DCC207 – Algoritmos 2

Aula 02 – Algoritmos para manipulação de sequências (KMP, Boyer-Moore-Horspool, Shift-And)

Professor Renato Vimieiro

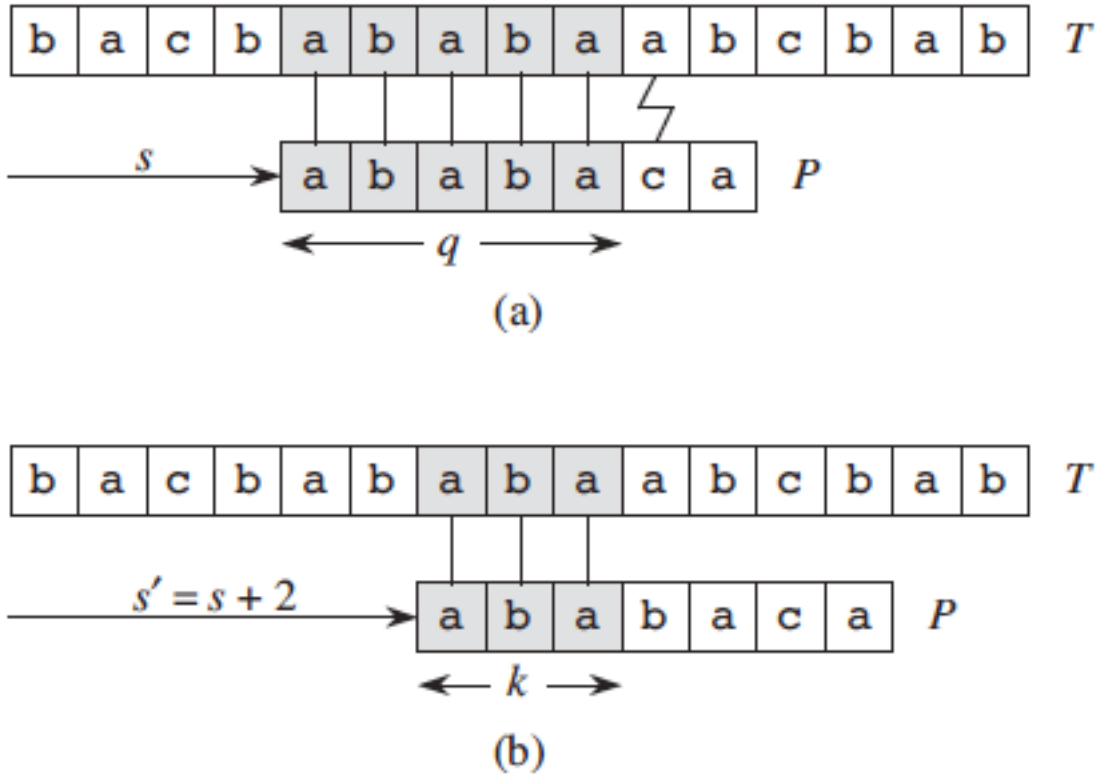
DCC/ICEx/UFMG

Introdução

- Vimos que o principal problema relacionado ao uso de AFDs para casamento de padrões é o alto custo para construção da função de transição
- Em 1977, Knuth, Morris e Pratt observaram que a construção do autômato ou sua função de transição não era de fato necessária
- Eles notaram que, na verdade, a função de transição era usada somente para calcular os deslocamentos do padrão sobre o texto, evitando comparações desnecessárias
- O algoritmo KMP utiliza uma função auxiliar que informa o deslocamento necessário para seguir a computação, caso uma falha de casamento ocorra

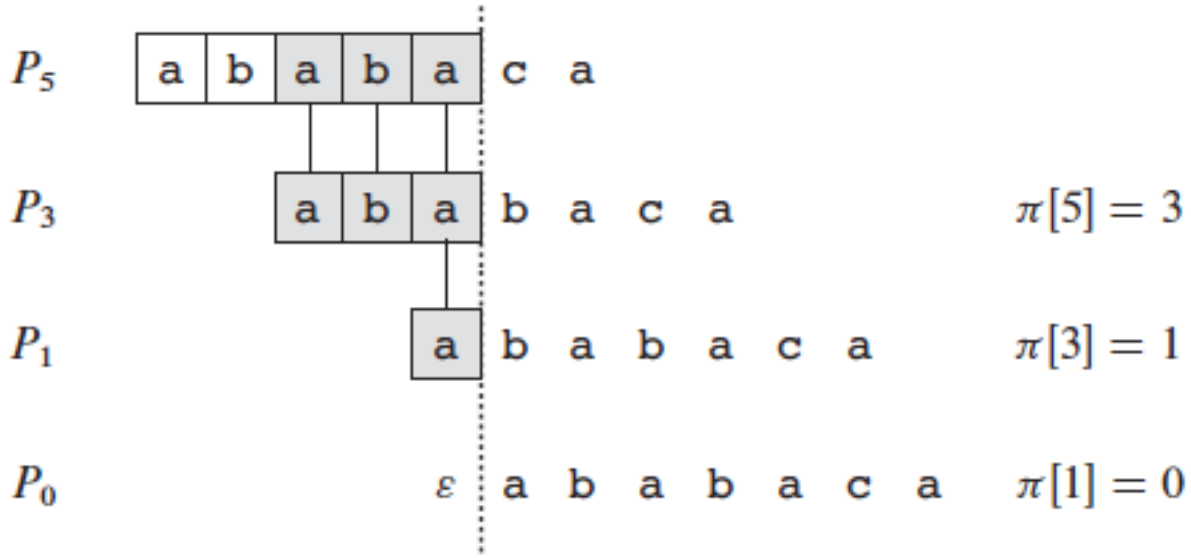
A função de prefixo

- Considere a situação ao lado em que queremos casar o padrão P com o texto T
- O casamento na parte (a) falha, pois os caracteres na posição s+6 são distintos
- O problema em questão é definir o deslocamento necessário para continuar o casamento
  - Deslocar 1 posição como no algoritmo força-bruta é desperdício de esforço
  - O deslocamento de s+5-k é mais adequado por observarmos um sub-prefixo de tamanho k no que foi casado até agora



A função prefixo

- Devemos avaliar todos os possíveis deslocamentos e escolher o mais adequado para a situação
- Podemos deslocar o padrão sobre o casamento parcial obtido até o momento para descobrir sufixos que possam ser aproveitados



A função prefixo

- Nossa intenção é aproveitar ao máximo os casamentos parciais, evitando assim retrabalho
- Dessa forma, a função auxiliar deve informar o tamanho do maior prefixo que podemos aproveitar do casamento parcial obtido
- Essa função é chamada de função prefixo
- Formalmente, a função prefixo é definida por
  - $\pi(q) = \max \{k \mid k < q, P_k \sqsupset P_q\}$

A função prefixo

- Para deduzirmos um algoritmo para computar a função prefixo, vamos introduzir a seguinte notação:
  - $\pi^1(q) = \pi(q)$
  - $\pi^k(q) = \pi(\pi^{k-1}(q))$
- A composição das funções retorna o tamanho do maior prefixo de um prefixo de um casamento parcial
- Agora, suponha que saibamos computar $\pi(1), \pi(2), ..., \pi(q)$
- Para computar $\pi(q+1)$, precisamos considerar alguns casos:
  - Se $P[\pi(q)+1] = P[q+1]$, então $\pi(q+1) = \pi(q)+1$ (por def. $P_{\pi(q)} \sqsupset P_q$)
  - Caso contrário, $\pi(q+1) = \min_k P[\pi^k(q)+1] = P[q+1]$, ou **zero** se não existir tal k
- Intuitivamente, caso não seja possível estender o prefixo atual, tentamos sucessivamente um prefixo menor, ou concluímos que é necessário recomeçar as comparações do início

A função prefixo

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a

A função prefixo

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

O algoritmo KMP

- O casamento de padrões pelo KMP segue a mesma lógica da computação da função prefixo
- Lemos o texto da esquerda para a direita
- Toda vez que ocorrer um casamento entre P[q] e T[i], incrementamos q e i
- Caso não haja um casamento, usamos a função prefixo para buscar uma nova posição de onde reiniciar as comparações

O algoritmo KMP

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

O algoritmo KMP

- Exemplo:
  - T = abababacaba
  - P = ababaca

Análise do algoritmo KMP

- A computação da função prefixo tem custo O(m)
  - K pode ser incrementado no máximo m-1 vezes (linha 9)
  - O laço das linhas 6 e 7 só decrementam o valor de k
- O custo do algoritmo principal é O(n+m)
  - O laço externo contribui com O(n)
  - O valor de q só pode ser incrementado na linha 9
  - O laço das linhas 6 e 7 só decrementam o valor de q (O(m))
- O algoritmo se mostra mais interessante para alfabetos grandes
  - Reduz tempo de pré-processamento e custo de armazenamento
- O algoritmo é adequado para aplicações em streams de dados, já que não há necessidade de retroceder no texto

Algoritmo de Horspool

- O algoritmo de Horspool foi proposto em 1980 como uma simplificação do algoritmo de Boyer-Moore
- Esse algoritmo supõe que retrocessos no texto não são um problema
- A ideia do algoritmo é tentar saltar o maior número de caracteres possível, diminuindo, assim, o número de comparações
- O casamento do padrão com o sufixo corrente do texto é feito da direita para a esquerda
  - T = JIM_SAW_ME_IN_A_BARBERSHOP
  - P = BARBER

Algoritmo de Horspool

- Ao casar um padrão com um sufixo, teremos 4 situações possíveis em relação ao último caractere c do sufixo
- Caso 1: c não acontece no padrão
  - Deslocar m posições à direita
- Caso 2: c não casa com o último caractere do padrão, mas ele acontece entre os outros m-1 primeiros caracteres do padrão
  - Deslocar rfind(c,P) posições à direita
- Caso 3: o erro acontece em uma posição k < m-1, mas o caractere c não ocorre em outra posição do padrão
  - Equivalente ao caso 1
- Caso 4: o erro acontece em uma posição k < m-1, e o caractere c ocorre em outra posição do padrão
  - Equivalente ao caso 2

Algoritmo de Horspool

- Os deslocamentos podem ser computados online, percorrendo-se o texto em busca dos caracteres
- Alternativamente, podemos construir uma tabela de deslocamentos a partir do padrão e do alfabeto usado
- A função de deslocamento deve retornar:
  - shift(c) = m, se c não ocorrer (mais) em P
  - shift(c) = m-1-rfind(c,P), caso contrário
- Ela pode ser computada a priori, usando um vetor de tamanho $|\Sigma|$
  - O vetor é inicializado com m em todas as posições
  - Depois, varremos o padrão atualizando a distância de cada caractere ao último

Algoritmo de Horspool

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Algoritmo de Horspool

- Tendo construído a tabela de deslocamentos, a ideia do algoritmo é bastante simples
  - Inicie a busca do começo do texto (mais à esquerda)
  - Case os caracteres do padrão com o texto da direita para a esquerda
  - Se um casamento for encontrado, retorne a posição i-m
  - Caso contrário, desloque i à direita conforme a tabela de deslocamento,
    - Se i ultrapassar o último caractere do texto, pare.
    - Senão repita os passos anteriores

Algoritmo de Horspool

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//          or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else  i ← i + Table[T[i]]
return −1
```

Algoritmo de Horspool

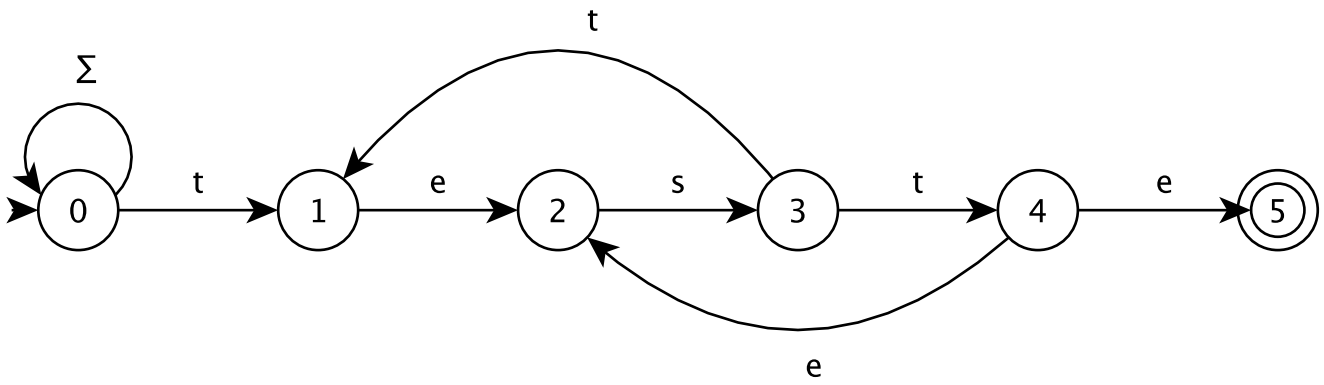
- Exemplo:
  - T = JIM_SAW_ME_IN_A_BARBERSHOP
  - P = BARBER
- É fácil perceber que o algoritmo possui complexidade O(nm) no pior caso
  - Qual é esse caso?
- No caso médio, o algoritmo possui complexidade O(n)
- Otimizações na implementação podem deixá-lo ainda mais rápido na prática

Algoritmo Shift-And

- O algoritmo Shift-And foi proposto Baeza Yates e Gonnet em 1989
- De forma similar ao KMP, ele também utiliza um autômato para armazenar os prefixos do padrão que são sufixos do texto lido até o momento
- A diferença é que ele utiliza um autômato finito não-determinístico para realizar os casamentos
  - O AFN pode ter múltiplas transições a partir de um dado estado e caractere
- Além disso, o algoritmo utiliza um vetor de bits como representação dos estados
  - A representação é mais compacta
  - Operações de transição podem ser feitas em tempo O(1) (operações bitwise)

Algoritmo Shift-And

- Exemplo: P=teste
- Cada estado um prefixo de P que é sufixo do texto lido



Algoritmo Shift-And

- O algoritmo armazena o conjunto de estados em um vetor de bits R
- Um bit R[j] é 1 sse $P_j \sqsupset T_i$
- O padrão é encontrado no texto sse R[m-1] = 1
- Como R representa o AFN, R[j+1] estará ativo somente se R[j] estiver ativo e P[j+1] == T[i+1], para um j arbitrário
  - R’ = ((R << 1)+1) & ((P[j+1] == T[i+1]) << j+1)
  - Por que (R << 1)+1?
- Podemos eliminar o deslocamento e comparação do segundo termo, representando o padrão por uma matriz de ocorrência de caracteres
  - M[c] = (P[j] == c) para $0 \leq j < m$, $c \in \Sigma$ (M[c] marca todas as ocorrências de c em P)

Algoritmo Shift-And

Algorithm 1: Shift-And Exato

foreach $c \in \Sigma$ **do** $M[c] = 0$;
for $j = 0$ **to** $m - 1$ **do** $M[P[j]] \mid= 1 << j$;
for $i = 0$ **to** $n - 1$ **do**
  $R = ((R << 1) + 1)\ \&\ M[T[i]]$;
  if $(R\ \&\ (1 << m - 1)) > 0$ **then** imprimir $i - m + 1$;
end

Algoritmo Shift-And

- Exemplo:
  - T = A_BARBERSHOP
  - P = BARBER
- O custo do algoritmo é O(n) desde que as operações bitwise tenham custo O(1) (m é aproximadamente do tamanho da palavra do processador)
- A complexidade de espaço é O(|Σ|+m)
  - Logo é negligenciável para alfabetos e padrões pequenos; como é o caso palavras em linguagem natural
- Pode ser facilmente estendido para computar padrões aproximados
  - Unix agrep: Wu e Manber 1991

Leitura

- Cormen seção 32.4
- Levitin seção 7.2
- Ziviani páginas 329-332

DCC207 – Algoritmos 2

Aula 02 – Algoritmos para manipulação de sequências (KMP, Boyer-Moore-Horspool, Shift-And)

Professor Renato Vimieiro

DCC/ICEx/UFMG

Análise do algoritmo KMP

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

- A computação da função prefixo tem custo O(m)
  - K pode ser incrementado no máximo m-1 vezes (linha 9)
  - O laço das linhas 6 e 7 só decrementam o valor de k
- O custo do algoritmo principal é O(n+m)
  - O laço externo contribui com O(n)
  - O valor de q só pode ser incrementado na linha 9
  - O laço das linhas 6 e 7 só decrementam o valor de q (O(m))
- O algoritmo se mostra mais interessante para alfabetos grandes
  - Reduz tempo de pré-processamento e custo de armazenamento
- O algoritmo é adequado para aplicações em streams de dados, já que não há necessidade de retroceder no texto

Análise do algoritmo KMP

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0
 5  for i = 1 to n
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]
 8      if P[q + 1] == T[i]
 9          q = q + 1
10      if q == m
11          print "Pattern occurs with shift"
12          q = π[q]
```

- A computação da função prefixo tem custo O(m)
  - K pode ser incrementado no máximo m-1 vezes (linha 9)
  - O laço das linhas 6 e 7 só decrementam o valor de k
- O custo do algoritmo principal é O(n+m)
  - O laço externo contribui com O(n)
  - O valor de q só pode ser incrementado na linha 9
  - O laço das linhas 6 e 7 só decrementam o valor de q (O(m))
- O algoritmo se mostra mais interessante para alfabetos grandes
  - Reduz tempo de pré-processamento e custo de armazenamento
- O algoritmo é adequado para aplicações em streams de dados, já que não há necessidade de retroceder no texto