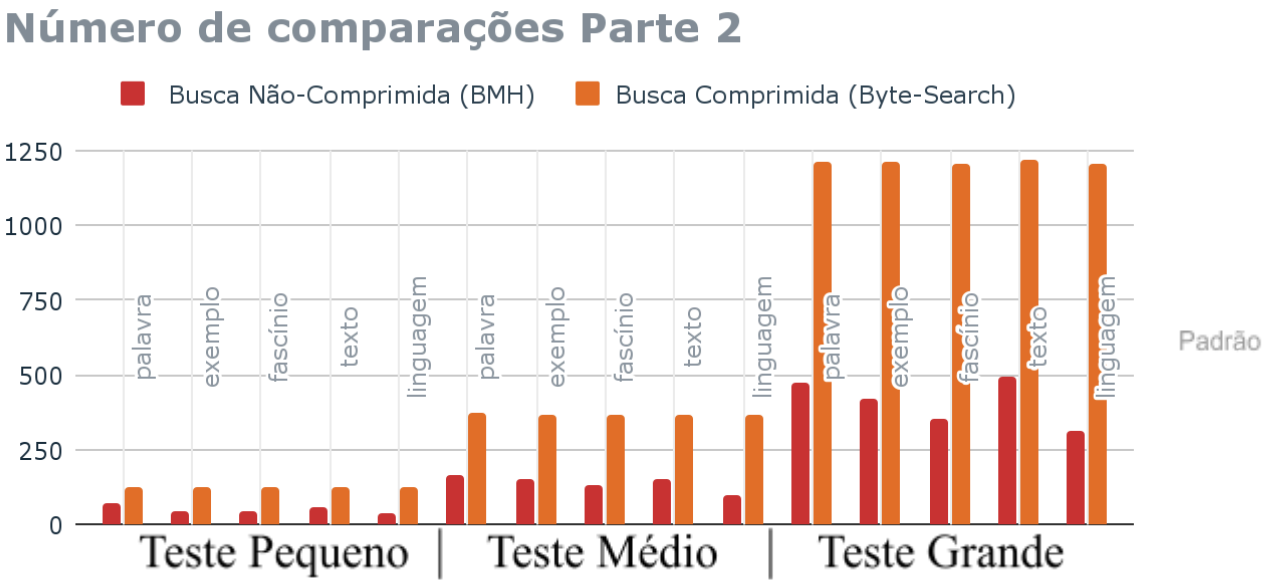


Número de comparações Parte 1



Tempo médio (s) para cada padrão parte 2





5. Compilação e execução

Para compilar, executar ou limpar os arquivos, é usado comandos do makefile:

make tp4_parte1 : Compila o arquivo “tp4_parte1.c” .

make tp4_parte1 : Compila o arquivo “tp4_parte2.c”.

make run : Roda 3 testes todos com o mesmo arquivo de padrões e com o mesmo texto, no caso “Arquivo_Padroes.txt” e “Arquivo_texto.txt”. Primeiro testa a parte 1 com o algoritmo Programação Dinâmica, depois testa a parte 1 com o algoritmo Shift And. Logo em seguida testa a parte 2.

make clean : limpa todos os arquivos gerados pela compilação e execução do programa.

6. Saída Gerada

Ao rodar o programa com : ./tp4_parte1 <algoritmo> <texto> <arquivo padroes>
Deverá aparecer algo desse tipo no terminal:
Executando Parte 1 com o algoritmo: ShiftAnd
Texto: Arquivo_texto.txt (62 caracteres)
Padrões: Arquivo_Padroes.txt

Padrão	k	Tempo (s)	Comparações
palavra	0	0.0000010	62
palavra	1	0.000007	62
palavra	2	0.000009	62
palavra	3	0.000009	62

…
A primeira linha informa qual algoritmo está rodando, depois exibimos o nome do arquivo e o número de caracteres, em seguida um cabeçalho de uma tabela, que informa para cada padrão e cada número de erros permitido: o tempo gasto naquela busca e o número de comparações realizadas.
E salva as posições de ocorrência com k=3 dentro do arquivo "saida_<algoritmo>_ocorrencias.txt", cada linha é:
<padrão> <pos1> <pos2> … <posn>
…
Ao rodar : ./tp4_parte2 <texto> <arquivo_padrões>
Deverá aparecer algo desse tipo no terminal:
Executando Parte 2: BMH (Não Comprimido) vs Busca (Comprimido)
Texto: Arquivo_texto.txt (62 caracteres)
Padrões: Arquivo_Padroes.txt

Compressão Huffman:
- Tempo de compressão: 0.000030 s
- Tamanho Original: 62 bytes
- Tamanho Comprimido: 32 bytes
- Taxa de Compressão: 48.39%

Padrão | Busca Não-Comprimida (BMH) | Busca Comprimida (Byte-Search)
| Tempo (s) | Comparações | Tempo (s) | Comparações
---
palavra | 0.000016 | 22 | 0.000002 | 31
exemplo | 0.000005 | 17 | 0.000003 | 30
…

A parte Compressão de Huffman mostra quanto tempo levou para comprimir o texto e o tamanho antes e depois, além da taxa de compressão(%).
A tabela compara para cada padrão: o tempo e o número de comparações do BMH no texto original e o tempo e número de comparações do "byte a byte" no texto já comprimido.
E salva as posições de cada padrão igual na parte 1 só que no arquivo "saida_parte2_ocorrencias.txt".

7. Análise de Complexidade

m representa o tamanho do padrão
n representa o tamanho do texto
k representa o número de erros
Σ representa o tamanho do alfabeto
h representa o número de nós na árvore

Algoritmo	Temporal	Justificativa
Programação Dinâmica	O(m*n)	Para cada caractere do texto (**n**), compara com cada caractere do padrão (**m**), preenchendo uma matriz de custos.
Shift-And	O(n*k)	Para cada posição do texto (**n**), realiza atualizações em **k**+1 máscaras de estado, uma para cada possível número de erros.
BMH	O(n)	Em média, salta **m** posições no texto a cada comparação; no pior caso, compara **m** caracteres a cada uma das **n** posições.
Huffman	O(n+ΣlogΣ)	Leva O(**Σ**log**Σ**) para construir a árvore com os **Σ** símbolos distintos, e O(**n**) para codificar o texto de **n** caracteres.

Algoritmo	Espacial	Justificativa
Programação Dinâmica	O(m)	Em vez de guardar a matriz inteira na memória (o que custaria O(**n*m**)), a implementação otimizada precisa apenas da coluna anterior para calcular a coluna atual.
Shift-And	O(Σ+k)	O algoritmo precisa armazenar, para cada caractere do alfabeto, uma máscara de bits que indica em quais posições esse caractere aparece no padrão.
BMH	O(Σ)	A memória extra utilizada é para a tabela de deslocamentos, que tem um

		tamanho fixo, dependente apenas do número de caracteres no alfabeto.
Huffman	O(Σ+h)	O espaço necessário é para armazenar a tabela de frequências, a própria árvore de Huffman e a tabela com os novos códigos de bits. Todas essas estruturas dependem apenas da quantidade de caracteres diferentes no alfabeto, e não do tamanho do texto.

8. Conclusão

Na primeira parte deste trabalho foi implementado e comparar dois algoritmos de busca aproximada, podemos comparar alguns parâmetros e entender seus impactos no tempo de execução:

- **Tamanho do Texto:** Ambos aumentam tempo conforme o tamanho aumenta, um exemplo é que no teste pequeno o padrão “palavra” de 0.000074s passou para 0.000330s no PD e de 0.000024s 0.000054s o Shift-And.
- **Número de erros permitidos (k)**: O número de erros impactou muito mais o Shift-And, diferente do Programação Dinâmica que praticamente não houve diferença. Por exemplo, o padrão “palavra”, no shift-And com k=0 demorou 0.000054s já com k=3 0.000106s, enquanto a PD mantém um tempo estável.

Comprovando a superioridade do Shift-And que em alguns casos foi até 6x mais rápido que o PD, e com uma melhor escalabilidade ideal com o crescimento do texto.

Na segunda parte foi implementado e comparado algoritmos de busca em arquivos comprimidos e não comprimidos, e com os resultados obtidos podemos afirmar que a compressão de Huffman, reduz o tamanho dos arquivos em 40 - 70% do tamanho original, e mesmo aumentando o número de comparações, obteve uma melhora na performance, pois é uma operação mais leve. Para sistemas de grande escala e robustos, com certeza a busca em arquivos comprimidos é uma ótima estratégia, equilibrando economia de armazenamento, e o desempenho.