Algoritmos de *string matching*

Pedro E. Melha Lemos[1]

[1]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Faculdade de Informática -– Bacharelado em Ciência da Computação
Av. Ipiranga, 6681 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS – Brazil

pedro.elpidio@acad.pucrs.br

Resumo. *Este relatório apresenta alguns dos algoritmos de string matching, seus exemplos de execução e aplicações.*

1. Introdução

O problema de *string matching* pode ser descrito basicamente em como encontrar todas as ocorrências de um padrão ou sequência de caracteres – sendo este uma palavra ou frase – em um documento de texto, de acordo com [CORMEN et al. 2009], em que o algoritmo empregado para tal busca de ocorrências pode vir a implicar na responsividade do programa de edição de texto.

Neste relatório é apresentado inicialmente uma abordagem ingênua sobre o problema e então alguns outros algoritmos, como o algoritmo de *Rabin-Karp*, o algoritmo de *Knuth-Morris-Pratt* e o algoritmo de busca de subcadeias baseda em autômatos finitos. Para fins de formalização deste problema, seguindo o mesmo descrito em [CORMEN et al. 2009], o texto T é uma sequência de caracteres, cujo o comprimento é n, e que o padrão P é também uma sequência de caracteres, cujo o comprimento é m e é menor ou igual n.

Os caracteres tanto em T quanto em P existem em um alfabeto Σ e deve-se considerar que o padrão P pode ou não existir em T. Caso o padrão P exista em T, considerando que ocorra só uma vez, podemos dizer que há um ”deslocamento”em T em que P está igualmente representado a partir deste deslocamento, como o apresentado na Figura 1.

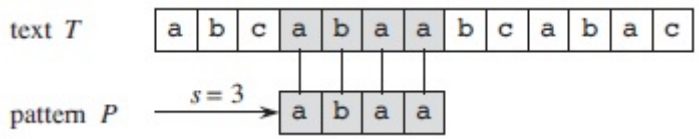

Figura 1. Deslocamento s em T dado o padrão P retirado de [CORMEN et al. 2009]

De acordo o mesmo autor, os algoritmos mencionados anteriomente (com exceção da abordagem ingênua) aplicam um preprocessamento no padrão para realizar a busca dos deslocamentos válidos.

2. Algoritmo ingênuo

Na abordagem ingênua são feitas tentativas de encontrar P em uma ”fatias”de tamanho m de T deslocando-se do início de T à $n - m$. Caso sejam encontradas, são impressos os deslocamentos.

```
naive_string_matcher(T, P)
    n := T.length
    m := P.length

    for s = 0 to n - m
        if P[0:m] is T[s:s + m]
            print("Padrão ocorre com deslocamento" + s)
```

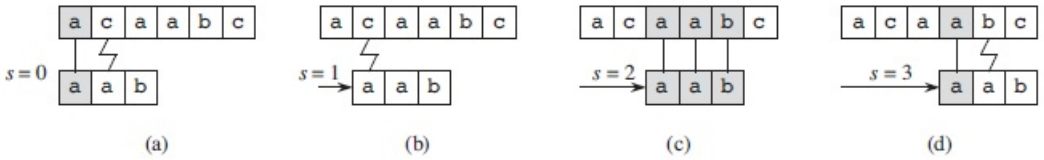

Figura 2. Exemplo de execução do algoritmo ingênuo retirado de [CORMEN et al. 2009]

A Figura 2 ilustra a execução passo a passo desse algoritmo. Este algoritmo possui complexidade de tempo $O((n - m + 1) * m)$ e, de acordo com [CORMEN et al. 2009], isso pode significar que no pior caso seja $\Theta(n^2)$. O mesmo autor menciona que o pior caso é quando T possui caracteres em que Σ possui somente um único símbolo. Isso implica que será encontrado um "*match*"do padrão P a cada deslocamento.

3. Algoritmo de *Rabin-Karp*

Este algoritmo, ao contrário do algoritmo ingênuo, possui um preprocessamento antes de realizar o "*matching*". De acordo com [CORMEN et al. 2009], este algoritmo executa bem na prática e também generaliza para outros problemas relacionados, como o *pattern matching* em arrays de duas dimensões. Este algoritmo necessita de mais outros dois argumentos:

- $d = |\Sigma|$
- q : um número primo

Dessa forma ele os utiliza para calcular e aplicar uma função de *hashing* sobre o padrão P e os primeiros m caracteres de T durante a etapa de preprocessamento. Durante a etapa de "*matching*"é realizada a comparação do valor do *hash* sobre P com os m caracteres de T no intervalo $[s, s + m]$ sendo que s varia de 0 à m.

```
rabin_karp_matcher(T, P, d, q)
    n := T.length
    m := P.length
    h := dˆ(m-1) mod q
    p := 0
    t := 0

    for i = 0 to m              # preprocessing:
        p := (d*p + P[i]) mod q
        t := (d*t + T[i]) mod q
```

```
for s = 0 to n - m          # matching:
    if p = t and P[0:m] is T[s:s + m]
            print("Padrão ocorre com deslocamento" + s)

    if s < n - m
        t := (d*(t - T[s]*h) + T[s + m]) mod q
```

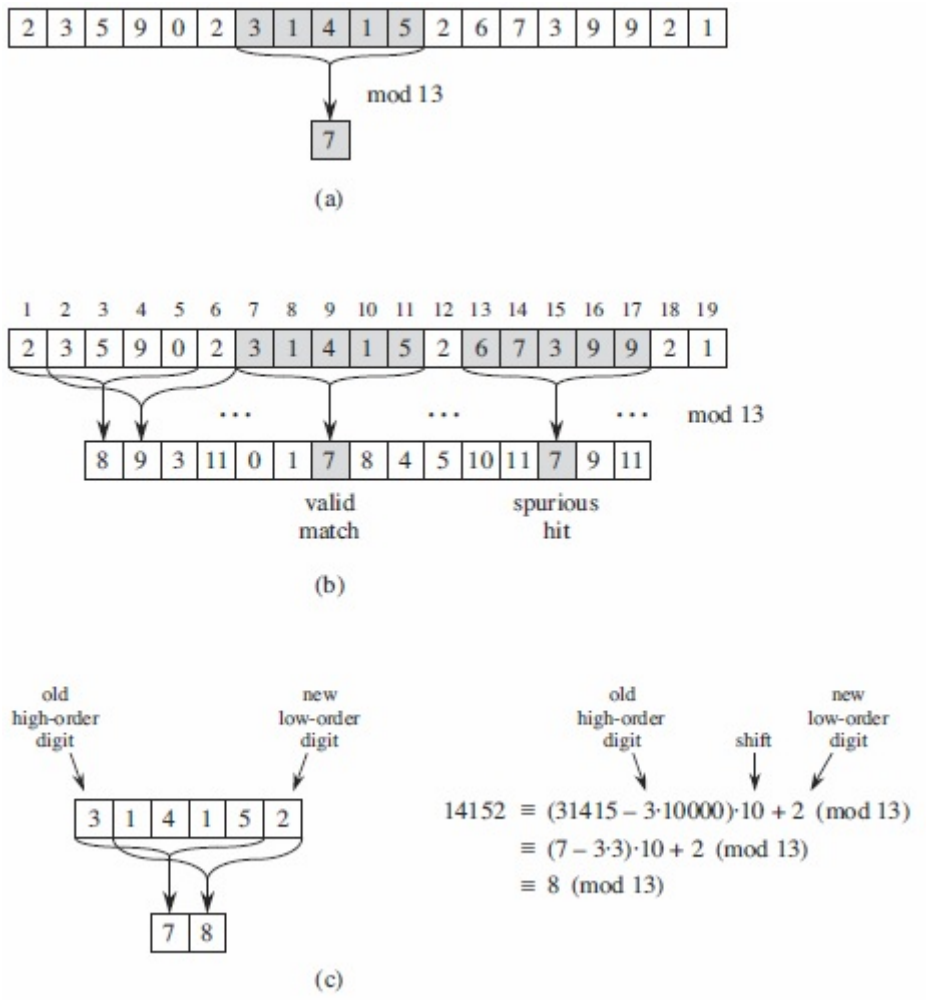

Figura 3. Exemplo de execução do algoritmo de *Rabin-Karp* retirado de [CORMEN et al. 2009]

A Figura 3 ilustra a execução de um passo desse algoritmo. Este algoritmo, somente na etapa de "*matching*", possui complexidade de tempo $O((n-m+1)*m)$ no pior caso – assim como a abordagem ingênua –, já a complexidade do seu preprocessamento é $\Theta(m)$, de acordo com [CORMEN et al. 2009].

4. Algoritmo de busca de subcadeias baseda em autômatos finitos

Basicamente, este algoritmo exige a contrução de uma máquina de estados, isto é, uma função de transição sobre P dado seu alfabeto Σ e a simples iteração sobre os caracteres

de T verificando se foi alcançado o estado de aceitação. Segue o algoritmo abaixo baseado no mesmo descrito em [CORMEN et al. 2009].

```
finite_automata_matcher(T, transition_function, acceptance_state)
    n := T.length
    q := 0

    for i = 0 to n
        q := transition_function(q, T[i])

        if q = acceptance_state
            print("Padrão ocorre com deslocamento" i - m)

compute_transition_function(P, alphabet)
    m := P.length

    for q = 0 to m
        for each character in alphabet
            k := min(m + 1, q + 2)

            repeat
                k := k - 1
            until P[k] is suffix of P[q]+character

            transition_function(q, character) := k

    return transition_function
```

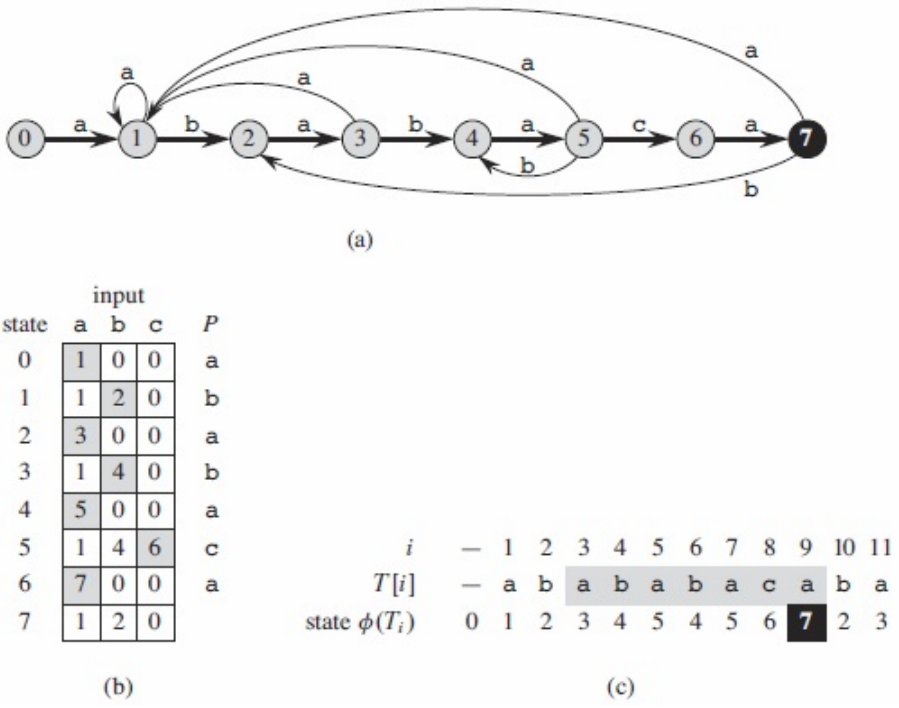

Figura 4. Exemplo de execução do algoritmo de busca de subcadeias baseda em autômatos finitos retirado de [CORMEN et al. 2009]

A Figura 4 ilustra a execução desse algoritmo. Inclusive, este algoritmo em especial pode ter como padrão uma expressão regular, já que por sua estrutura ser um autômato finito, é viável a busca por expressões regulares. Este algoritmo, somente na etapa de "*matching*", possui complexidade de tempo $\Theta(n)$, já a complexidade do seu preprocessamento é $O(m * |\Sigma|)$, de acordo com [CORMEN et al. 2009].

5. Algoritmo de *Knuth-Morris-Pratt*

Este é o primeiro algoritmo de *string matching* de complexidade de tempo linear. Além disso, diferente do algoritmo de busca de subcadeias baseda em autômatos finitos, esse algoritmo dispensa uma função de transição de estados: um *array* de prefixos acaba assumindo sua funcionalidade.

```
kmp_matcher(T, P)
    n := T.length
    m := P.length
    prefixes = compute_prefix_function(P)
    q = 0

    for i = 1 to n
        while q > 0 and P[q] != T[i - 1]
            q := prefixes[q - 1]

        if P[q] = T[i - 1]
            q := q + 1

        if q = m
            print("Padrão ocorre com deslocamento" i - m)
            q := prefixes[q - 1]

compute_prefix_function(P)
    m := P.length
    prefixes := a new array of length m
    prefixes[1] := 0
    i = 0

    for q = 2 to m
        while i > 0 and P[i + 1] != P[q]
            i = prefixes[i]

        if P[i + 1] = P[q]
            i := i + 1

        prefixes[q] = i

    return prefixes
```

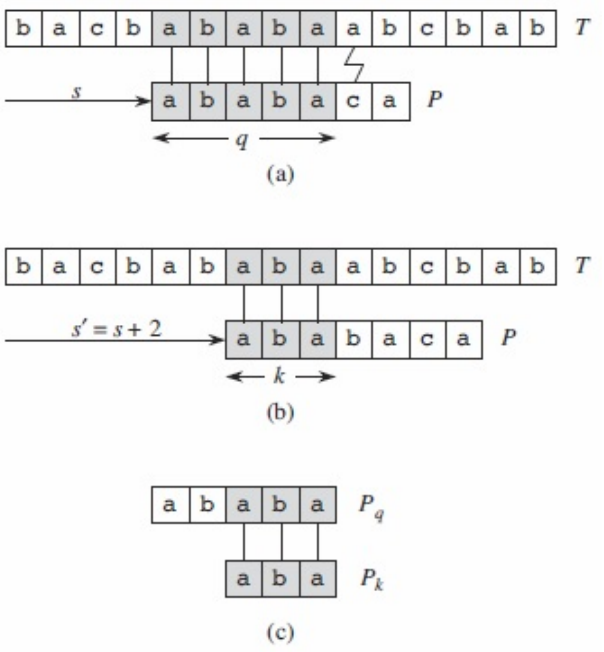

Figura 5. Exemplo de execução do algoritmo de *Knuth-Morris-Pratt* retirado de [CORMEN et al. 2009]

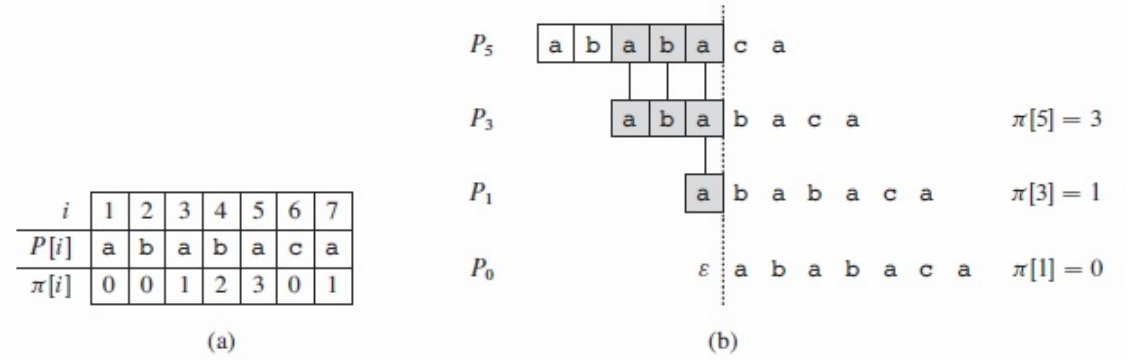

Figura 6. Exemplo de execução do algoritmo de *Knuth-Morris-Pratt* retirado de [CORMEN et al. 2009]

As Figuras 5 e 6 ilustram a execução desse algoritmo. Este algoritmo, somente na etapa de "*matching*", possui complexidade de tempo $\Theta(n)$, já a complexidade do seu preprocessamento é $\Theta(m)$, de acordo com [CORMEN et al. 2009]

6. Conclusão

Algoritmos de *string matching* possuem grande importância na era da informação em que estamos vivênciando. Podem ser dados exemplos como os motores de busca de páginas *web* que usam esses algoritmos para selecionar páginas pela internet com conteúdo relevante e, não se restringindo somente à essa área, na biologia estes algoritmos também são usados para a busca de padrões específicos em sequências de DNA, de acordo com [CORMEN et al. 2009].

Referências

CORMEN, T. H., LEISERSON, C., RIVEST, R., and STEIN, C. (2009). *Introduction to Algorithms*. The MIT Press, $3^{rd}edition$.