Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas (ICEA)
Departamento de Computação e Sistemas

Algoritmos e Estruturas de Dados II

Análise Comparativa de Algoritmos de Casamento de Padrões

Aluno: João Victor Teixeira Pereira. **Matrícula:** 22.1.8052
Professor: Rafael Frederico Alexandre

19 de fevereiro de 2026

Sumário

1 Introdução **3**

2 Fundamentação Teórica **3**
2.1 Força Bruta 3
2.2 Rabin-Karp 3
2.3 Knuth-Morris-Pratt (KMP) 3
2.4 Família Boyer-Moore 4

3 Metodologia **4**
3.1 Implementação e Ambiente 4
3.2 Cenários de Teste 4

4 Resultados e Análise **5**
4.1 Análise Crítica dos Resultados 5
4.1.1 Desempenho da Família Boyer-Moore 5
4.1.2 A Anomalia do Sunday em Alfabetos Pequenos 6
4.1.3 KMP vs Força Bruta e o Custo da Linguagem 6
4.1.4 A Ineficiência Prática do Rabin-Karp 6

5 Conclusão **6**

1 Introdução

A busca de padrões em strings é um problema clássico da Ciência da Computação, com aplicações diretas em editores de texto, ferramentas de busca, bioinformática, detecção de plágio e sistemas de recuperação da informação. O problema consiste, formalmente, em encontrar todas as ocorrências de um padrão P de tamanho m dentro de um texto T de tamanho n.

Este trabalho prático tem como objetivo implementar, comparar e analisar o comportamento de diferentes algoritmos de busca de padrões, focando tanto na complexidade teórica quanto no desempenho empírico em diferentes cenários.

Os algoritmos analisados são:

- Força Bruta
- Rabin-Karp
- Knuth-Morris-Pratt (KMP)
- Boyer-Moore
- Boyer-Moore-Horspool
- Boyer-Moore-Horspool-Sunday

2 Fundamentação Teórica

2.1 Força Bruta

O algoritmo de Força Bruta verifica todas as posições possíveis do texto onde o padrão pode iniciar. Ele compara caractere a caractere até encontrar uma discrepância ou uma correspondência completa. Embora simples, possui complexidade de pior caso $O(n \cdot m)$.

2.2 Rabin-Karp

O algoritmo de Rabin-Karp utiliza uma função de *hash* para comparar o padrão com substrings do texto. A comparação caractere a caractere só ocorre quando os valores de hash coincidem. Sua complexidade média é $O(n + m)$, mas o pior caso pode chegar a $O(n \cdot m)$ em situações de muitas colisões de hash.

2.3 Knuth-Morris-Pratt (KMP)

O KMP evita comparações redundantes utilizando uma tabela de prefixos (função de falha) que indica o maior prefixo próprio do padrão que também é sufixo. Isso permite que o algoritmo saiba o quanto pode avançar no texto sem retroceder o índice de leitura, garantindo complexidade $O(n + m)$ no tempo e $O(m)$ no espaço.

2.4 Família Boyer-Moore

Esta família de algoritmos realiza a comparação da direita para a esquerda, permitindo saltos maiores que um caractere.

- **Boyer-Moore:** Utiliza duas heurísticas combinadas: o *mau-caractere* e o *bom-sufixo*. No melhor caso, sua complexidade é sublinear ($O(n/m)$).
- **Horspool:** Uma simplificação que utiliza apenas a heurística do *mau-caractere*. É conhecido por ser mais simples de implementar e muitas vezes tão rápido quanto o original em alfabetos grandes.
- **Sunday:** Uma variação do Horspool que considera o caractere imediatamente *após* a janela de comparação para determinar o deslocamento, visando maximizar o tamanho do salto.

3 Metodologia

3.1 Implementação e Ambiente

Os algoritmos foram implementados na linguagem **Python**. Os testes foram realizados em um ambiente computacional pessoal, medindo-se o tempo de CPU (em segundos) para a execução completa da busca.

3.2 Cenários de Teste

Foram definidos três cenários distintos para avaliar o comportamento dos algoritmos sob diferentes condições de tamanho de entrada e alfabeto:

1. **Cenário 1 (Texto Pequeno):** Texto de 10.000 caracteres, padrão de 5 caracteres, alfabeto ASCII completo.
2. **Cenário 2 (Texto Médio):** Texto de 100.000 caracteres, padrão de 50 caracteres, alfabeto ASCII completo.
3. **Cenário 3 (DNA / Pior Caso para Heurísticas):** Texto de 500.000 caracteres, padrão de 100 caracteres, alfabeto reduzido (A, C, G, T). Este cenário simula bioinformática e testa a eficiência dos saltos em alfabetos pequenos.

4 Resultados e Análise

Os tempos de execução médios obtidos nos experimentos estão detalhados na Tabela 1 e ilustrados na Figura 1.

Tabela 1: Comparativo de Tempo de Execução (em segundos)

Algoritmo *Configuração*	**Cenário 1 (10k)** *ASCII / Padrão 5*	**Cenário 2 (100k)** *ASCII / Padrão 50*	**Cenário 3 (500k)** *DNA / Padrão 100*
Força Bruta	0.0053	0.0350	0.1483
Rabin-Karp	0.0110	0.0753	0.2878
KMP	0.0043	0.0380	0.1766
Boyer-Moore	0.0030	0.0040	**0.0823**
Horspool	0.0027	0.0023	0.0906
Sunday	**0.0023**	**0.0017**	0.1053

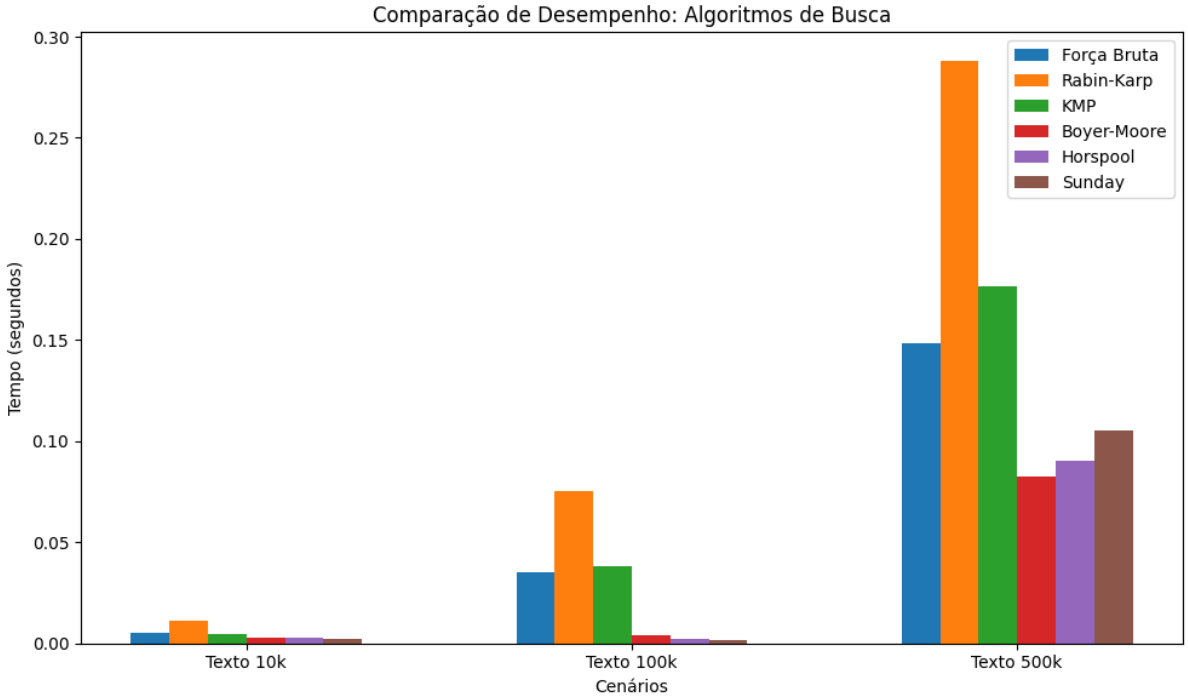

Figura 1: Gráfico comparativo de desempenho entre os algoritmos.

4.1 Análise Crítica dos Resultados

4.1.1 Desempenho da Família Boyer-Moore

Como previsto pela teoria, a família Boyer-Moore dominou os testes de desempenho. Nos cenários de texto comum (1 e 2), o algoritmo **Sunday** mostrou-se o mais rápido, aproveitando-se de saltos maiores. No entanto, no Cenário 3 (DNA), o **Boyer-Moore Completo** (0.0823s) superou tanto o Horspool quanto o Sunday.

4.1.2 A Anomalia do Sunday em Alfabetos Pequenos

Observou-se que no Cenário 3, o algoritmo Sunday (0.1053s) foi mais lento que o Horspool (0.0906s). Isso ocorre devido ao tamanho reduzido do alfabeto (apenas 4 caracteres). O Sunday decide o salto olhando o caractere *após* a janela. Com poucas letras disponíveis, a probabilidade desse caractere existir no padrão é alta, o que impede os "saltos gigantes"típicos do algoritmo. O custo computacional extra de acessar a memória fora da janela, sem o benefício dos grandes saltos, degradou seu desempenho.

4.1.3 KMP vs Força Bruta e o Custo da Linguagem

Um resultado contraintuitivo foi o KMP apresentar tempos superiores à Força Bruta (ex: 0.1766s vs 0.1483s no Cenário 3). Teoricamente, o KMP é $O(n)$ e a Força Bruta $O(n \cdot m)$. Contudo, em textos aleatórios, a Força Bruta raramente retrocede, comportando-se quase linearmente. Como a implementação foi feita em Python (linguagem interpretada), o *overhead* de gerenciar a tabela de estados do KMP custou mais tempo de processamento do que o loop simples e otimizado da Força Bruta.

4.1.4 A Ineficiência Prática do Rabin-Karp

O Rabin-Karp foi consistentemente o pior algoritmo (0.2878s no DNA). O custo de calcular o *Rolling Hash* — que envolve aritmética de grandes números e operações de módulo a cada caractere — mostrou-se extremamente pesado em Python, tornando-o inviável para busca simples de texto quando comparado a qualquer outra opção.

5 Conclusão

Com base nos experimentos realizados e na análise dos dados, conclui-se:

- **Melhor Desempenho Geral:** A família Boyer-Moore é imbatível na prática. O algoritmo **Horspool** oferece o melhor equilíbrio, sendo muito rápido e extremamente simples de implementar.
- **Simplicidade vs. Eficiência:** Para textos aleatórios comuns, a **Força Bruta** ainda é uma opção viável e até mais rápida que algoritmos complexos como o KMP, devido à simplicidade de suas instruções de máquina.
- **Compensação do Pré-processamento:** O custo de pré-processamento do Boyer-Moore compensa amplamente, pois permite pular grandes seções do texto. Já o pré-processamento do KMP não se justificou nestes testes práticos com dados aleatórios.
- **Recomendação:** Para implementações gerais em linguagens de alto nível, recomenda-se o uso do **Boyer-Moore-Horspool** devido à sua facilidade de manutenção e alta performance, exceto em casos muito específicos de alfabetos minúsculos onde o Boyer-Moore completo pode ter ligeira vantagem.