


 TARTU ÜLIKOOL
ARVUTITEADUSE INSTITUUT



Text Algorithms (6EAP)

Lecture 2: Exact pattern matching

Jaak Vilo
2012 fall

Jaak Vilo MTAT.03.190 Text Algorithms 1

Exact pattern matching

- $S = s_1 s_2 \dots s_n$ (text) $|S| = n$ (length)
- $P = p_1 p_2 \dots p_m$ (pattern) $|P| = m$
- Σ - alphabet $|\Sigma| = c$
- Does S contain P ?
 - Does $S = S' P S''$ for some strings S' ja S'' ?
 - Usually $m \ll n$ and n can be (very) large

Find occurrences in text



Algorithms

- Brute force
- Knuth-Morris-Pratt
- Karp-Rabin
- Shift-OR, Shift-AND
- Boyer-Moore
- Factor searches

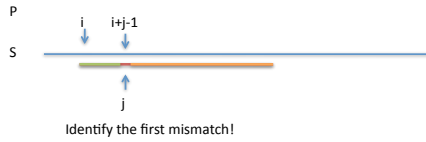
Animations

- <http://www-igm.univ-mlv.fr/~lecrog/string/>
- **EXACT STRING MATCHING ALGORITHMS Animation in Java**
- Christian Charras - Thierry Lecroq
Laboratoire d'Informatique de Rouen
Université de Rouen
Faculté des Sciences et des Techniques
76821 Mont-Saint-Aignan Cedex
FRANCE
- e-mails: {[Christian.Charras](mailto:Christian.Charras@laposte.net), [Thierry.Lecroq](mailto:Thierry.Lecroq@laposte.net)}
@laposte.net

Brute force: BAB in text?

A B A C A B A B B A B B B A
B A B

Brute Force



Question:

- Problems of this method? ☹
- Ideas to improve the search? ☹

Brute force

Algorithm Naive

Input: Text $S[1..n]$ and pattern $P[1..m]$

Output: All positions i , where P occurs in S

```
for( i=1 ; i <= n-m+1 ; i++)
  for ( j=1 ; j <= m ; j++)
    if ( S[i+j-1] != P[j] ) break ;
  if ( j > m ) print i ;
```

```
attempt 1:
goatogcagagagatatacagtaog
GCAG...

attempt 2:
goatogcagagagatatacagtaog
g.....

attempt 3:
goatogcagagagatatacagtaog
g.....

attempt 4:
goatogcagagagatatacagtaog
g.....

attempt 5:
goatogcagagagatatacagtaog
g.....

attempt 6:
goatogcagagagatatacagtaog
GCAGAGAG

attempt 7:
goatogcagagagatatacagtaog
g.....
```

NaiveSearch

- 1 function NaiveSearch(string s[1..n], string sub[1..m])
- 2 for i from 1 to n-m+1
- 3 for j from 1 to m
- 4 if $s[i+j-1] \neq \text{sub}[j]$
- 5 jump to next iteration of outer loop
- 6 return i
- 7 return not found

C code

```
int bf_2( char* pat, char* text , int n ) /* n = textlen */
{
  int m, i, j ;
  int count = 0 ;
  m = strlen(pat);

  for ( i=0 ; i + m <= n ; i++) {

    for( j=0; j < m && pat[j] == text[i+j] ; j++) ;

    if( j == m )
      count++ ;
  }

  return(count);
}
```

C code

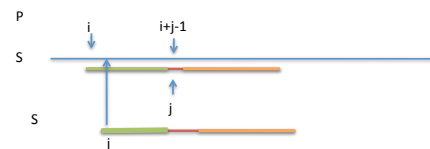
```
int bf_1( char* pat, char* text )
{
  int m ;
  int count = 0 ;
  char *tp;

  m = strlen(pat);
  tp=text ;

  for( ; *tp ; tp++ ) {
    if( strcmp( pat, tp, m ) == 0 ) {
      count++ ;
    }
  }

  return( count ) ;
}
```

Main problem of Naive



- For the next possible location of P , check again the same positions of S

Goals

- Make sure only a constant nr of comparisons/ operations is made for each position in S
 - Move (only) from left to right in S
 - How?
 - After a test of $S[i] \neq P[j]$ what do we now ?

D. Knuth, J. Morris, V. Pratt:
Fast Pattern Matching in strings.
SIAM Journal on Computing 6:323-350, 1977.

Knuth-Morris-Pratt

- Make sure that no comparisons “wasted”



- After such a mismatch we already know exactly the values of green area in S !

Knuth-Morris-Pratt

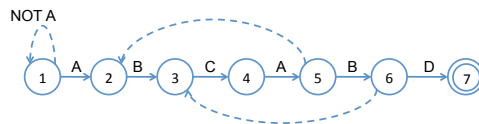
D. Knuth, J. Morris, V. Pratt:
Fast Pattern Matching in strings.
SIAM Journal on Computing 6:323-350, 1977.

- Make sure that no comparisons “wasted”

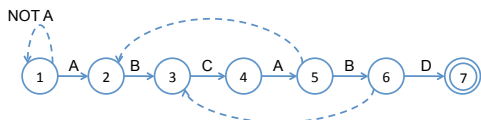


- P – longest suffix of any prefix that is also a prefix of a pattern
- Example: ABCABD
 ABCABD

Automaton for ABCABD



Automaton for ABCABD



Fail:

0	1	1	1	2	3	1
---	---	---	---	---	---	---

Pattern:

A	B	C	A	B	D
1	2	3	4	5	6

KMP matching

Input: Text $S[1..n]$ and pattern $P[1..m]$

Output: First occurrence of P in S (if exists)

```

i=1; j=1;
initfail(P) // Prepare fail links
repeat
  if j==0 or S[i] == P[j]
    then i++, j++ // advance in text and in pattern
  else j = fail[j] // use fail link
until j>m or i>n
if j>m then report match at i-m
    
```

Initialization of fail links

Algorithm: KMP_Initfail

Input: Pattern P[1..m]

Output: fail[] for pattern P

```

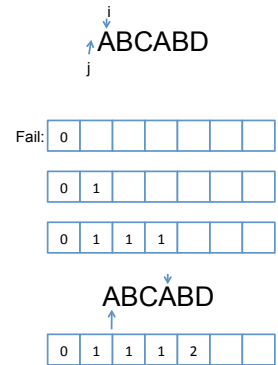
i=1, j=0 , fail[1]= 0
repeat
  if j==0 or P[i] == P[j]
    then i++ , j++ , fail[i] = j
    else j = fail[j]
until i>=m
    
```

Initialization of fail links

i=1, j=0 , fail[1]= 0

```

repeat
  if j==0 or P[i] == P[j]
    then i++ , j++ , fail[i] = j
    else j = fail[j]
until i>=m
    
```



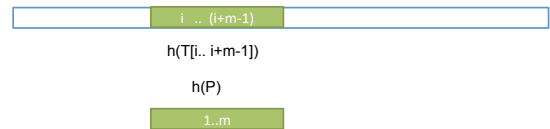
Analysis of time complexity

- At every cycle either i and j increase by 1
- Or j decreases (j=fail[j])
- i can increase n (or m) times
- Q: How often can j decrease?
 - A: not more than nr of increases of i
- **Amortised analysis: O(n)** , preprocess O(m)

Karp-Rabin

R.Karp and M. Rabin: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31 (1987), 249-260.

- Compare in O(1) a hash of P and S[i..i+m-1]

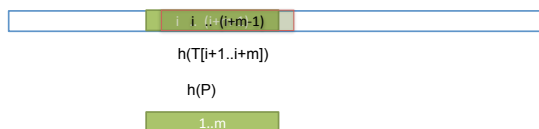


- Goal: O(n).
- $f(h(T[i.. i+m-1]) \rightarrow h(T[i+1..i+m])) = O(1)$

Karp-Rabin

R.Karp and M. Rabin: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31 (1987), 249-260.

- Compare in O(1) a hash of P and S[i..i+m-1]



- Goal: O(n).
- $f(h(T[i.. i+m-1]) \rightarrow h(T[i+1..i+m])) = O(1)$

Hash

- “Remove” the effect of T[i] and “Introduce” the effect of T[i+m] – in O(1)
- Use base |Σ| arithmetics and treat charcters as numbers
- In case of hash match – check all m positions
- Hash collisions => Worst case O(nm)

Let's use numbers

- $T = 57125677$
- $P = 125$ (and for simplicity, $h = 125$)
- $H(T[1]) = 571$
- $H(T[2]) = (571 - 5 * 100) * 10 + 2 = 712$
- $H(T[3]) = (H(T[2]) - \text{ord}(T[1]) * 10m) * 10 + T[3+m-1]$

hash

- c – size of alphabet
- $H_{Si} = H(S[i..i+m-1])$
- $H(S[i+1 .. i+m]) = (H_{Si} - \text{ord}(S[i]) * c^{m-1}) * c + \text{ord}(S[i+m])$
- Modulo arithmetic – to fit value in a word!

- $\text{hash}(w[0 .. m-1]) = (w[0] * 2^{m-1} + w[1] * 2^{m-2} + \dots + w[m-1] * 2^0) \bmod q$

Karp-Rabin

```

Input: Text S[1..n] and pattern P[1..m]
Output: Occurrences of P in S
1. c=20; /* Size of the alphabet, say nr. of aminoacids */
2. q = 33554393 /* q is a prime */
3. cm = cm-1 mod q
4. hp = 0 ; hs = 0
5. for i = 1 .. m do hp = ( hp*c + ord(p[i]) ) mod q // H(P)
6. for i = 1 .. m do hs = ( hp*c + ord(s[i]) ) mod q // H(S[1..m])
7. if hp == hs and P == S[1..m] report match at position

8. for i=2 .. n-m+1
9.   hs = ( (hs - ord(s[i-1])*cm) * c + ord(s[i+m-1]) ) mod q
10.  if hp == hs and P == S[i..i+m-1]
11.    report match at position i

```

More ways to ensure $O(n)$?

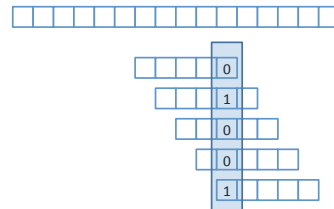
Shift-AND / Shift-OR

- **Ricardo Baeza-Yates , Gaston H. Gonnet**
A new approach to text searching
Communications of the ACM October 1992,
 Volume 35 Issue 10
[\[ACM Digital Library:http://doi.acm.org/10.1145/135239.135243\]](http://doi.acm.org/10.1145/135239.135243) [DOI]
- [PDF](#)

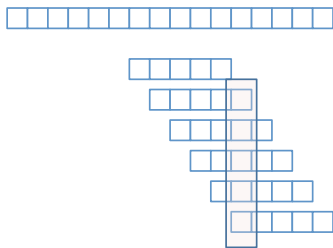
Bit-operations

- Maintain a set of all prefixes that have so far had a perfect match
- On the next character in text update all previous pointers to a new set
- Bit vector: for every possible character

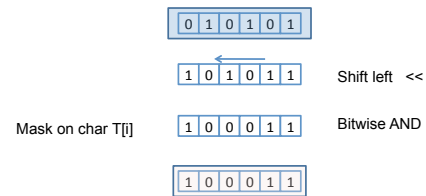
State: which prefixes match?



Move to next:



Track positions of prefix matches



Vectors for every char in Σ

- P=aste

```

a s t e b c d .. z
1 0 0 0 0 ...
0 1 0 0 0 ...
0 0 1 0 0 ...
0 0 0 1 0 ...

```

- T=lasteaed

```

l a s t e a e d
0 1
0 0
0 0
0 0

```

- T= lasteaed

```

l a s t e a e d
0 1 0
0 0 1
0 0 0
0 0 0
    
```

- T= lasteaed

```

l a s t e a e d
0 1 0 0 0 1
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
    
```

- T= lasteaed

```

l a s t e a e d
0 1 0 0 0 1
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
    
```

Summary

Algorithm	Worst case	Ave. Case	Preprocess
Brute force	$O(mn)$	$O(n * (1+1/ \Sigma +..))$	
Knuth-Morris-Pratt	$O(n)$	$O(n)$	$O(m)$
Rabin-Karp	$O(mn)$	$O(n)$	$O(m)$
Boyer-Moore		$O(n/m) ?$	
BM Horspool			
Factor search			
Shift-OR	$O(n)$	$O(n)$	$O(m \Sigma)$

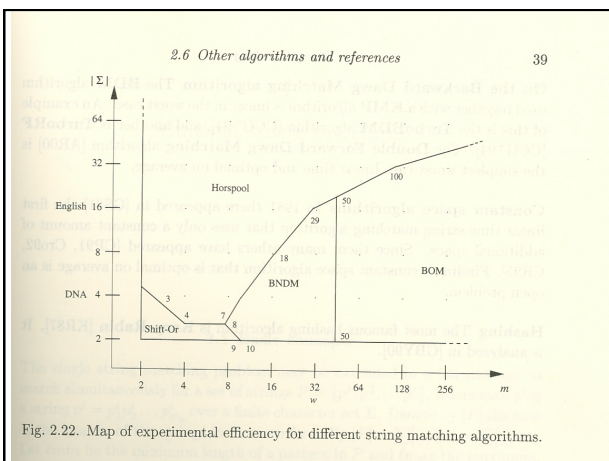


Fig. 2.22. Map of experimental efficiency for different string matching algorithms.