String Matching

Bin Wang

School of Software
Tsinghua University

December 6, 2019

Outline

1. **The String Matching problem**
   - Overview
   - Notation and terminology
2. **The String Matching Algorithms**
   - The naïve string-matching algorithm
   - The Rabin-Karp algorithm
   - String Matching with finite automata
   - The Knuth-Morris-Pratt algorithm
   - The Boyer-Moore algorithm

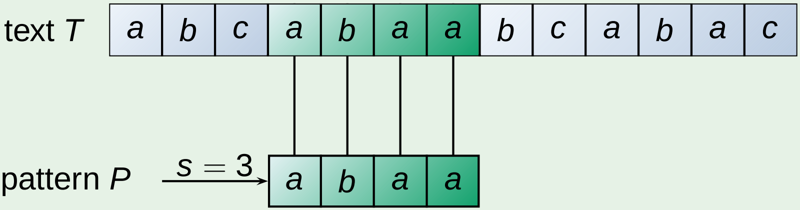
The string-matching problem

Definition

- The ***string-matching problem*** is to find all occurrences of the pattern $P[1..m]$ in the text $T[1..n]$.
- We further assume that the elements of P and T are characters drawn from a finite alphabet $\sum$. For example, we may have $\sum = \{0, 1\}$ or $\sum = \{a, b, \ldots, z\}$.
- We say that pattern P occurs with shift s in text T if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$.

The string-matching problem

Example



The string-matching algorithms

Algorithms comparison

Algorithm	Preprocessing	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\sum\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m+\|\sum\|)$	$\Omega(n/m), O(mn), O(n)$
Shift-Or	$\Theta(m+\|\sum\|)$	$\Theta(n)$
Reverse Factor	$O(m)$	$O(mn), O(n(\log m)/m)$

EXACT STRING MATCHING ALGORITHMS:
`http://www-igm.univ-mlv.fr/~lecroq/string/`

Notation and terminology

- $\sum$: a finite alphabet.
- $\sum^*$: the set of all finite-length strings formed using characters from $\sum$.
- concatenation of of two strings x and y : xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.
- prefix of a string, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \sum^*$.
- suffix of a string, denoted $w \sqsupset x$, if $x = yw$ for some string $y \in \sum^*$.

Notation and terminology

Example

$ab \sqsubset abcca$, $cca \sqsupset abcca$, $x \sqsupset y \iff xa \sqsupset ya$

Using the notation

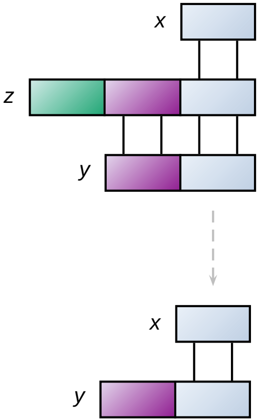
- We denote the k-character prefix $P[1..k]$ of the pattern $P[1..m]$ by P_k. Thus, $P_0 = \varepsilon$ and $P_m = P = P[1..m]$. Similarly, we denote the k-character prefix of the text T as T_k .
- Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$.

Overlapping-suffix lemma

Lemma 32.1 (Overlapping-suffix lemma)

Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

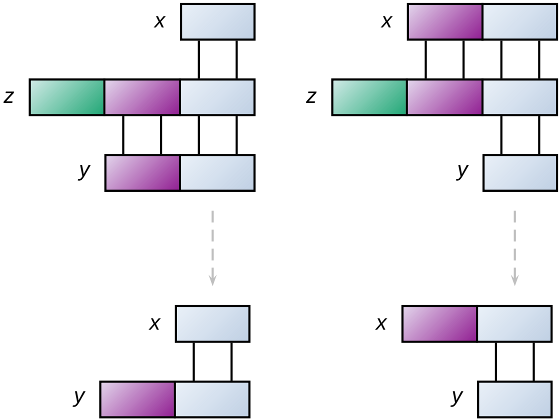
Overlapping-suffix lemma



(a) if $|x| \leq |y|$,

then $x \sqsupset y$.

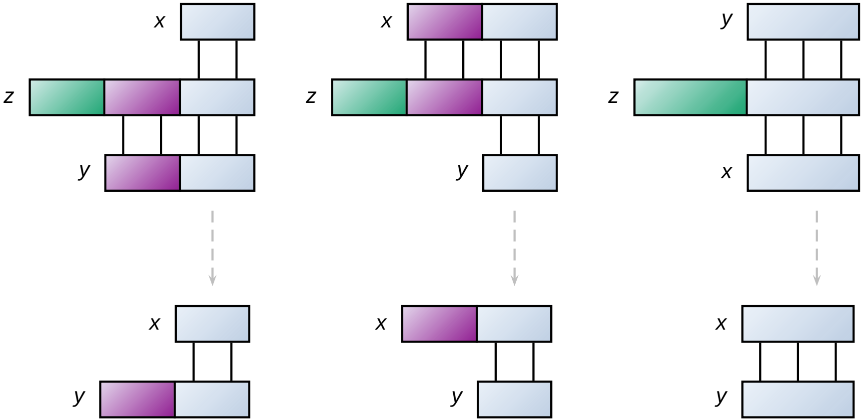
Overlapping-suffix lemma



(a) if $|x| \leq |y|$, then $x \sqsupset y$.

(b) if $|x| \geq |y|$, then $y \sqsupset x$.

Overlapping-suffix lemma



(a) if $|x| \leq |y|$, then $x \sqsupset y$.

(b) if $|x| \geq |y|$, then $y \sqsupset x$.

(c) if $|x| = |y|$, then $x = y$.

The naïve string-matching algorithm

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

Running time

$$\Theta((n - m + 1)m)$$

The Rabin-Karp algorithm–Basic idea

Intuition

- Let us assume that $\sum = \{0, 1, 2, \ldots, 9\}$.
- Given a pattern $P[1..m]$, let p denote its corresponding decimal value.
  Example: $31415 \Rightarrow p = 31,415$
- Given a pattern $P[1..m]$, we let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$. Thus

$$T[s+1..s+m] = P[1..m] \iff t_s = p$$

The Rabin-Karp algorithm–Basic idea

Running time

- We can compute p in time $\Theta(m)$ using Horner's rule.
  $p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$
- Compute all the t_s values in $\Theta(n-m+1)$.
  $t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$.

Example

$m = 5, t_s = 31415, T[s+5+1] = 2$
$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152$

The Rabin-Karp algorithm–Improvement

Use modulus

Compute p and t_s's modulo by choosing a suitable modulus q.

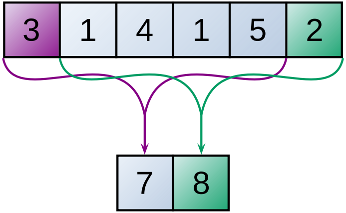
$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

The Rabin-Karp algorithm–Improvement

Use modulus

Compute p and t_s's modulo by choosing a suitable modulus q.

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

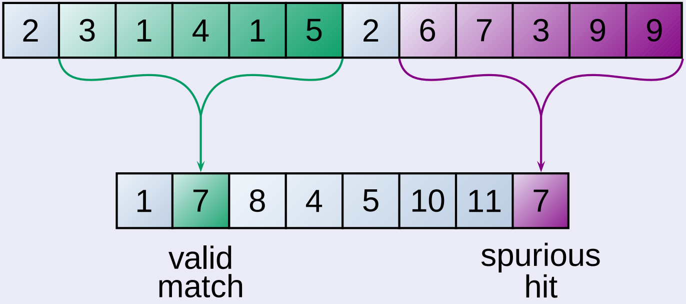


Example

$$14152 \equiv 10 \cdot (7 - 3 \cdot 3) + 2 \pmod{13}$$
$$\equiv 8 \pmod{13}$$

The Rabin-Karp algorithm–Improvement

Spurious hit



The Rabin-Karp algorithm–Improvement

Rabin-Karp-Matcher

RABIN-KARP-MATCHER(T, P, d, q)

$n = T.length$
$m = P.length$
$h = d^{m-1} \bmod q$
$p = 0$
$t_0 = 0$
for $i = 1$ **to** m // Preprocessing
    $p = (dp + P[i]) \bmod q$
    $t_0 = (dt_0 + T[i]) \bmod q$

The Rabin-Karp algorithm–Improvement

RABIN-KARP-MATCHER (Cont.)

for $s = 0$ **to** $n - m$ // Matching
    if $p == t_s$
        if $P[1..m] == T[s+1..s+m]$
            print "Pattern occurs with shift" s
    if $s < n - m$
        $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

The Rabin-Karp algorithm–Improvement

Running time

- If $P = a^m$ and $T = a^n$, then the verifications take time $\Theta((n-m+1)m)$, since each of the $n-m+1$ possible shifts is valid.
- In many applications, we expect few valid shifts and $O(n/q)$ spurious hits.
- The expected matching time: $O(n) + O(m(v + n/q))$, where v is the number of valid shifts.
- If $v = (O(1))$ and we choose $q \geq m$, the expected matching time is $O(n)$.

Finite automata

Definition

A ***finite automaton*** M is a 5-tuple $(Q, q_0, A, \sum, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start states***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- $\sum$ is a finite ***input alphabet***,
- δ is a function from $Q \times \sum$ into Q, called the ***transition function*** of M.

Finite automata

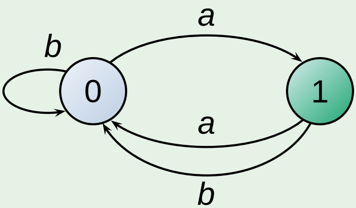
Definition

- If the auomaton is in state q and reads input character a, it moves from state q to state $\delta(q, a)$.
- If its current state q is a member of A, the machine M is said to have accepted the string read so far.
- A finite automaton M induces a function ϕ, called **final-sate function**.
  $\phi(\varepsilon) = q_0$
  $\phi(wa) = \delta(\phi(w), a)$, for $w \in \sum^*, a \in \sum$.

Finite automata

Example

state	input a	input b
0	1	0
1	0	0



$A = \{1\}$.

ϕ(abaaa) = 1 : accepted, ϕ(abbaa) = 0 : rejected.

String-matching automata

String-matching automaton to a given pattern $P[1..m]$

- $M = (Q, q_0, A, \sum, \delta)$,
- $Q = \{0, 1, \ldots, m\}$,
- $q_0 = 0, A = \{m\}$,

String-matching automata

String-matching automaton to a given pattern $P[1..m]$

- **suffix function** of $P : \sigma(x) = \max\{k : P_k \sqsupset x\}$.
  $P = ab, \sigma(\varepsilon) = 0, \sigma(ccaca) = 1$, and $\sigma(ccab) = 2$.
  For a pattern P of length m, we have $\sigma(x) = m \iff P \sqsupset x$,
- $\delta(q, a) = \sigma(P_q a)$.

String-matching automata

Example

$P = ababaca$:

$\delta(0, a) = \sigma(P_0 a) = \sigma(a) = 1,$
$\delta(0, b) = \sigma(P_0 b) = \sigma(b) = 0,$
$\delta(0, c) = \sigma(P_0 c) = \sigma(c) = 0,$

$\delta(1, a) = \sigma(P_1 a) = \sigma(aa) = 1,$
$\delta(1, b) = \sigma(P_1 b) = \sigma(ab) = 2,$
$\delta(1, c) = \sigma(P_1 c) = \sigma(ac) = 0,$

String-matching automata

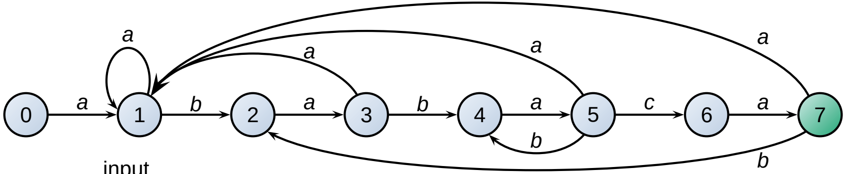
Example

$P = ababaca$:

$$\delta(2, a) = \sigma(P_2 a) = \sigma(aba) = 3,$$
$$\delta(2, b) = \sigma(P_2 b) = \sigma(abb) = 0,$$
$$\delta(2, c) = \sigma(P_2 c) = \sigma(abc) = 0,$$

$$\delta(3, a) = \sigma(P_3 a) = \sigma(abaa) = 1,$$
$$\delta(3, b) = \sigma(P_3 b) = \sigma(abab) = 4,$$
$$\delta(3, c) = \sigma(P_3 c) = \sigma(abac) = 0.$$

String-matching automata



state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

String-matching automata

Finite automaton matcher

FINITE-AUTOMATON-MATCHER(T, δ, m)

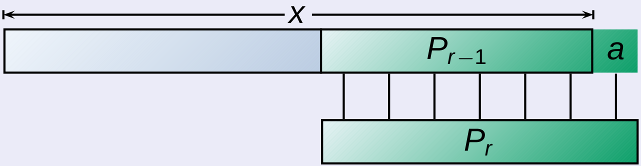
$n = T.length$
$q = 0$
for $i = 1$ **to** n
  $q = \delta(q, T[i])$
  if $q == m$
    print "Pattern occurs with shift" $i - m$

Correctness of FINITE-AUTOMATON-MATCHER

Lemma 32.2 (suffix-function inequality)

For any string x and character a, we have $\sigma(xa) \leq \sigma(x) + 1$.

Proof.



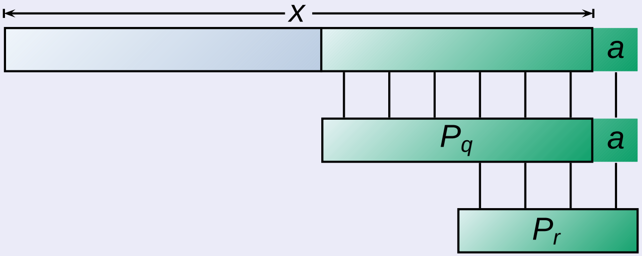
$\square$

Correctness of FINITE-AUTOMATON-MATCHER

Lemma 32.3 (suffix-function recursion lemma)

For any string x and character a, if $q = \sigma(x)$, then $\sigma(xa) = \sigma(P_q a)$

Proof.



□

Correctness of FINITE-AUTOMATON-MATCHER

Theorem 32.4

If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, the $\phi(T_i) = \sigma(T_i)$,for $i = 0, 1, \ldots, n$.

Correctness of FINITE-AUTOMATON-MATCHER

Proof.

$$
\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(by the definition of } T_{i+1}\text{)} \\
&= \delta(\phi(T_i), a) && \text{(by the definition of } \phi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P_q a) && \text{(by the definition of } \delta\text{)} \\
&= \sigma(T_i a) && \text{(by Lemma 32.3)} \\
&= \sigma(T_{i+1})
\end{aligned}
$$

$\square$

Computing the transition function

COMPUTE-TRANSITION-FUNCTION$(P, \sum)$

$m = P.length$
for $q = 0$ **to** m
  for each character $a \in \sum$
    $k = \min(m+1, q+2)$
    repeat
      $k = k - 1$
    until $P_k \sqsupset P_q a$
    $\delta(q, a) = k$
return δ

The prefix function

Basic Idea

The **prefix function** for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.

- Avoid testing useless shifts in the naïve pattern-matching algorithm.
- Avoid the pre-computation of δ for a string-matching automaton.

The prefix function

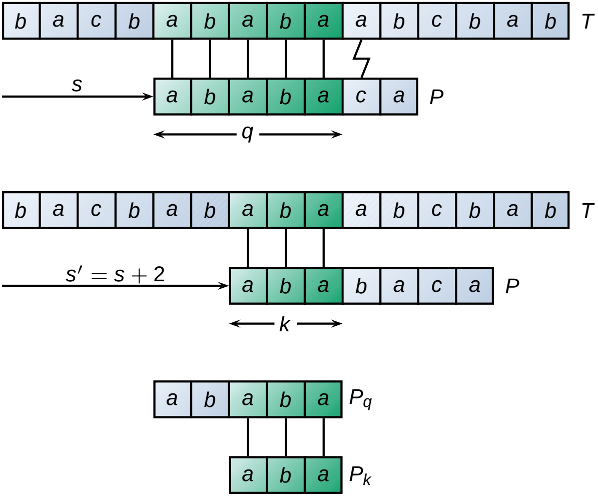
Basic Idea

- Given that pattern $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q$,

$$P[1..k] = T[s'+1..s'+k],$$

where $s'+k = s+q$?

The prefix function



The prefix function

Basic Idea

- Given a pattern $P[1..m]$, the **prefix function** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}.$$

The prefix function

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

P_8 $\boxed{a\,b\,a\,b\,a\,b\,a\,b}$ $c\ a$

P_6 $\boxed{a\,b\,a\,b\,a\,b}$ $a\ b\ c\ a$ $\pi[8] = 6$

P_4 $\boxed{a\,b\,a\,b}$ $a\ b\ a\ b\ c\ a$ $\pi[6] = 4$

P_2 $\boxed{a\,b}$ $a\ b\ a\ b\ a\ b\ c\ a$ $\pi[4] = 2$

P_0 ε $a\ b\ a\ b\ a\ b\ a\ b\ c\ a$ $\pi[2] = 0$

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.
```

KMP-Matcher

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-MATCHER

Running-time

Analysis of COMPUTE-PREFIX-FUNCTION:

- We use the aggregate method of amortized analysis and start by making some observations about k.
- The running time is $\Theta(m)$.

Analysis of KMP-MATCHER:

- We use a similar aggregate analysis by observing q.
- The running time is $\Theta(n)$.

Correctness of the prefix-function

Definition

Let $\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\}$, where $\pi^*[q]$ is defined in terms of functional iteration, so that $\pi^{(0)}[q] = q$ and $\pi^{(i+1)}[q] = \pi[\pi^{(i)}[q]]$ for $i \geq 1$. Example

Lemma 32.5 Prefix-function iteration lemma

Let P be a pattern of length m with prefix function π. Then for $q = 1, 2, \ldots m$, we have $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$.

Correctness of the prefix-function

Proof.

- We first prove that $i \in \pi^*[q]$ implies $P_i \sqsupset P_q$. Therefore $\pi^*[q] \subseteq \{k : k < q \text{ and } P_k \sqsupset P_q\}$.
- We prove that $\{k : k < q \text{ and } P_k \sqsupset P_q\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that the set $\{k : k < q \text{ and } P_k \sqsupset P_q\} - \pi^*[q]$ is nonempty, and let j be the largest such value. We must have $j < \pi[q]$.

Correctness of the prefix-function

Proof.

- We let j' denote the smallest integer in $\pi^*[q]$ that is greater than j. We have $P_j \sqsupset P_q$, and we have $P_{j'} \sqsupset P_q$. Thus $P_j \sqsupset P_{j'}$ by Lemma 32.1, and j is the largest value less than j' with this property.

  Therefore, we must have $\pi[j'] = j$, since $j' \in \pi^*[q]$, we mush have $j \in \pi^*[q]$ as well.

$\square$

Correctness of the prefix-function

Lemma 32.6

Let P be a pattern of length m, and let π be the prefix function for P. For $q = 1, 2, \ldots m$, if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q-1]$.

Proof.

If $r = \pi[q] > 0$, then $r < q$ and $P_r \sqsupset P_q$; thus, $r - 1 < q - 1$ and $P_{r-1} \sqsupset P_{q-1}$(by dropping the last character from P_r and P_q). By Lemma 32.5, therefore, $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. $\square$

Correctness of the prefix-function

Corollary 32.7

Let P be a pattern of length m, and let π be the prefix function for P. For $q = 2, 3, \ldots m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\} & \text{if } E_{q-1} \neq \emptyset \end{cases}$$

where E_{q-1} is the subset of $\pi^*[q-1]$ for $q = 2, 3, \ldots, m$.

Correctness of the prefix-function

Corollary 32.7(cont.)

$$
\begin{aligned}
E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\
&= \{k : k < q-1 \text{ and } P_k \sqsupset P_{q-1} \\
&\quad \text{and } P[k+1] = P[q]\} \\
&= \{k : k < q-1 \text{ and } P_{k+1} \sqsupset P_q\}
\end{aligned}
$$

Correctness of the prefix-function

Proof.

If E_{q-1} is empty, there is no $k \in \pi^*[q-1]$. Therefore $\pi[q] = 0$.

If E_{q-1} is nonempty, then for each $k \in \pi^*[q-1]$ we have $k+1 < q$ and $P_{k+1} \sqsupset P_q$. Therefore, from the definition of $\pi[q]$, we have

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\}.$$

Correctness of the prefix-function

Proof.

Note that $\pi[q] > 0$. Let $r = \pi[q] - 1$, so that $r + 1 = \pi[q]$. Since $r + 1 > 0$, we have $P[r + 1] = P[q]$. Furthermore, by Lemma 32.6, we have $r \in \pi^*[q - 1]$. Therefore, $r \in E_{q-1}$, and so

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\}.$$

□

Correctness of the prefix-function

Correctness of the prefix-function

- At the start of each iteration of the for loop of lines 5-10 in COMPUTE-PREFIX-FUNCTION, we have that $k = \pi[q-1]$.
- The loop on lines 6-7 searches through all values $k \in \pi^*[q-1]$ until one is found for which $P[k+1] = P[q]$; at that point, k is the largest value in the set E_{q-1}, so that, by Corollary 32.7, we can set $\pi[q]$ to $k+1$.

Correctness of the prefix-function

Correctness of the prefix-function

- If no such k is found, $k = 0$ in line 8. If $P[1] = P[q]$, then we should set both k and $\pi[q]$ to 1; otherwise we should leave k alone and set $\pi[q]$ to 0. Lines 8-10 set k and $\pi[q]$ correctly in either case.

Correctness of the KMP algorithm

Correctness of the KMP algorithm

- The procedure KMP-MATCHER can be viewed as a reimplementation of the procedure FINITE-AUTOMATON-MATCHER.
- Specifically, we shall prove that the code in lines 6-9 of KMP-MATCHER is equivalent to line 4 of FINITE-AUTOMATON-MATCHER , which sets q to $\delta(q, T[i]) = \sigma(T[i])$.

Correctness of the KMP algorithm

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

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

i	–	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	–	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Correctness of the KMP algorithm

Correctness of the KMP algorithm

- Instead of using a stored value of $\sigma(T[i])$, however, this value is recomputed as necessary from π.
- The proof proceeds by induction on the number of loop iterations. Initially, both procedures set q to 0 as they enter their respective for loops **for** the first time.

Correctness of the KMP algorithm

Correctness of the KMP algorithm

- Consider iteration i of the **for** loops in KMP-MATCHER, let q' be state at the start of this loop iteration. By the inductive hypothesis, we have $q' = \sigma(T_{i-1})$. We need to show that $q = \sigma(T_i)$ at line 10.
- When we consider the character $T[i]$, the longest prefix of P that is a suffix of T_i is either $P_{q'+1}$(if $P[q'+1] = T[i]$) or some prefix of $P_{q'}$.

Correctness of the KMP algorithm

Correctness of the KMP algorithm

- If $\sigma(T_i) = 0$, the $P_0 = \epsilon$ is the only prefix of P that is a suffix of T_i. Therefore, $q = 0$ at line 10, so that $q = \sigma(T_i)$.
- If $\sigma(T_i) = q' + 1$, the $P[q' + 1] = T[i]$, we have $q = q' + 1 = \sigma(T_i)$.
- If $0 < \sigma(T_i) \leq q'$, we have $q + 1 = \sigma(P_{q'} T[i]) = \sigma(T_{i-1} T[i]) = \sigma(T_i)$. when the **while** loop terminates. After line 9 increments q, we have $q = \sigma[T_i]$.

Correctness of the KMP algorithm

Correctness of the KMP algorithm

- Line 12 is necessary in KMP-MATCHER to avoid a possible reference to $P[m+1]$ on line 6 after an occurrence of P has been found.

Idea

Basic idea

More information is gained by matching the pattern from the **right** than from the left.

Observation 1

If current *char* is known not to occur in *pattern*, then we know we need not consider the possibility of an occurrence of *pattern* at *text* positions $1, 2, \ldots,$ or m: Such an occurrence would require that *char* be a character of *pattern*.

Idea

Basic idea

More information is gained by matching the pattern from the **right** than from the left.

Observation 1

If current *char* is known not to occur in *pattern*, then we know we need not consider the possibility of an occurrence of *pattern* at *text* positions $1, 2, \ldots,$ or m: Such an occurrence would require that *char* be a character of *pattern*.

Idea

Observation 2

More generally, if the **last(right-most)** occurrence of *char* in *pattern* is δ_1 characters from the right end of *pattern*, then we know we can slide *pattern* down δ_1 positions without checking for matches.

Idea

Observation 3

When a mismatch occurs at position δ_2 characters from the right end of *pattern*, we can slide to a position to match the *subpattern* $P_{m-\delta_2} \cdots P_m$.

Idea

Example

pattern : AT−THAT

text : ···WHICH−FINALLY−HALTS·−−AT−THAT−POINT···

↑

Since "F" is known not to occur in *pattern*, we can appeal to **Observation 1** and move the pointer down by 7:

Idea

Example

pattern : AT−THA$\boxed{\text{T}}$

text : ···WHICH−FINALLY$\boxed{-}$HALTS·−−AT−THAT−POINT···

$\uparrow$

Appealing to **Observation 2**, we can move the pointer down 4 to align the two hyphens:

Idea

Example

pattern : AT−THAT

text : ···WHICH−FINALLY−HALTS·−−AT−THAT−POINT···

Now *char* matches its opposite in *pattern*. Therefore we step left by one:

Idea

Example

pattern : AT−THAT

text : ···WHICH−FINALLY−HALTS·−−AT−THAT−POINT···

Appealing to **Observation 1**, we can move the *pattern* to the right by 6:

Idea

Example

pattern : AT−THAT

text : ···WHICH−FINALLY−HALTS·−−AT−THAT−POINT···

Again *char* matches the last character of *pattern*.
Stepping to the left twice produces:

Idea

Example

pattern : AT–THAT

text : · · · WHICH–FINALLY–HALTS · – –AT–THAT–POINT · · ·

Noting that we have a mismatch, we appeal to **Observation 3**. The best move is to align the discovered substring "AT" with the beginning of *pattern*.

Idea

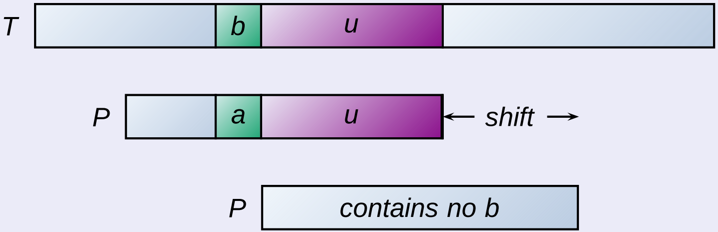
Example

pattern : AT−THAT

text : ···WHICH−FINALLY−HALTS·−−AT−THAT−POINT···

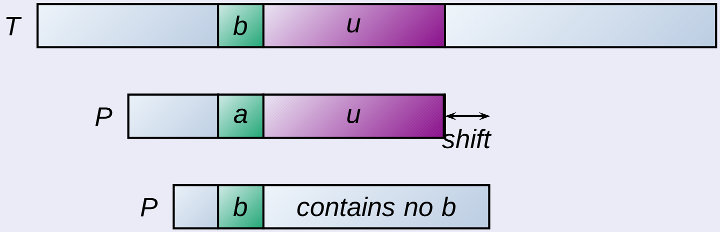
This time we discover the *pattern*. Note that we made only 14 reference to *text*.

Bad-Character Shift



The bad-character shift, *b* does not occur in *P*.

Bad-Character Shift



The bad-character shift, *b* occurs in *P*.

Bad-Character Shift

Definition

The bad-character shift is stored in a table $bmBc$ of size $|\Sigma|$. For $c \in \Sigma$:

$$bmBc[c] = \begin{cases} \min\{i : 1 \leq i < m - 1 \text{ and} \\ \qquad P[m-i] = c\}, & \text{if } c \text{ occurs in } P \\ m, & \text{otherwise.} \end{cases}$$

Bad-Character Shift

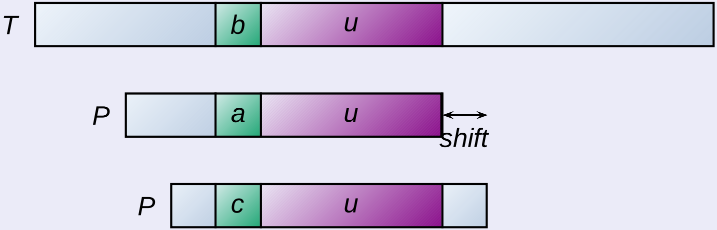
Definition

The bad-character shift is stored in a table $bmBc$ of size $|\Sigma|$. For $c \in \Sigma$:

$$bmBc[c] = \begin{cases} \min\{i : 1 \leq i < m - 1 \text{ and} \\ \qquad P[m-i] = c\}, & \text{if } c \text{ occurs in } P \\ m, & \text{otherwise.} \end{cases}$$

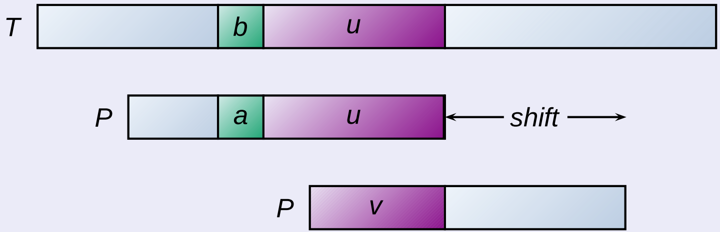
$$shift : s = bmBc[c] + j - m$$

Good-Suffix Shift



The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

Good-Suffix Shift



The good-suffix shift, only a suffix of u re-occurs in P.

Good-Suffix Shift

Definition

The good-suffix shift function is stored in a table *bmGs*.

$$Cs(i,s): \text{ for each } k \text{ such that } i < k \leq m, \\ P[k-s] = P[k] \text{ or } s \geq k$$

and

$$Co(i,s): \text{ if } s < i \text{ then } P[i-s] \neq P[i]$$

Good-Suffix Shift

Definition

Then, for $1 \leq i \leq m$

$$bmGs[i] = \min\{s > 0 : Cs(i, s) \text{ and } Co(i, s) \text{ hold }\}$$

Boyer-Moore Algorithm

BM-MATCHER(T, P)

```
n = T.length
m = P.length
COMPUTE-BMBC(P, bmBc)
COMPUTE-BMGS(P, bmGs)
s = 0
while s <= n - m
      i = m
      while P[i] == T[s + i]
            if i == 1
                  print "Pattern occurs with shift" s
            else i = i - 1
      s = s + MAX(bmGs[i], bmBc[T[s + i]] - m + i)
```

Boyer-Moore Algorithm

COMPUTE-BMBC($P, bmBc$)

$m = P.length$
for each character $a \in \sum$
    $bmBc[a] = m$
for $i = 1$ **to** $m - 1$
    $bmBc[P[i]] = m - i$

Boyer-Moore Algorithm

Example

Pattern: GCAGAGAG

$bmBc[A] = 1; bmBc[C] = 6;$
$bmBc[G] = 2; bmBc[T] = 8$

Pattern: ANPANMAN

$bmBc[A] = 1; bmBc[M] = 2;$
$bmBc[N] = 3; bmBc[P] = 5$

How to computer good-suffix shift?

Overlapping Suffix Function

$$Osuff[i] = \max\{k : P[i-k+1..i] = P[m-k+1..m]\}$$

Example

Table: Compute Overlapping Suffix

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8

How to computer good-suffix shift?

Overlapping Suffix Function

$$Osuff[i] = \max\{k : P[i-k+1..i] = P[m-k+1..m]\}$$

Example

Table: Compute Overlapping Suffix

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8

How to computer good-suffix shift?

Overlapping Suffix Function

$$Osuff[i] = \max\{k : P[i-k+1..i] = P[m-k+1..m]\}$$

Example

Table: Compute Overlapping Suffix

i	1	2	3	4	5	6	7	8
P[i]	A	N	P	A	N	M	A	N
Osuff[i]	0	2	0	0	2	0	0	8

How to computer good-suffix shift?

```
COMPUTE-BMGS(P, bmGs)
m = P.length
COMPUTE-OSUFF(P, Osuff)
for i = 1 to m
    bmGs[i] = m
j = 1
for i = m − 1 downto 1
    if Osuff[i] == i
        while j ≤ m − i
            if bmGs[j] == m
                bmGs[j] = m − i
            j = j + 1
for i = 1 to m − 1
    bmGs[m − Osuff[i]] = m − i
```

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	7	7	7	7	7	7	7	8
	7	7	7	2	7	4	7	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	7	7	7	7	7	7	7	8
	7	7	7	2	7	4	7	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	7	7	7	7	7	7	7	8
	7	7	7	2	7	4	7	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	G	C	A	G	A	G	A	G
Osuff[i]	1	0	0	2	0	4	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	7	7	7	7	7	7	7	8
	7	7	7	2	7	4	7	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	A	N	P	A	N	M	A	N
Osuff[i]	0	2	0	0	2	0	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	6	6	6	6	6	6	8	8
	6	6	6	6	6	3	8	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	A	N	P	A	N	M	A	N
Osuff[i]	0	2	0	0	2	0	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	6	6	6	6	6	6	8	8
	6	6	6	6	6	3	8	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	A	N	P	A	N	M	A	N
Osuff[i]	0	2	0	0	2	0	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	6	6	6	6	6	6	8	8
	6	6	6	6	6	3	8	1

How to computer good-suffix shift?

Example

Table: Compute bmGs

i	1	2	3	4	5	6	7	8
P[i]	A	N	P	A	N	M	A	N
Osuff[i]	0	2	0	0	2	0	0	8
	8	8	8	8	8	8	8	8
bmGs[i]	6	6	6	6	6	6	8	8
	6	6	6	6	6	3	8	1

BM Algorithm Example

Example

pattern : GCAGAGAG

text : GCATCGCAGAGAGTATACAGTACG

Shift by 1 ($bmGs[8] = bmBc[A] - 7 + 7 = 1$):

BM Algorithm Example

Example

pattern : GCAGAGAG

text : GCATCGCAGAGAGTATACAGTACG

↑

Shift by 4 ($bmGs[6] = bmBc[C] - 7 + 5 = 4$):

BM Algorithm Example

Example

pattern : GCAGAGAG

text : GCATCGCAGAGAGTATACAGTACG

↑

Shift by 7 ($bmGs[1] = 7$):

BM Algorithm Example

Example

pattern : GCAGAGAG

text : GCATCGCAGAGAGTATACAGTACG

Shift by 4 ($bmGs[6] = bmBc[C] - 7 + 5 = 4$):

BM Algorithm Example

Example

pattern : GCAGAGAG

text : GCATCGCAGAGAGTATACAGTACG

Shift by *bmGs*[7]. The Boyer-Moore algorithm performs 17 text character comparisons on the example.

BM Algorithm History

- It was shown at first that the BM algorithm makes at most $6n$ comparisons if the pattern does not occur in the text.
- Guibas and Odlyzko [1980] reduced this to $4n$ under the same assumption.
- Cole[1991] finally proved an essentially tight bound of $3n - \Omega(n/m)$ comparisons for the BM algorithm, whether or not the pattern(a non periodic pattern) occurs in the text.

BM Algorithm History

- The Turbo BM algorithm takes an additional constant amount of space to complete a search within $2n$ comparisons.
- Visit http://www-igm.univ-mlv.fr/~lecroq/string/ (with Visualization Demos, Descriptions and C codes for 35 different string matching algorithms)

Exercise

Exercise

Compute the bad-character shift and good-suffix shift of pattern “AT-THAT” and “AGAGTAGAG”