11 Text search

Summer Term 2010

Robert Elsässer

Albert-Ludwigs-Universität Freiburg

Text search

Different scenarios:

Dynamic texts

- Text editors
- Symbol manipulators

Static texts

- Literature databases
- Library systems
- Gene databases
- World Wide Web

Text search

Data type **string**:

- array of character
- file of character
- list of character

Operations: (Let *T*, *P* be of type **string**)

Length:	length ()
i-**th character**:	*T* [*i*]
concatenation:	cat (*T*, *P*) *T.P*

Problem definition

Input:

Text $t_1\, t_2 \,....\, t_n \quad \in \Sigma^n$

Pattern $p_1 p_2 \,...\, p_m \quad \in \Sigma^m$

Goal:

Find one or all occurrences of the pattern in the text,
i.e. shifts i $(0 \leq i \leq n - m)$ such that

$$p_1 = t_{i+1}$$
$$p_2 = t_{i+2}$$
$$\vdots$$
$$p_m = t_{i+m}$$

Problem definition

i $i+1$ $i+m$

Text: t_1 t_2 .... t_{i+1} .... t_{i+m} ..... t_n

Pattern: $\longrightarrow$ p_1 .... p_m

Estimation of cost (time) :

1. # possible shifts: $n - m + 1$ # pattern positions: m
   → $O(n \cdot m)$

2. At least 1 comparison per m consecutive text positions:
   → $\Omega(m + n/m)$

Naïve approach

For each possible shift $0 \le i \le n - m$ check at most m pairs of characters.
Whenever a mismatch occurs, start with the next shift.

```
textsearchbf  := proc (T : : string, P : : string)
#  Input:   Text T und Muster P
# Output:  List L of shifts i, at which P occurs in T
    n := length (T); m := length (P);
    L := [];
    for i from 0 to n-m {
            j := 1;
            while j ≤ m and T[i+j] = P[j]
                    do j := j+1 od;
            if j = m+1 then L := [L [] , i] fi;
    }
    RETURN (L)
end;
```

Naïve approach

Cost estimation (time):

0 0 ... 0 ... 0 ... 0 0 ...

$\xrightarrow{i}$ 0 ... 0 ... 0 1

Worst Case: $\Omega(m \cdot n)$

In practice: mismatch often occurs very early

→ running time ~ $c \cdot n$

Method of Knuth-Morris-Pratt (KMP)

Let t_i and p_{j+1} be the characters to be compared:

$$\boxed{t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots}$$

$$= \quad = \quad = \quad = \quad \neq$$

$$\boxed{p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m}$$

If, at a shift, the first mismatch occurs at
t_i and p_{j+1}, then:

- The last *j* characters inspected in *T* equal the first *j* characters in *P*.
- $t_i \neq p_{j+1}$

Method of Knuth-Morris-Pratt (KMP)

Idea:

Determine $j' = next[j] < j$ such that t_i can then be compared with $p_{j'+1}$.

Determine $j' < j$ such that $P_{1...j'} = P_{j-j'+1...j}$.

Find the longest prefix of P that is a proper suffix of $P_{1...j}$.

$t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots$

$= \; = \; = \; = \quad \neq$

$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$

Method of Knuth-Morris-Pratt (KMP)

Example for determining *next*[*j*]:

t_1 t_2 ... 01011 01011 0 ...

01011 01011 1

01011 01011 1

next[*j*] = length of the longest prefix of P that is a proper suffix of $P_{1\ldots j}$.

Method of Knuth-Morris-Pratt (KMP)

⇒for P = 0101101011, *next* = [0,0,1,2,0,1,2,3,4,5] :

1	2	3	4	5	6	7	8	9	10
0	1	0	1	1	0	1	0	1	1
		0							
		0	1						
					0				
					0	1			
					0	1	0		
					0	1	0	1	
					0	1	0	1	1

Method of Knuth-Morris-Pratt (KMP)

```
KMP := proc (T : : string, P : : string)
# Input: text T and pattern P
# Output: list L of shifts i at which P occurs in T
    n := length (T); m := length(P);
    L :=  []; next := KMPnext(P);
    j  :=  0;
    for i from 1 to n do
          while j>0 and T[i] <> P[j+1] do j := next [j] od;
          if T[i]  = P[j+1]  then j := j+1 fi;
          if j = m then L := [L[] , i-m] ;
                         j  := next [j]
          fi;
     od;
     RETURN (L);
end;
```

Method of Knuth-Morris-Pratt (KMP)

Pattern: abracadabra, *next* = [0,0,0,1,0,1,0,1,2,3,4]

```
a b r a c a d a b r a b r a b a b r a c ...
| | | | | | | | | | |
a b r a c a d a b r a
```

next[11] = 4

```
a b r a c a d a b r a b r a b a b r a c ...
              - - - - ∤
              a b r a c
```

next[4] = 1

Method of Knuth-Morris-Pratt (KMP)

a b r a c a d a b r a b r a b a b r a c ...

\- | | | $\nmid$

a b r a c

next [4] = 1

a b r a c a d a b r a b r a b a b r a c ...

\- | $\nmid$

a b r a c

next [2] = 0

a b r a c a d a b r a b r a b a b r a c ...

| | | | |

a b r a c

Method of Knuth-Morris-Pratt (KMP)

Correctness:

$$t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots$$

$$= \;\; = \;\; = \;\; = \quad \neq$$

$$\boxed{p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m}$$

Situation at start of the for-loop:

$P_{1\ldots j} = T_{i-j\ldots i-1}$ and $j \neq m$

if $j = 0$: we are at the first character of P

if $j \neq 0$: P can be shifted while $j > 0$ and $t_i \neq p_{j+1}$

Method of Knuth-Morris-Pratt (KMP)

If $T[i] = P[j+1]$, j and i can be increased (at the end of the loop).

When P has been compared completely ($j = m$), a position was found, and we can shift.

Method of Knuth-Morris-Pratt (KMP)

Time complexity:

- Text pointer *i* is never reset
- Text pointer *i* and pattern pointer *j* are always incremented together
- Always: *next*[j] < *j*;
  j can be decreased only as many times as it has been increased.

The KMP algorithm can be carried out in time $O(n)$,
if the *next*-array is known.

Computing the *next*-array

next[i] = length of the longest prefix of P that is a proper suffix of $P_{1 \ldots i}$.

next[1] = 0

Let *next*[i-1] = j:

$$\begin{array}{cccccccccc} p_1 & p_2 & \ldots & & \ldots & & p_i & & \ldots & \ldots \\ & & = & = & = & = & \neq & & & \\ & & p_1 & \ldots & p_j & & p_{j+1} & \ldots & p_m & \end{array}$$

Computing the *next*-array

Consider two cases:

1) $p_i = p_{j+1} \rightarrow next[i] = j + 1$

2) $p_i \neq p_{j+1} \rightarrow$ replace j by $next[\, j\,]$, until $p_i = p_{j+1}$ or $j = 0$.
If $p_i = p_{j+1}$, we can set $next[i] = j + 1$,
otherwise $next[i] = 0$.

Computing the *next*-array

```
KMPnext := proc (P : : string)
#Input    :  pattern P
#Output :   next-Array for P
    m := length (P);
    next := array (1..m);
    next [1] := 0;
    j := 0;
    for i from 2 to m do
       while j > 0 and P[i] <> P[j+1]
           do j := next [j] od;
       if P[i] = P[j+1] then j := j+1 fi;
       next [i] := j
    od;
    RETURN (next);
end;
```

Running time of KMP

The KMP algorithm can be carried out in time O(n + m).

Can text search be even faster?