Análise sobre Algoritmos de Casamento Padrão

Eduardo G. S. Cardoso [1]

[1]Instituto de Ciências Exatas e Naturais - Universidade Federal do Pará (UFPA)

Resumo. *Este artigo busca fazer a apresentação de alguns algoritmos de casamento padrão em strings, sendo eles: Força Bruta, Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, Shift-And exato e Shift-And aproximado. No mais, também será realizada uma análise sobre a complexidade de cada um desses algoritmos na realização de uma mesma tarefa. Com o decorrer da analise, é possível inferir que enquanto alguns dos algoritmos contam com uma complexidade dependente do tamanho do padrão a ser buscado, outros dependem apenas do tamanho do texto, de forma que é podemos atribuir diferentes qualidades aos algoritmos, e inferir situações onde seu uso seria mais efetivo.*

Abstract. *This paper seeks to make a presentation of some string pattern matching algorithms, such as: Brute-Force, Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, exact Shift-And and approximated Shift-And. Furthermore, an analysis will be performed on the complexity of each of these algorithms in performing the same task. As the analysis go, it is possible to infer that while some of the algorithms have a complexity dependent on the size of the pattern to be searched, others depend only on the text size, in a way that we can assign different qualities to the algorithms, and infer situations where its use would be more effective.*

1. Introdução

Já há algum tempo, encontrar as ocorrências de um determinado padrão numa *string* (cadeia de caracteres) tomou o interesse da computação. Seja essa string um texto, um código, ou uma sequência de DNA, sempre será conveniente ter um algoritmo à mão que seja capaz de encontrar certos padrões de interesse.

Conforme essas strings aumentam, se faz necessária a utilização de algoritmos mais e mais eficientes, para que uma busca em determinada string seja possível em tempo hábil. Dessa forma, mecanismos de busca que precisam realizar buscas em milhões de amostras em frações de segundo não poderiam contar com um algoritmo quadrático para dar uma resposta a seus usuários num período de tempo satisfatório.

Ainda nessa questão dos mecanismos de busca, nos deparamos com uma outra situação. Realizar buscas exatas de padrões em uma amostra extensa nem sempre deve retornar tudo o que esperamos. Existem variantes, como as flexões de uma determinada palavra, e possíveis erros por parte da entrada do usuário. Percebemos aí que é interessante também a existência de um método para uma busca *aproximada* de um determinado padrão, para aumentar a chance de encontrarmos o que estamos buscando.

Neste artigo, realizaremos testes com alguns algoritmos que vêm solucionar essa questão da busca por padrões em strings. Cada algoritmo será testado com entradas iguais, e seus tempos serão calculados e analisados. Serão realizados testes com textos de 500 a 5000 caracteres (adquiridos por um gerador online), e três padrões de diferentes tamanhos, sendo os padrões: ”justo”, ”ipsum ultrices”e ”Quisque eget ligula in tortor commodo”.

2. Algoritmos

Os algoritmos de casamento padrão analisados neste artigo são:

1. Força Bruta
2. Boyer-Moore-Horspool
3. Boyer-Moore-Horspool-Sunday
4. Shift-And Exato
5. Shift-And Aproximado

Vamos agora conhecer um pouco sobre cada um deles.

2.1. Força Bruta

O algoritmo de força bruta, obviamente, faz uso de uma abordagem baseada em força bruta. Servindo para casamento exato, o algoritmo é simples, e vai de caractere em caractere de um texto T procurando por uma certa string S.

Como deve-se imaginar, este algoritmo não é um dos mais eficientes, visto que não há nenhum tipo de aproveito de informação conforme o algoritmo corre sobre S. Ou seja, sempre que ele passar para o caractere seguinte ele realiza o mesmo procedimento, independente de qualquer informação que a iteração partindo do caractere anterior pudesse ter lhe dado.

2.2. Boyer-Moore-Horspool (BMH)

O algoritmo Boyer-Moore-Horspool se apresenta numa simplificação do algoritmo Boyer-Moore, de forma que, para compreendê-lo, precisamos antes entender a base do algoritmo Boyer-Moore.

”An algorithm is presented that searches for the location, ’i,’ of the first occurrence of a character string, ’pat,’ in another string, ’string.’ During the search operation, the characters of pat are matched starting with the last character of pat. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched.”[Boyer and Moore 1977]

Como podemos perceber pela definição acima, ao contrário do algoritmo de força bruta, o algoritmo BM se beneficia de informações obtidas em seus estados anteriores, permitindo que o algoritmo ”pule”alguns caracteres ao longo de suas iterações, evitando checagens desnecessárias.

A diferença entre o BM e o BMH é que, enquanto que o BM realiza deslocamentos baseados somente em colisões, o BMH cria uma tabela de deslocamentos, onde guarda valores para cada caractere do alfabeto, e se baseia nessa tabela para realizar seus deslocamentos, tomando o valor de deslocamento do caractere do texto relativo ao ultimo caractere no padrão.

2.3. Boyer-Moore-Horspool-Sunday (BMHS)

Tal qual o algoritmo BMH veio simplificar o BM, o algoritmo Boyer-Moore-Horspool-Sunday apresenta uma simplificação do BMH. O conceito aqui é bem parecido, também fazendo uso de uma tabelas de deslocamento. Contudo, em vez de usar o valor de deslocamento do caractere do texto relativo ao ultimo caractere no padrão, o algoritmo BMHS toma o valor do próximo caractere no texto(fora da janela), para realizar seu deslocamento.

2.4. Shift-And Exato

Se valendo do paralelismo de bits, o método de Shift-And tira proveito das operações binárias realizadas pela máquina para atualizar valores ao longo da busca numa única operação.

Com isso, o algoritmo consegue encontrar todas as ocorrências de um padrão correndo o texto uma única vez, atualizando seus valores a cada caractere. Para tornar isso realidade, o algoritmo faz uso de máscaras bits, que são usadas em operações com o valor resultado da operação anterior para a obtenção de um novo resultado. Caso o último bit da palavra resultado seja 1, há um casamento exado se encerrando naquele ponto.

2.5. Shift-And Aproximado

Ainda utilizando a mesma estratégia de operações binárias, esse método se diferencia do Shift-And Exato (e de todos os outros previamente apresentados) pelo fato de ele buscar não somente por um casamento exato, mas também por um casamento aproximado, levando em consideração possíveis remoções, adições ou substituições no padrão original.

Isso é possível por conta de algumas operações binárias adicionais que são realizadas, e que conseguem manter as ”possibilidades”de casamento aproximado na palavra resultado ao longo das iterações. Neste artigo, faremos uso de um algoritmo Shift-And que tolera um máximo de 2 operações, sejam elas se remoção, adição ou substituição.

3. Ambiente Computacional Utilizado

Para realizar a análise dos algoritmos supracitados, foi utilizado um notebook Dell Inspiron 15, com um processador Intel Core i7 de 8ª Geração e 8 GB de Memória RAM. Os algoritmos foram implementados e analisados com uso da linguagem interpretada Python, na versão 3.7, executado sobre o sistema operacional Windows 10.

4. Resultados Obtidos

Primeiramente, foram realizados os testes buscando pelo padrão ”justo”. A Figura 1 nos mostra o tempo de execução (em milissegundos) de cada algoritmo, em relação ao tamanho do texto. A linha de regressão nos dá uma ideia do comportamento assintótico dos algoritmos (linear).

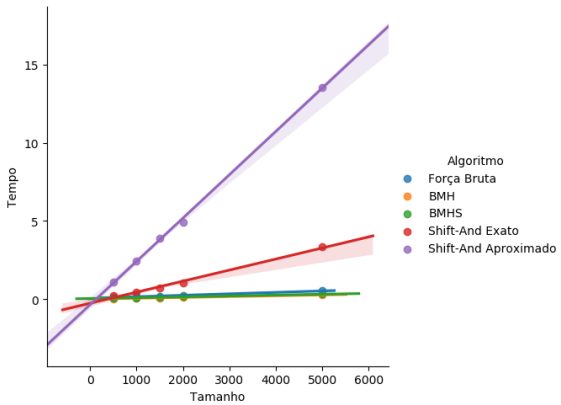

Figura 1. Busca por ”justo”

Ainda na Figura 1, é possível notar que os algoritmos Shift-And apresentam um comportamento que gera uma linha de regressão com um coeficiente angular consideravelmente maior que a dos demais algoritmos. Isso se deve à maior quantidades de instruções básicas executadas por esses algoritmos, considerando que os testes foram realizados por uma linguagem interpretada, mais distante das operações binárias, o que nega algumas das vantagens dos algoritmos.

Para facilitar a visualização das curvas dos algoritmos que não fazem uso do Shift-And, vamos tirá-los do quadro na Figura 2. Nesta figura, percebemos uma certa inconstância nos tempos de execução do algoritmo de força bruta, provavelmente relacionado ao *backtracking* envolvido no algoritmo, que diminui o tempo de execução de algumas iterações, descartando possibilidades que já não satisfazem o padrão buscado, enquanto que os algoritmos derivados do Boyer-Moore sem mostram bem mais constantes, deviando pouco da sua linha de regressão.

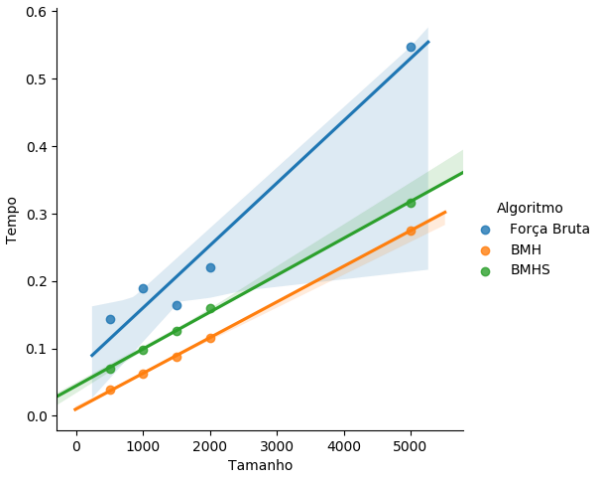

Figura 2. Busca por ”justo”

Vamos analisar agora o que acontece com o comportamento destes algoritmos quando aumentamos o tamanho do padrão buscado.

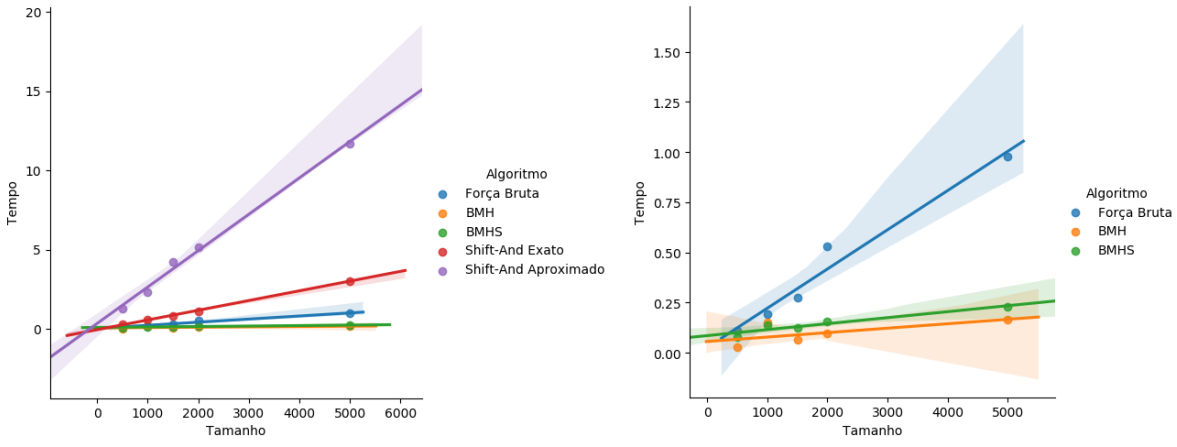

Figura 3. Busca por ”ipsum ultrices”

Comparando a Figura 1 à Figura 3, percebemos pouca mudança na curva dos algoritmos Shift-And, o que demonstra que o comportamento assintótico destes algoritmos depende não do comprimento do padrão buscado, mas sim do tamanho do texto onde se busca. Isso é esperado, visto que os algoritmos Shift-And apenas correm sobre o texto, fazendo as mesmas operações binárias em cada caractere.

Observando agora a Figura 3 em relação à Figura 2, podemos perceber claramente um mudança nas linhas de regressão. O algoritmo de força bruta se afasta agora do BMH e BMHS. Isso ocorre porque, conforme o tamanho da entrada vai aumentando, mais comparações o algoritmo de força bruta deve efetuar a cada iteração. Em contrapartida, o comportamento dos algoritmos BMH e BMHS geram retas na Figura 3 com um coeficiente angular levemente menor que na Figura 2, em consequência do fato que agora a janela sobre a qual o algoritmo é executado realiza deslocamentos maiores, já que a chave de busca é maior.

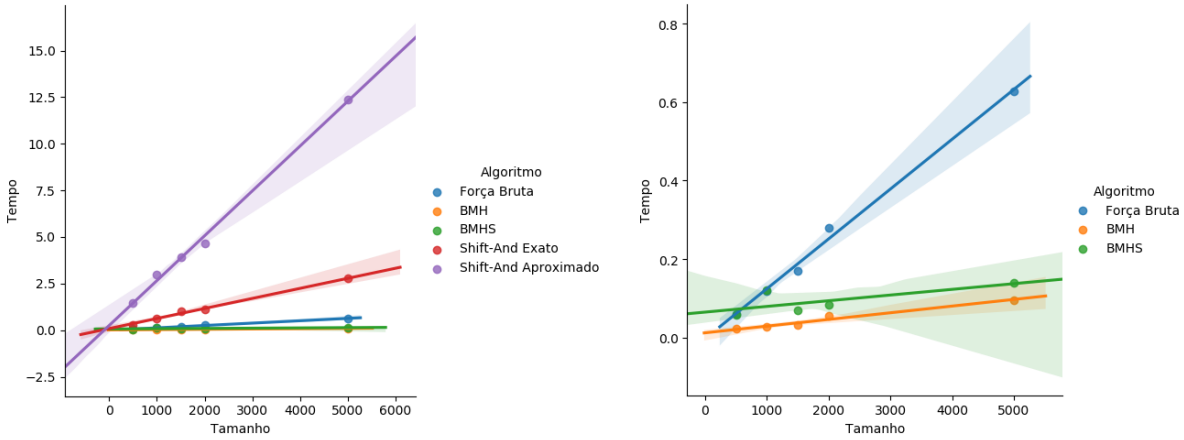

Figura 4. Busca por ”Quisque eget ligula in tortor commodo”

A Figura 4 apresenta o resultado dos testes realizados para um entrada ainda maior. Note que o comportamento dos algoritmos Shift-And é preservado, atendo sua complexidade somente ao tamanho do texto.

No mais, podemos perceber que o algoritmo de força bruta continua se afastando dos BM, que mostram em sua linha de regressão agora um coeficiente angular ainda menor. Observando os algoritmos BM mais atentamente, nota-se que o BMH apresenta um ritmo de crescimento maior que o BMHS, tornando possível inferir que para entradas maiores, ou mesmo textos maiores, o algoritmo BMHS se mostraria mais eficiente que o BMH, devido à diferença (mesmo que sutil), no tamanho dos seus deslocamentos.

5. Conclusão

A partir da análise feita e das deduções realizadas sobre os gráficos apresentados, podemos tirar algumas conclusões.

Primeiramente, podemos concluir a estabilidade dos algoritmos Shift-And. Note que, mesmo esses algoritmos aparentando ser menos eficiente nos gráficos apresentados, isso se deve em grande parte ao fato dos testes serem realizados com uma linguagem interpretada, distante dos bits. A despeito disso, é extremamente conveniente o uso de

algoritmos tão estáveis. Poder contar com o fato que o algoritmo vai sempre concluir sua execução sobre um texto com mesmo custo é extremamente interessante.

No que tange os demais algoritmos, podemos inferir que o algoritmo de força bruta é interessante talvez para entradas menores, mas um tanto quanto custoso para entradas maiores. Em contrapartida, os algoritmos BM se mostram bem eficientes para entradas maiores, fazendo uso dos seus deslocamentos, sendo que, conforme o texto e/ou a entrada vão ficando maiores, o algoritmo BMHS se mostra mais eficiente que o BMH.

Referências

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

```
def forca_bruta(P, T):
    """
    Realiza a busca pelo método de força brura
    :param str T:
    :param str P:
    :return int[]:
    """

    m = len(P)
    n = len(T)

    r = []

    for i in range(n - m + 1):

        k, j = i, 0

        while j < m and T[k] == P[j]:
            j += 1
            k += 1

        if j == m: r.append(i)

    return r

def bmh(P, T):

    """
    Realiza a busca pelo método Boyer-Moore-Horspool
    :param str T:
    :param str P:
    :return int[]:
    """

    m = len(P)
    n = len(T)

    r, d = [], [m for _ in range(256)]

    for i in range(m-1): d[ ord(P[i]) ] = m - i - 1

    i = m - 1

    while i < n:

        k, j = i, m-1
```

```
        while j >= 0 and T[k] == P[j]:

            j -= 1
            k -= 1

        if j < 0: r.append(k+1)

        i += d[ ord(T[i]) ]

    return r

def bmhs(P, T):

    """
    Realiza a busca pelo método Boyer-Moore-Horspool-Sunday
    :param str T:
    :param str P:
    :return int[]:
    """

    m = len(P)
    n = len(T)

    r, d = [], [ m + 1 for _ in range(1000)]

    for i in range(m): d[ ord(P[i]) ] = m - i

    i = m - 1

    while i < n-1:

        k, j = i, m - 1

        while j >= 0 and T[k] == P[j]:

            j -= 1
            k -= 1

        if j < 0: r.append(k+1)

        i += d[ ord(T[ i + 1 ]) ]

    return  r
```

```
def shift_and(P, T):

    """
    Realiza a busca por meio de shift-and exato
    :param str P:
    :param str T:
    :return int[]:
    """

    r = []

    m = len(P)
    n = len(T)

    M = {}

    for c in P:
        M[c] = 0


    for j in range(m):
        M[P[j]] = M[P[j]] | 2**(m-j-1)


    R = 0

    for i in range(n):

        if T[i] in P: R = ((R >> 1) | 2 ** (m-1)) & M[T[i]]
        else: R = ((R >> 1) | 2 ** (m-1)) & 0

        if R & 1 != 0 :r.append(i - m + 1)

    return r


def shift_and_aprox(P, T, k = 2):

    """
    Realiza a busca por meio de shift-and exato
    :param str P:
    :param str T:
    :param int k:
    :return int[]:
    """
```

```
r = []

m = len(P)
n = len(T)

M = {}



for c in P:
    M[c] = 0


for j in range(m):
    M[P[j]] = M[P[j]] | 2**(m-j-1)


R = [0 for _ in range(k+1)]

for j in range(k+1):
    R[j] = (2**j -1) * 2**(m-j)

for i in range(n):

    Ra = R[0]

    if T[i] in P: Rn =  ((Ra >> 1) | 2 **(m-1)) & M[T[i]]
    else: Rn = ((Ra >> 1) | 2 **(m-1)) & 0

    R[0] = Rn

    for j in range(1,k+1):

        if T[i] in P: Rn = ((R[j] >> 1)&M[T[i]]) | Ra |
            ((Ra | Rn) >> 1)

        else: Rn = ((R[j] >> 1) & 0) | Ra | ((Ra | Rn) >> 1)

        Ra= R[j]

        R[j] = Rn | 2**(m-1)

    if Rn & 1 != 0 :r.append(i)

return r
```