Universidade Federal de São João del Rei
Campus Tancredo Neves - Ciência da Computação

TRABALHO PRÁTICO 3

Oscar Alves Jonson Neto
Geraldo Arthur Detomi

São João del-Rei
2025

Sumário

---

1. Introdução

Este projeto tem como objetivo apresentar o desenvolvimento do Trabalho Prático 3 da disciplina de Projeto e Análise de Algoritmos, ofertada pelo docente Leonardo Chaves Dutra da Rocha. O trabalho aborda a localização de padrões em arquivos de texto, considerando cenários com e sem compressão de dados.

1.1 Enunciado

A proposta do trabalho consiste em desenvolver e analisar um sistema para localizar padrões em arquivos de texto, abordando dois cenários distintos. O primeiro é a busca aproximada em arquivos não comprimidos, onde o objetivo é encontrar ocorrências de um padrão permitindo um determinado número K de erros. O segundo é a busca exata em arquivos comprimidos, onde o mesmo padrão deve ser encontrado diretamente em um arquivo que foi previamente comprimido.

1.2 Análise de entradas e saídas

Ambos os cenários recebem dois arquivos, um deles contendo o texto para busca e o outro contendo os padrões a serem procurados, separados linha por linha. Além disso, a parte 1 do TP, recebe também:

- a estratégia a ser utilizada (1 para Programação Dinâmica e 2 para Shift-And).

Para cada padrão procurado, a saída deve conter a lista de ocorrências.

- Exemplo de entrada de texto: "Texto exemplo, texto tem palavras, palavras exercem fascínio."
- Exemplo de padrões: "palavras", "exemplo"
- Exemplo de saída: "palavras 26 36", "exemplo 7"

2. Análise do Problema

2.1 Desafios e Componentes Fundamentais

Os principais desafios residem na capacidade de lidar com diferentes cenários de busca. A busca aproximada demanda algoritmos que calculam a distância de edição, permitindo K erros no casamento entre o padrão e o texto.

Por sua vez, a busca em arquivos comprimidos exige que o algoritmo opere diretamente sobre a representação compactada dos dados, o que implica um tratamento específico do padrão e do texto. A eficiência é uma métrica constante, buscando comparar o desempenho dos algoritmos em termos de tempo e número de comparações em todas as variações do problema.

Problemas de busca de padrões em textos comprimidos ou não, são um desafio fundamental na ciência da computação com aplicações em diversas áreas como processamento de texto e segurança.

2.2 Estratégias das Resoluções

Para abordar os desafios propostos neste trabalho, as estratégias a serem empregadas foram as instruídas no TP e se concentram em dois grandes grupos. Para a busca aproximada em arquivos não comprimidos, fomos orientados a utilizar o algoritmo de **Programação Dinâmica** e o algoritmo **Shift-And**. A **Programação Dinâmica**, conhecida por sua capacidade de encontrar a solução ótima, calcula a distância de edição entre o padrão e o texto. O **Shift-And**, por sua vez, oferece uma abordagem eficiente baseada em operações bit a bit para identificar ocorrências com tolerância a erros.

Já para a busca exata em arquivos, fomos direcionados a aplicar o algoritmo **Boyer-Moore-Horspool** (**BMH**). Este algoritmo será comparado em seu desempenho tanto no texto em sua forma original quanto após ser comprimido utilizando o código de **Huffman**.

2.2.1 Exemplo de funcionamento do Shift-And

Pesquisa do padrão P = {exemplo } no texto, T = {os exemplos... }.

Texto	(R >> 1) | 10^m-1							R'						
o	1	0	0	0	0	0	0	0	0	0	0	0	0	0
s	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	0
e	1	0	0	0	0	0	0	1	0	0	0	0	0	0
x	1	1	0	0	0	0	0	0	1	0	0	0	0	0
e	1	0	1	0	0	0	0	1	0	1	0	0	0	0
m	1	1	0	1	0	0	0	0	0	0	1	0	0	0
p	1	0	0	0	1	0	0	0	0	0	0	1	0	0
l	1	0	0	0	0	1	0	0	0	0	0	0	1	0
o	1	0	0	0	0	0	1	0	0	0	0	0	0	1
s	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	0

3. Desenvolvimento do Código

Os principais módulos que compõem a arquitetura do projeto são:

- entrada.c
- LSE.c
- resposta.c
- tempo.c
- strategydp.c
- strategy_shiftand.c
- huffmanbyte.c
- solucao_arq_descomp.c
- main.c

Vamos analisar mais a fundo os módulos principais que são: **strategydp.c**, **strategy_shiftand.c**, **solucao_arq_descomp.c**, **huffmanbyte.c** e **main.c**

3.1 strategydp.c

O módulo **strategydp.c** implementa a estratégia de Programação Dinâmica para a busca aproximada. A função, **encontrar_casamento_aproximados_dp**, é responsável por alocar uma matriz dinâmica para calcular a distância de edição entre o padrão e o texto. Esta matriz é preenchida considerando as operações de substituição, remoção e inserção, a função **buscar_casamentos_aproximados_dp** preenche essa matriz e identifica as posições onde a distância de edição é menor ou igual ao número de erros permitidos, inserindo-as na lista de soluções. Os métodos auxiliares **alocar_matriz_dp** e **liberar_matriz_dp** gerenciam a memória da matriz.

3.2 strategy_shiftand.c

O módulo **strategy_shiftand.c** contém a aplicação do algoritmo Shift-And para busca aproximada. A função principal **shift_and** utiliza máscaras de bits para cada caractere do alfabeto e um array de estados para controlar as ocorrências do padrão com diferentes números de erros permitidos. A função **encontrar_casamento_aproximados_sa** atua como uma interface para o algoritmo.

3.3 solucao_arq_descomp.c

Este módulo é dedicado à busca exata de padrões em arquivos de texto não comprimidos, utilizando o algoritmo Boyer-Moore-Horspool (BMH). A função principal **bmh_texto_descomprimido** implementa o BMH tradicional, que otimiza a busca através da construção de uma tabela de deslocamentos (d) para cada caractere. Isso permite que o algoritmo realize saltos maiores no texto ao invés de avançar caractere por caractere, reduzindo o número de comparações. A função **buscar_arquivo_descomprimido** serve como interface para o processo.

3.4 huffmanbyte.c

O módulo **huffmanbyte.c** implementa o algoritmo de compressão Huffman por byte e uma versão adaptada do Boyer-Moore-Horspool para busca em arquivos comprimidos. A função **compressao** coordena as três etapas principais do processo de compressão: construção do vocabulário, codificação e escrita dos dados compactados no arquivo. Funções como **define_alfabeto**, **primeira_etapa**, **segunda_etapa** e **terceira_etapa** gerenciam as fases de compressão. Para a busca em arquivos comprimidos, a função **processar_padrao** codifica o padrão para sua forma compactada e então chama **bmh_byte**, que é a versão do Boyer-Moore-Horspool adaptada para operar sobre o texto comprimido, para realizar a busca.

3.5 main.c

Existem duas implementações distintas para **main.c**, porém eles compartilham diversas funções essenciais. Ambos são responsáveis pela validação dos argumentos de linha de comando, pelo gerenciamento de arquivos, pela medição e exibição do tempo de execução, e pela liberação da memória alocada para as soluções com **destroi_solucao_casamento**, com todos os ponteiros de arquivo sendo fechados ao fim da execução.

O **main.c** da **Parte 1** lê a quantidade de erros a serem tolerados, e a estratégia de busca a partir dos argumentos de linha de comando. Ele então chama a função correspondente à estratégia escolhida: **encontrar_casamento_aproximados_sa** para Shift-And ou **encontrar_casamento_aproximados_dp** para Programação Dinâmica.

O **main.c** da **Parte 2** lê uma opção que determina se a busca será em arquivo comprimido ou não. Se a opção com compressão for selecionada, o código primeiramente chama

comprimir_arquivo_entrada, em seguida lê os dados comprimidos e realiza a busca utilizando **processar_padrao**. Caso a opção sem compressão seja selecionada, o texto original é lido e a busca é realizada através da função **buscar_arquivo_descomprimido**.

4. Testes

Agora que entendemos como o programa funciona, precisamos rodá-lo e analisar seus resultados e também sua complexidade.

4.1 Análise de Complexidade Busca Aproximada

A estratégia para busca aproximada foi desenvolvida visando encontrar ocorrências de um padrão com tolerância a um número k de erros. Sua análise de complexidade considera as seguintes variáveis principais:

- n (tamanho do texto)
- m (tamanho do padrão)
- k (número máximo de erros permitidos)
- $|\Sigma|$ (tamanho do alfabeto)

4.1.1 Função buscar_casamentos_aproximados_dp (Programação Dinâmica)

A função **buscar_casamentos_aproximados_dp** implementa o algoritmo de Programação Dinâmica para casamento aproximado. A complexidade desta função é dominada pelo preenchimento da matriz de dp. Essa matriz possui dimensões de $(m+1)$ por $(n+1)$, e cada célula é calculada em tempo constante.

Portanto, a complexidade de tempo total para a Programação Dinâmica é de:

$$O(m \cdot n)$$

4.1.2 Função shift_and (Shift-And)

A função **shift_and** inicializa uma tabela de máscaras de bits de tamanho 256 (representando os caracteres ASCII), um processo de tempo constante para um alfabeto fixo.

A complexidade de tempo do loop principal de busca é determinada pela iteração sobre cada caractere do texto (n vezes). Para cada caractere, o algoritmo realiza operações que atualizam todos os $k+1$ níveis de estados (do zero erro até k

erros), incluindo o caso base e os k níveis de erro (substituição, inserção, deleção e erro acumulado).

Portanto, a complexidade de tempo para o Shift-And com busca aproximada é de:

$$O(n \cdot (k+1))$$

Considerando que k pode ser até $m-1$, a complexidade pode ser expressa como $O(n \cdot m)$ no pior caso.

4.2 Análise de Complexidade Busca Exata

A estratégia de busca exata emprega o algoritmo Boyer-Moore-Horspool (BMH). A complexidade desta análise considera as seguintes variáveis:

- n (tamanho do texto original)
- m (tamanho do padrão)
- $|\Sigma|$ (tamanho do alfabeto, geralmente 256 para caracteres/bytes)
- n' (tamanho do texto comprimido)
- m' (tamanho do padrão comprimido, ou seja, o comprimento do código Huffman do padrão)

4.2.1 Função bmh_texto_descomprimido (Boyer-Moore-Horspool em texto não comprimido)

A função **bmh_texto_descomprimido** implementa o BMH para busca em textos originais. A fase de pré-processamento, que constrói a tabela de deslocamentos, tem uma complexidade de $O(|\Sigma|+m)$. Na fase de busca, o melhor cenário alcança uma complexidade de:

$$O(n/m)$$

Que é caracterizado por grandes saltos. Contudo, no pior caso, a complexidade pode ser de $O(n \cdot m)$.

4.2.2 Função bmh_byte (Boyer-Moore-Horspool em texto comprimido)

A função **bmh_byte** é a versão adaptada do BMH para operar sobre o texto comprimido por Huffman. A sua performance é intrinsecamente ligada ao tamanho do texto comprimido (n') e ao tamanho do padrão comprimido (m'). A função **processar_padrao** primeiro codifica o padrão para sua forma comprimida (determinando m'), antes de chamar **bmh_byte** para a busca. A complexidade desta busca segue os princípios do BMH, sendo sua complexidade no melhor caso:

$$O(n'/m')$$

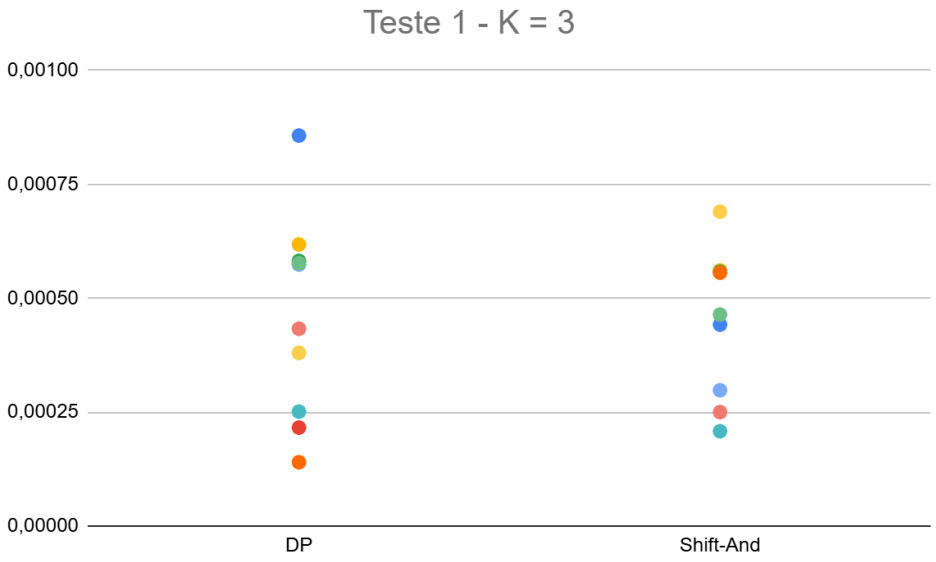
e $O(n' \cdot m')$ no pior caso. É fundamental considerar o custo da compressão inicial do texto (**compressao**) na análise do tempo total.

4.3 Análise de Resultados

Agora que entendemos a complexidade do código, vamos rodar os mesmos e analisar os resultados.

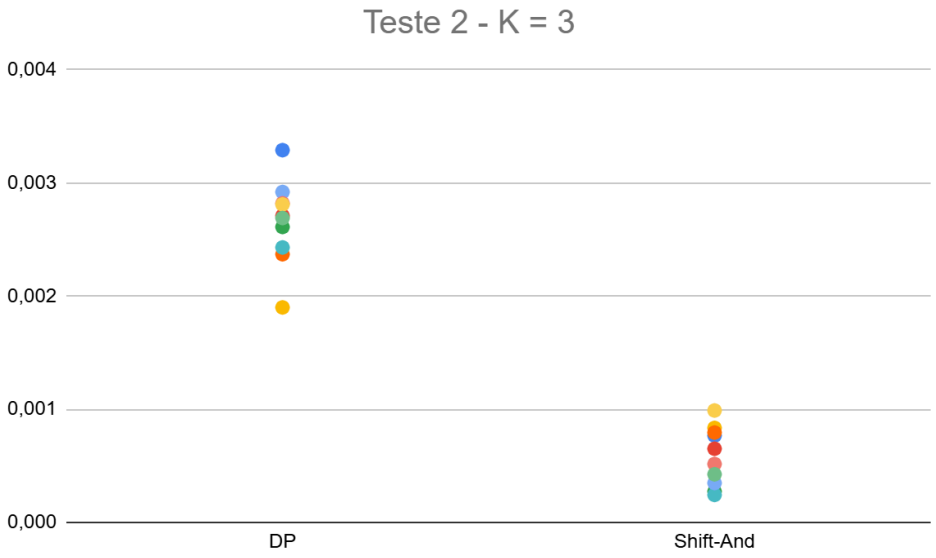
4.3.1 Análise da Parte 1

Os testes para a Parte 1 foram conduzidos em 10 repetições para cada estratégia todos utilizando $k = 3$, primeiro utilizaremos o conjunto de texto e padrão especificado no enunciado da documentação



No segundo teste iremos utilizar o seguinte texto:

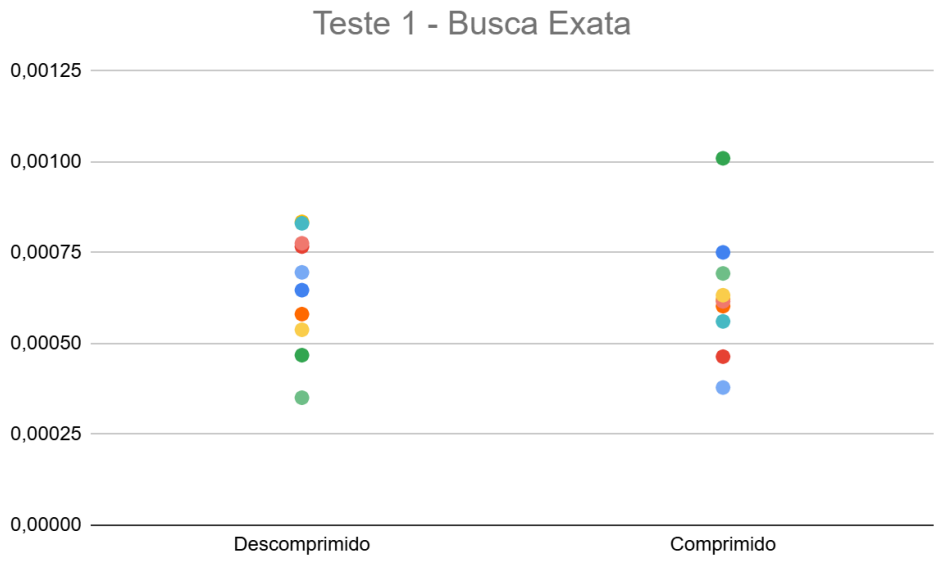
A vida é uma jornada, uma jornada cheia de desafios. Cada dia, um novo capítulo se inicia, repleto de oportunidades. Buscamos conhecimento, por vezes, com determinação incansável. Algoritmos complexos processam dados rapidamente, são eficientes. Dados gigantes, grandes volumes de informação a serem analisados. O texto contém palavras, números (como 123 e 456) e símbolos ($#@). Busca por padrões, a essência da análise de caracteres e textos. Jornada contínua, uma verdadeira aventura no mundo da programação. Com os seguintes padrões: *“uma” e “jornada”*.



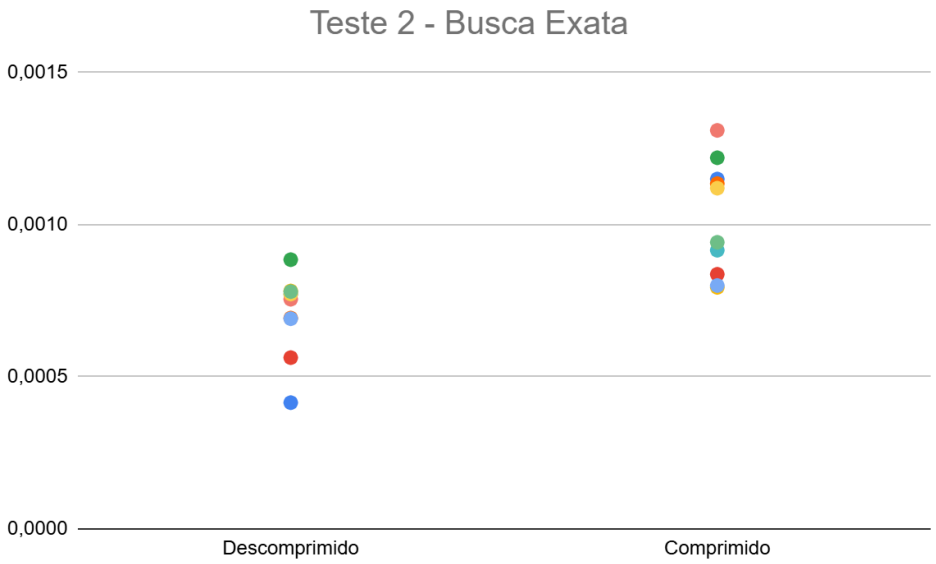
Como podemos ver, para um texto relativamente longo, a eficiência algorítmica do Shift-And proporciona um ganho de desempenho notável, atingindo uma velocidade até duas vezes superior no 2º teste.

4.3.2 Análise da Parte 2

Os testes para a Parte 2 também foram conduzidos em 10 repetições para cada estratégia (comprimido e descomprimido), primeiro utilizaremos o conjunto de texto e padrão especificado no enunciado da documentação.



No segundo teste iremos utilizar o mesmo texto utilizado na análise da parte 1, com os mesmo padrões.



Como podemos ver, em um texto pequeno o tempo de comprimir não influencia tanto no tempo final da execução, entretanto, à medida que o tamanho do texto cresce, o tempo de compressão se torna relevante, conforme ilustrado no gráfico acima.

5. Conclusão

Este trabalho prático sobre a busca de padrões em textos permitiu que explorassemos a localização de padrões em textos, cobrindo buscas aproximadas em dados originais e buscas exatas em arquivos comprimidos.

Busca Aproximada:

A Programação Dinâmica garantiu a **otimalidade** da solução, sendo ideal quando a precisão é prioritária.

O Shift-And se destacou pela **eficiência**, mostrando-se uma alternativa valiosa para cenários com poucos erros, onde a velocidade é crucial.

Busca Exata:

> A comparação do Boyer-Moore-Horspool (BMH) em arquivos originais versus comprimidos evidenciou o **dilema entre otimização de espaço e tempo de processamento**, ressaltando que a decisão deve ponderar o custo da compressão inicial em relação ao ganho na busca.

A seleção da estratégia mais adequada depende fundamentalmente dos requisitos específicos de cada aplicação, balanceando a necessidade de uma solução ótima com a demanda por tempo de resposta e uso eficiente de recursos.

6. Referências

[CORMEN et al., 2012] CORMEN, T. H., Leiserson, C., Rivest, R., and Stein, C (2012). Algoritmos: teoria e prática. LTC.

NIVIO ZIVIANI. Projeto De Algoritmos Com Implementações Em Pascal E C, 3a Ed. Rev. E Ampl. [s.l: s.n.].

Projeto de Algoritmos | Nivio Ziviani» implementações. Disponível em: <https://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-08.php>. Acesso em: 27 jun. 2025.