CH08-320201

Algorithms and Data Structures

ADS

Lecture 27

Dr. Kinga Lipskoch

Spring 2019

String Matching

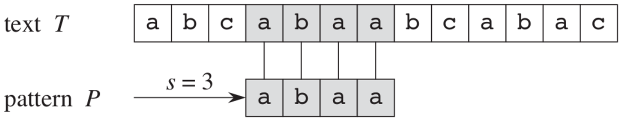
- Text-editing programs often need to find all occurrences of a pattern in the text.
- Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences.
- Internet search engines also use them to find web pages relevant to queries.

String Matching Problem (1)

- We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$.
- We further assume that the elements of P and T are characters drawn from a finite alphabet Σ.
- For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, ..., z\}$.
- The character arrays P and T are often called strings of characters.

String Matching Problem (2)

- The pattern P occurs with shift s in text T (or, P occurs beginning at position $s+1$ in text T)
- If P occurs with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift.
- The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.



Notation and Terminology

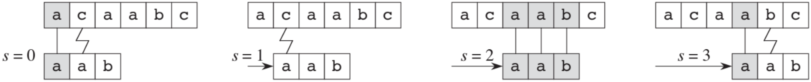
- Σ^* the set of all finite-length strings formed using characters from the alphabet Σ
- ϵ is the zero-length empty string
- $|x|$ is the length of a string x
- xy is the concatenation of two strings x and y
- $w \sqsubset x$ means w is a prefix of a string x, i.e., $x = wy$
- $w \sqsupset x$ means w is a suffix of a string x, i.e., $x = yw$

Naive String-Matching Algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n - m + 1$ possible values of s.

NAIVE-STRING-MATCHER(T, P)

```
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```



(a) (b) (c) (d)

Naive String-Matching Time Complexity

- For example, consider the text string a^n (a string of n a's) and the pattern a^m.
- For each of the $n - m + 1$ possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift.
- The worst-case running time is thus $\Theta((n - m + 1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$.
- The naive string-matcher is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s.

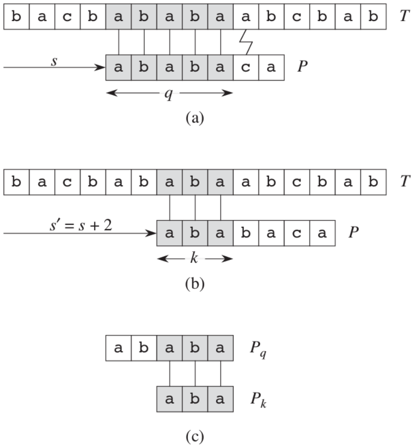
String Matching with Finite Automata

- Many string-matching algorithms build a finite automaton that scans the text string T for all occurrences of the pattern P.
- The matching time used after preprocessing the pattern to build the automaton is $\Theta(n)$.
- The time to build the automaton, however, can be large if Σ is large.
- The Knuth-Morris-Pratt algorithm has a clever way around this problem.

Knuth-Morris-Pratt Algorithm

- It is a linear-time string-matching algorithm due to Knuth, Morris, and Pratt.
- Its matching time is $\Theta(n)$ using just an auxiliary function π, which we precompute from the pattern in time $\Theta(m)$ and store in an array $\pi[1..m]$.

Prefix Function for a Pattern (1)



Prefix Function for a Pattern (2)

- Subfigure (a) shows a particular shift s of a template containing the pattern $P = ababaca$ against a text T.
- For this example, $q = 5$ of the characters have matched successfully, but the 6$^{\text{th}}$ pattern character fails to match the corresponding text character.
- The information that q characters have matched successfully determines the corresponding text characters.
- Knowing these q text characters allows us to determine immediately that certain shifts are invalid.
- The shift $s' = s + 2$ shown in subfigure (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match.

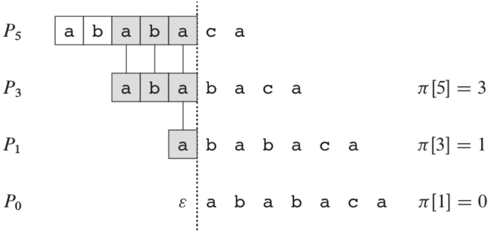
Prefix Function

Given a pattern $P[1..m]$, the prefix function for the pattern P is the function $\pi : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$.

Example: $P = ababaca$

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

(a)



(b)

KMP-Matcher

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

Compute Prefix

COMPUTE-PREFIX-FUNCTION(P)

```
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

KMP Example (1)

Position	0	1	2	3	4	5	6	7	8
Pattern:	a	b	a	b	c	a	b	a	b
π	0	0	1	2	0	1	2	3	4

Position:	*0*	*1*	*2*	*3*	*4*	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	*a*	*b*	*a*	*b*	**a**	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:	*a*	*b*	*a*	*b*	**c**	a	b	a	b												

KMP Example (2)

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:			a	b	a	b	c	a	b	a	b										

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	**b**	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:			a	b	a	b	c	**a**	b	a	b										

KMP Example (3)

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:								a	b	a	b	c	a	b	a	b					

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	**b**	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:								**a**	b	a	b	c	a	b	a	b					

KMP Example (4)

Position:	0	1	2	3	4	5	6	7	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	17	18	19	...
Text:	a	b	a	b	a	b	c	b	*a*	*b*	*a*	*b*	*c*	*a*	*b*	*a*	*b*	c	a	b	...
Pattern:									*a*	*b*	*a*	*b*	*c*	*a*	*b*	*a*	*b*				

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	*17*	*18*	*19*	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	*a*	*b*	*a*	*b*	*c*	*a*	*b*	...
Pattern:														*a*	*b*	*a*	*b*	*c*	*a*	*b*	...

Time Complexity

- First, line 4 starts k at 0, and the only way that k increases is by the increment operation in line 9, which executes at most once per iteration of the for loop of lines $5 - 10$.
- Thus, the total increase in k is at most $m - 1$.
- Second, since $k < q$ upon entering the for loop and each iteration of the loop increments q, we always have $k < q$.
- Therefore, the assignments in lines 3 and 10 ensure that $\pi[q] < q$ for all $q = 1, 2, ..., m$, which means that each iteration of the while loop decreases k.
- Third, k never becomes negative.
- Therefore, the total decrease in k is $m - 1$.
- COMPUTE-PREFIX-FUNCTION runs in time $\Theta(m)$.
- Similarly, KMP-MATCHER runs in $\Theta(n)$.

Final Exam

- Friday, the 31st of May, 2019 from 12:30 to 14:30 in SSC Hall 3 and 4
- Material covered after the midterm exam: Lecture 11 slide 17 until Lecture 27
- No cheat sheet allowed
- No phones or calculators allowed
- Grand tutorial: Wednesday, the 29th of May, 2019 in CSLH, Research I, 19:00 - 21:00