Text Algorithms

Jaak Vilo
2016 fall
Topics

- Exact matching of one pattern (string)
- Exact matching of multiple patterns
- Suffix trie and tree indexes
  - Applications
- Suffix arrays
- Inverted index
- Approximate matching
Algorithms

One-pattern
- Brute force
- Knuth-Morris-Pratt
- Karp-Rabin
- Shift-OR, Shift-AND
- Boyer-Moore
- Factor searches
- Regular expressions(?)
- Weight matrices(?)

Multi-pattern
- Aho Corasick
- Commentz-Walter

Indexing
- Trie (and suffix trie)
- Suffix tree
Exact pattern matching

• \( S = s_1 \, s_2 \ldots \, s_n \) (text) \quad |S| = n \quad (length)

• \( P = p_1 p_2 \ldots p_m \) (pattern) \quad |P| = m

• \( \Sigma \) - alphabet \quad | \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
Animations


• 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, Thierry.Lecroq}@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

Identify the first mismatch!

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:
gcatgcagagagtacagtacg
GCAg....

attempt 2:
gcatgcagagagtacagtacg
g........

attempt 3:
gcatgcagagagtacagtacg
g........

attempt 4:
gcatgcagagagtacagtacg
g........

attempt 5:
gcatgcagagagtacagtacg
g........

attempt 6:
gcatgcagagagtacagtacg
GCAGAGAG

attempt 7:
gcatGCAGAGGtacagtacg
g........
Brute force or 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] ≠ sub[j]
5          jump to next iteration of outer loop
6      return i
7  return not found
```c
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);
}
```
int bf_1( char* pat, char* text )
{
    int m ;
    int count = 0 ;
    char *tp;

    m = strlen(pat);
    tp=text ;

    for( ; *tp ; tp++ ) {
        if( strncmp( 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 $S[i] \not< \not> P[j]$ what do we now?
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

• Make sure that no comparisons “wasted”

• $P$ – longest suffix of any prefix that is also a prefix of a pattern

• Example: $\text{ABCABD}$

\[\text{ABCABD}\]
Automaton for ABCABD

1. A
2. B
3. C
4. A
5. B
6. D
7. NOT A
Automaton for ABCABD

Fail: 0 1 1 1 2 3 1

Pattern: A B C A B D
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

\[ \begin{align*}
i &= 1, \ j &= 0, \ \text{fail}[1] &= 0 \\
\text{repeat} & \\
& \quad \text{if } j = 0 \text{ or } P[i] = P[j] \\
& \quad \quad \text{then } i++ , \ j++ , \ \text{fail}[i] = j \\
& \quad \text{else } j = \text{fail}[j] \\
\text{until } i & \geq m \\
\end{align*} \]
Time complexity of 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 $i-m$
```
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


- 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


- 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 $|\Sigma|$ arithmetics and treat characters as numbers

- In case of hash match – check all $m$ positions
- Hash collisions $\Rightarrow$ Worst case $O(nm)$
Let’s use numbers

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

• $c$ – size of alphabet

• $HS_i = H(S[i..i+m-1])$

• $H(S[i+1..i+m]) = (HS_i - \text{ord}(S[i]) \cdot c^{m-1}) \cdot c + \text{ord}(S[i+m])$

• Modulo arithmetic – to fit value in a word!
• \( \text{hash}(w[0..m-1]) = (w[0]*2^{m-1} + w[1]*2^{m-2} + \cdots + w[m-1]*2^0) \mod q \)
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. \( cm = c^{m-1} \mod q \)
4. \( hp = 0 \); \( hs = 0 \)
5. for \( i = 1 \) .. \( m \) do \( hp = ( hp*c + \text{ord}(p[i]) ) \mod q // H(P) \)
6. for \( i = 1 \) .. \( m \) do \( hs = ( hp*c + \text{ord}(s[i]) ) \mod q // H(S[1..m]) \)
7. if \( hp == hs \) and \( P == S[1..m] \) report match at position

8. for \( i=2 \) .. \( n-m+1 \)
9. \( hs = ( (hs - \text{ord}(s[i-1])*cm) * c + \text{ord}(s[i+m-1]) ) \mod q \)
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] [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:
shift 1, introduce 1, bitwise and
Track positions of prefix matches

Mask on char $T[i]$:

Bitwise AND:

Shift left $<<$

\[0 1 0 1 0 1\]

\[1 0 1 0 1 1\]

\[1 0 0 0 1 1\]

\[1 0 0 0 1 1\]
Vectors for every char in $\Sigma$

- $\mathbf{P} = \text{aste}$

\[
\begin{align*}
\text{a} & \quad \text{s} & \quad \text{t} & \quad \text{e} & \quad \text{b} & \quad \text{c} & \quad \text{d} & \quad \ldots & \quad \text{z} \\
1 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad \ldots \\
0 & \quad 1 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad \ldots \\
0 & \quad 0 & \quad 1 & \quad 0 & \quad 0 & \quad 0 & \quad \ldots \\
0 & \quad 0 & \quad 0 & \quad 1 & \quad 0 & \quad 0 & \quad \ldots \\
0 & \quad 0 & \quad 0 & \quad 0 & \quad 1 & \quad 0 & \quad \ldots \\
0 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad 1 & \quad \ldots \\
\end{align*}
\]
• $T = \text{lasteaed}$
• $T = \text{lasteaed}$

<table>
<thead>
<tr>
<th></th>
<th>l</th>
<th>a</th>
<th>s</th>
<th>t</th>
<th>e</th>
<th>a</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• $T = \text{lasteaed}$

\[
\text{lasteaed}
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
• $T = \text{lasteaed}$
The Shift Or algorithm uses bitwise techniques. Let $R$ be a bit array of size $m$. Vector $R_j$ is the value of the array $R$ after text character $y[j]$ has been processed (see figure 5.1). It contains informations about all matches of prefixes of $x$ that end at position $j$ in the text for $0 < i \leq m-1$:

$$R_j[i] = \begin{cases} 0 & \text{if } x[0, i] = y[j-i, j], \\ 1 & \text{otherwise.} \end{cases}$$

![Diagram of the Shift Or algorithm](http://www.igm.univ-mlv.fr/~lecroq/string/node6.html)

**Figure 5.1:** Meaning of vector $R_j$.

The vector $R_{j+1}$ can be computed after $R_j$ as follows. For each $R_j[i]=0$:

$$R_{j+1}[i + 1] = \begin{cases} 0 & \text{if } x[i + 1] = y[j + 1], \\ 1 & \text{otherwise,} \end{cases}$$

and

$$R_{j+1}[0] = \begin{cases} 0 & \text{if } x[0] = y[j + 1], \\ 1 & \text{otherwise.} \end{cases}$$

If $R_{j+1}[m-1]=0$ then a complete match can be reported.
The C code

```c
int preSo(char *x, int m, unsigned int S[]) {
    unsigned int j, lim;
    int i;
    for (i = 0; i < ASIZE; ++i)
        S[i] = ~0;
    for (lim = i = 0, j = 1; i < m; ++i, j <<= 1) {
        S[x[i]] &= ~j;
        lim |= j;
    }
    lim = ~(lim>>1);
    return(lim);
}

void SO(char *x, int m, char *y, int n) {
    unsigned int lim, state;
    unsigned int S[ASIZE];
    int j;
    if (m > WORD)
        error("SO: Use pattern size <= word size");

    /* Preprocessing */
    lim = preSo(x, m, S);

    /* Searching */
    for (state = ~0, j = 0; j < n; ++j) {
        state = (state<<1) | S[y[j]];
        if (state < lim)
            OUTPUT(j - m + 1);
    }
}
```
The example

Searching phase

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>G</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>G</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

As $R_{12}[7]=0$ it means that an occurrence of $x$ has been found at position $12-8+1=5$. 


### Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Ave. Case</th>
<th>Preprocess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>O(mn)</td>
<td>O(n * (1+1/</td>
<td>Σ</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>O(mn)</td>
<td>O(n)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td></td>
<td>O(n/m) ?</td>
<td></td>
</tr>
<tr>
<td>BM Horspool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factor search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift-OR</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O( m</td>
</tr>
</tbody>
</table>
Fig. 2.22. Map of experimental efficiency for different string matching algorithms.
• R. Boyer, S. Moore: A fast string searching algorithm. *CACM* 20 (1977), 762-772 [PDF]

Find occurrences in text

• Have we missed anything?
Find occurrences in text

• What have we learned if we test for a potential match from the end?
Our search algorithm may be specified as follows:

\[
\text{stringlen} \leftarrow \text{length of string}.
\]

\[
i \leftarrow \text{patlen}.
\]

**top:** if \( i > \text{stringlen} \) then return false.

\[
j \leftarrow \text{patlen}.
\]

**loop:** if \( j = 0 \) then return \( i + 1 \).

if \( \text{string}(i) = \text{pat}(j) \)

then

\[
j \leftarrow j - 1.
\]

\[
i \leftarrow i - 1.
\]

goto **loop**.

close;

\[
i \leftarrow i + \max(\text{delta}_1(\text{string}(i)), \text{delta}_2(j)).
\]

goto **top**.

If the above algorithm returns false, then \( \text{pat} \) does not occur in \( \text{string} \). If the algorithm returns a number, then it is the position of the left end of the first occurrence of \( \text{pat} \) in \( \text{string} \).
Find occurrences in text
Bad character heuristics
maximal shift on $S[i]$

\[ \text{delta}_1( S[i] ) = \begin{cases} |m| & \text{if pattern does not contain } S[i] \\ \text{patlen-j} & \text{max } j \text{ so that } P[j] == S[i] \end{cases} \]

First x in pattern (from end)
void bmInitOcc() {
    char a; int j;
    for(a=0; a<alphabetsize; a++)
        occ[a]=-1;
    for (j=0; j<m; j++) {
        a=p[j];
        occ[a]=j; }
}
Good suffix heuristics

1. $\delta_2( S[i] )$ – minimal shift so that matched region is fully covered or that the suffix of match is also a prefix of $P$
Boyer-Moore algorithm

Input: Text $S[1..n]$ and pattern $P[1..m]$  
Output: Occurrences of $P$ in $S$

preprocess_BM() // delta1 and delta2  
i=m  
while i <= n  
    for( j=m; j>0 and P[j]==S[i-m+j]; j-- ) ;  
    if j==0 report match at position i-m+1  
i = i+ max( delta1[ S[i] ], delta2[ j ] )
• http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm


• Animation: http://www-igm.univ-mlv.fr/~lecroq/string/
Simplifications of BM

- There are many variants of Boyer-Moore, and many scientific papers.
- On average the time complexity is sublinear
- Algorithm speed can be improved and yet simplify the code.
- It is useful to use the last character heuristics (Horspool (1980), Baeza-Yates(1989), Hume and Sunday(1991)).
Algorithm BMH (Boyer-Moore-Horspool)

- RN Horspool - Practical Fast Searching in Strings
  Software - Practice and Experience, 10(6):501-506 1980

Input: Text S[1..n] and pattern P[1..m]
Output: occurrences of P in S
1. for a in Σ do delta[a] = m
2. for j=1..m-1 do delta[P[j]] = m-j
3. i=m
4. while i <= n
5. if S[i] == P[m]
6. j = m-1
7. while ( j>0 and P[j]==S[i-m+j] ) j = j-1 ;
8. if j==0 report match at i-m+1
9. i = i + delta[ S[i] ]
String Matching: Horspool algorithm

• How the comparison is made?

Text:

Pattern:

From right to left: suffix search

• Which is the next position of the window?

Text:

Pattern:

It depends of where appears the last letter of the text, say it ‘a’, in the pattern:

Then it is necessary a preprocess that determines the length of the shift.
Algorithm Boyer-Moore-Horspool-Hume-Sunday (BMHHS)

- Use delta in a tight loop
- If match (delta==0) then check and apply original delta d

Input: Text S[1..n] and pattern P[1..m]
Output: occurrences of P in S
1. for a in Σ do delta[a] = m
2. for j=1..m-1 do delta[P[j]] = m-j
3. d = delta[ P[m] ]; // memorize d on P[m]
4. delta[ P[m] ] = 0; // ensure delta on match of last char is 0
5. for (i=m; i<= n; i = i+d)
6. repeat // skip loop
7. t=delta[ S[i] ]; i = i + t
8. until t==0
9. for( j=m-1; j> 0 and P[j]==S[i-m+j]; j = j-1 )
10. if j==0 report match at i-m+1

BMHHS requires that the text is padded by P: S[n+1]..S[n+m] = P
(in order for the algorithm to finish correctly – at least one occurrence!).
• **Daniel M. Sunday**: A very fast substring search algorithm [PDF]

*Communications of the ACM August 1990, Volume 33 Issue 8*

• **Loop unrolling:**
  - Avoid too many loops (each loop requires tests) by just repeating code within the loop.
  - Line 7 in previous algorithm can be replaced by:

```
7. i += delta[ S[i ] ];
i += delta[ S[i ] ];
i += (t = delta[ S[i ] ]); 
```
2.1.5 Popularity

Although providing a high performance, the degree to which the Boyer-Moore approach has been put into practice may have been curbed to a certain extent by conceptual difficulties in the preprocessing, particularly with the match heuristic. Horspool (1980) observed that “many programmers may not believe that the Boyer and Moore algorithm (if they have heard of it) is a truly practical approach.” Sedgewick (1983) also noted that “both the Knuth-Morris-Pratt and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used.” In fact, Morris discovered that after a few months his initial search implementation in a text editor, being too difficult to understand by other systems programmers, had been ruined by various gratuitous ‘fixes’ (Knuth, Morris and Pratt, 1977).

Hume and Sunday (1991) add that “partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace.” Arguing for a more consistent approach to algorithm development, and against “widespread chaotic algorithm presentation,” Woude (1989) asserts that the Knuth-Morris-Pratt preprocessing should be no more difficult to understand than the actual search procedure itself. There have also been some attempts of late to popularise the Boyer-Moore approach, particularly in its simplified form (e.g. Menico, 1989). As a step towards elucidating this field, Hume and Sunday (1991) have presented a means of classifying the various string-matching algorithms based on the Knuth-Morris-Pratt and Boyer-Moore approaches. A taxonomy of string-matching algorithms, including these two aforementioned approaches, has also been put forward by Watson and Zwaan (1992).
Forward-Fast-Search: Another Fast Variant of the Boyer-Moore String Matching Algorithm

• The Prague Stringology Conference '03
• Domenico Cantone and Simone Faro

Abstract: We present a variation of the Fast-Search string matching algorithm, a recent member of the large family of Boyer-Moore-like algorithms, and we compare it with some of the most effective string matching algorithms, such as Horspool, Quick Search, Tuned Boyer-Moore, Reverse Factor, Berry-Ravindran, and Fast-Search itself. All algorithms are compared in terms of run-time efficiency, number of text character inspections, and number of character comparisons. It turns out that our new proposed variant, though not linear, achieves very good results especially in the case of very short patterns or small alphabets.

• http://cs.felk.cvut.cz/psc/event/2003/p2.html
• PS.gz (local copy)
Factor based approach

• Optimal average-case algorithms
  – Assuming independent characters, same probability

• Factor – a substring of a pattern
  – Any substring
  – (how many?)
Factor based approach

simple. It is shown in Figure 2.13. Suppose that we have read backward a factor $u$ of the pattern, and that we failed on the next letter $\sigma$. This means that the string $\sigma u$ is no longer a factor of $p$, so no occurrence of $p$ can contain $\sigma u$, and we can safely shift the window to after $\sigma$.

Fig. 2.13. Basic idea for shifting the window with the factor search approach. If we failed to recognize a factor of the pattern on $\sigma$, then $\sigma u$ is not a factor of the pattern and the window can be safely shifted after $\sigma$. 
Do not compare characters, but find the longest match to any subregion of the pattern.
Examples

• Backward DAWG Matching (BDM)
  – Crochemore et al 1994

• Backward Nondeterministic DAWG Matching (BNDM)
  – Navarro, Raffinot 2000

• Backward Oracle Matching (BOM)
  – Allauzen, Crochemore, Raffinot 2001
Backward DAWG Matching BDM

Suffix automaton recognises all factors (and suffixes) in $O(n)$

Fig. 2.14. Basic search of the BDM algorithm with the suffix automaton. The variable last stores the beginning position of the longest suffix of the part read that is also a prefix of the pattern.

Do not compare characters, but find the longest match to any subregion of the pattern.
BNDM – simulate using bitparallelism

Bits – show where the factors have occurred so far

String matching

with 0 and 1 as with Shift-And. The number 1, representing an active state at position $j$ of $p$, means that the factor $p_j \ldots p_j+|u|-1$ is equal to $u$. Figure 2.15 shows this relationship. If the pattern is of size less than $w$, then the set fits in a computer word $D = d_m \ldots d_1$.

![Diagram of bit-parallel factor search with table D and pattern]

Fig. 2.15. Bit-parallel factor search. The table $D$ keeps a list of the positions in $p$ where the factor $u$ begins.
BNDM matches an NDA

NDA on the suffixes of ‘announce’

Fig. 2.17. Nondeterministic automaton recognizing all factors of the reverse string of “announce”.
Deterministic version of the same Backward Factor Oracle

Fig. 2.20. Factor oracle for the reverse string of “announce”.
72

Fig. 2.22. Map of experimental efficiency for different string matching algorithms.
String Matching of one pattern

CTACTACTACGTCTATACTGATCGTAGC
TACTACGCGTGATGACTAA

1. Prefix search
2. Suffix search
3. Factor search
Multiple patterns

S

{P}
Why?

• Multiple patterns
  • Highlight multiple different search words on the page
  • Virus detection – filter for virus signatures
  • Spam filters
  • Scanner in compiler needs to search for multiple keywords
  • Filter out stop words or disallowed words
  • Intrusion detection software
  • Next-generation sequencing produces huge amounts (many millions) of short reads (20-100 bp) that need to be mapped to genome!
• ...

...
Algorithms

• Aho-Corasick (search for multiple words)
  – Generalization of Knuth-Morris-Pratt

• Commentz-Walter
  – Generalization of Boyer-Moore & AC

• Wu and Manber
  – improvement over C-W

• Additional methods, tricks and techniques
Aho-Corasick (AC)

- Alfred V. Aho and Margaret J. Corasick (Bell Labs, Murray Hill, NJ)
  Efficient string matching. An aid to bibliographic search.
  *Communications of the ACM*, Volume 18, Issue 6, p333-340 (June 1975)

- **ABSTRACT** This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text. The algorithm consists of constructing a finite state pattern matching machine from the keywords and then using the pattern matching machine to process the text string in a single pass. Construction of the pattern matching machine takes time proportional to the sum of the lengths of the keywords. The number of state transitions made by the pattern matching machine in processing the text string is independent of the number of keywords. The algorithm has been used to improve the speed of a library bibliographic search program by a factor of 5 to 10.

• Generalization of KMP for many patterns
• Text S like before.
• Set of patterns $\mathbf{P} = \{ P_1, \ldots, P_k \}$
• Total length $|\mathbf{P}| = m = \sum_{i=1..k} m_i$
• Problem: find all occurrences of any of the $P_i \in \mathbf{P}$ from $S$
Idea

1. Create an **automaton** from all patterns

2. Match the automaton

- Use the PATRICIA trie for creating the main structure of the automaton
PATRICIA trie


- Abstract **PATRICIA is an algorithm which provides a flexible means of storing, indexing, and retrieving information in a large file, which is economical of index space and of reindexing time. It does not require rearrangement of text or index as new material is added. It requires a minimum restriction of format of text and of keys; it is extremely flexible in the variety of keys it will respond to. It retrieves information in response to keys furnished by the user with a quantity of computation which has a bound which depends linearly on the length of keys and the number of their proper occurrences and is otherwise independent of the size of the library. It has been implemented in several variations as FORTRAN programs for the CDC-3600, utilizing disk file storage of text. It has been applied to several large information-retrieval problems and will be applied to others.**
Trie

From Wikipedia, the free encyclopedia

In computer science, a trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

The term trie comes from "retrieval." Following the etymology, the inventor, Edward Fredkin, pronounces it [trī] ("tree") [1]. However, it is usually pronounced [trai] ("try").[2]
• **Word trie** - a good data structure to represent a set of words (e.g. a dictionary).

• **trie** (data structure)

• **Definition**: A tree for storing strings in which there is one node for every common preffix. The strings are stored in extra leaf nodes.

• See also digital tree, digital search tree, directed acyclic word graph, compact DAWG, Patricia tree, suffix tree.

• **Note**: The name comes from retrieval and is pronounced, "tree."

• To test for a word $p$, only $O(|p|)$ time is used no matter how many words are in the dictionary...
Trie for $P=\{\text{he, she, his, hers}\}$
Trie for $P=\{\text{he, she, his, hers}\}$
How to search for words like he, sheila, hi. Do these occur in the trie?
1. Create an automaton $M_p$ for a set of strings $P$.

2. Finite state machine: read a character from text, and change the state of the automaton based on the state transitions...

3. Main links: $\text{goto}[j,c]$ - read a character $c$ from text and go from a state $j$ to state $\text{goto}[j,c]$.

4. If there are no $\text{goto}[j,c]$ links on character $c$ from state $j$, use $\text{fail}[j]$.

5. Report the output. Report all words that have been found in state $j$. 

Aho-Corasick
AC Automaton (vs KMP)

goto[1,i] = 6.

fail[7] = 3,
fail[8] = 0 ,

**Output table**

state output[j]
2 he
5 she, he
7 his
9 hers
AC - matching

**Input:** Text $S[1..n]$ and an AC automaton $M$ for pattern set $P$

**Output:** Occurrences of patterns from $P$ in $S$ (last position)

1. `state = 0`
2. `for i = 1..n do`
   
3. `while (goto[state, S[i]] == ∅) and (fail[state] != state) do`
4. `state = fail[state]`
5. `state = goto[state, S[i]]`
6. `if (output[state] not empty)`
7. `then report matches output[state] at position i`
Algorithm Aho-Corasick preprocessing I (TRIE)

Input: $P = \{P_1, \ldots, P_k\}$
Output: goto[] and partial output[]
Assume: output(s) is empty when a state s is created;
        goto[s,a] is not defined.

procedure enter($a_1, \ldots, a_m$) /* $P_i = a_1, \ldots, a_m$ */
begin
1. $s=0$; $j=1$;
2. while $\text{goto}[s,a_j] \neq \emptyset$ do // follow existing path
3. \hspace{1em} $s = \text{goto}[s,a_j]$;
4. \hspace{1em} $j = j+1$;
5. for $p=j$ to $m$ do // add new path (states)
6. \hspace{3em} $\text{news} = \text{news}+1$ ;
7. \hspace{3em} $\text{goto}[s,a_p] = \text{news}$;
8. \hspace{3em} $s = \text{news}$;
9. output[$s$] = $a_1, \ldots, a_m$
end

\begin{align*}
\text{begin} \\
10. & \text{news} = 0 \\
11. & \text{for } i=1 \text{ to } k \text{ do enter( } P_i \text{ )} \\
12. & \text{for } a \in \Sigma \text{ do} \\
13. & \hspace{1em} \text{if } \text{goto}[0,a] = \emptyset \text{ then } \text{goto}[0,a] = 0 ; \\
\text{end}
\end{align*}
Preprocessing II for AC (FAIL)

queue = ∅
for a ∈ Σ do
  if goto[0,a] ≠ 0 then
    enqueue( queue, goto[0,a] )
    fail[ goto[0,a] ] = 0

while queue ≠ ∅
  r = take( queue )
  for a ∈ Σ do
    if goto[r,a] ≠ ∅ then s = goto[ r, a ]
      enqueue( queue, s )  // breadth first search
      state = fail[r]
    while goto[state,a] = ∅ do state = fail[state]
    fail[s] = goto[state,a]
    output[s] = output[s] + output[ fail[s] ]
Correctness

• Let string t "point" from initial state to state j.

• Must show that fail[j] points to longest suffix that is also a prefix of some word in P.

• Look at the article...
AC matching time complexity

- **Theorem** For matching the $M_p$ on text $S$, $|S| = n$, less than $2n$ transitions within $M$ are made.

- **Proof** Compare to KMP.

- There is at most $n$ goto steps.

- Cannot be more than $n$ Fail-steps.

- In total -- there can be less than $2n$ transitions in $M$. 
Individual node (goto)

- Full table
- List
- Binary search tree (?)
- Some other index?
AC thoughts

- Scales for many strings simultaneously.
- For very many patterns – search time (of grep) improves(??)
  - See Wu-Manber article
- When k grows, then more fail[] transitions are made (why?)
- But always less than n.
- If all goto[j,a] are indexed in an array, then the size is $|M_p| \times |\Sigma|$, and the running time of AC is $O(n)$.
- When k and c are big, one can use lists or trees for storing transition functions.

Then, $O(n \log(\min(k,c)))$. 
Advanced AC

• Precalculate the next state transition correctly for every possible character in alphabet

• Can be good for short patterns
Problems of AC?

- Need to rebuild on adding / removing patterns
- Details of branching on each node(?)
Commentz-Walter

- Generalization of Boyer-Moore for multiple sequence search
- Beate Commentz-Walter
  A String Matching Algorithm Fast on the Average

Aho and Corasick [AC75] presented a linear-time algorithm for this problem, based on an automata approach. This algorithm serves as the basis for the UNIX tool fgrep. A linear-time algorithm is optimal in the worst case, but as the regular string-searching algorithm by Boyer and Moore [BM77] demonstrated, it is possible to actually skip a large portion of the text while searching, leading to faster than linear algorithms in the average case.
Commentz-Walter [CW79]

- Commentz-Walter [CW79] presented an algorithm for the multi-pattern matching problem that combines the Boyer-Moore technique with the Aho-Corasick algorithm. The Commentz-Walter algorithm is substantially faster than the Aho-Corasick algorithm in practice. Hume [Hu91] designed a tool called gre based on this algorithm, and version 2.0 of fgrep by the GNU project [Ha93] is using it.

- Baeza-Yates [Ba89] also gave an algorithm that combines the Boyer-Moore-Horspool algorithm [Ho80] (which is a slight variation of the classical Boyer-Moore algorithm) with the Aho-Corasick algorithm.
Idea of C-W

- Build a **backward** trie of all keywords
- Match from the end until mismatch...
- Determine the shift based on the combination of heuristics
Horspool for many patterns

Search for ATGTATG, TATG, ATAAAT, ATGTG

1. Build the trie of the inverted patterns

2. $l_{\text{min}}=4$

3. Table of shifts

4. Start the search

Slides courtesy of: Xavier Messeguer Peynoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAAT, ATGTG

The text ACATGCTATGTGACA...

Slides courtesy of: Xavier Messeguer Peyoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

Slides courtesy of: Xavier Messeguer PeyPOCH (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA…

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

Slides courtesy of: Xavier Messeguer Peypoch (http://wwwlsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA…

Slides courtesy of: Xavier Messeguer PeyPOCH (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4 (lmin)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Slides courtesy of: Xavier Messeguer Peypoche (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

Short Shifts!

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
What are the possible limitations for C-W?

- Many patterns, small alphabet – minimal skips
- What can be done differently?
Wu-Manber

- Citeseer: http://citeseer.ist.psu.edu/wu94fast.html [Postscript]
- We present a different approach that also uses the ideas of Boyer and Moore. Our algorithm is quite simple, and the main engine of it is given later in the paper. An earlier version of this algorithm was part of the second version of agrep [WM92a, WM92b], although the algorithm has not been discussed in [WM92b] and only briefly in [WM92a]. The current version is used in glimpse [MW94]. The design of the algorithm concentrates on typical searches rather than on worst-case behavior. This allows us to make some engineering decisions that we believe are crucial to making the algorithm significantly faster than other algorithms in practice.
Key idea

• Main problem with Boyer-Moore and many patterns is that, the more there are patterns, the shorter become the possible shifts...

• Wu and Manber: check several characters simultaneously, i.e. increase the alphabet.
• Instead of looking at characters from the text one by one, we consider them in blocks of size $B$.

• $\log_c 2M$; in practice, we use either $B = 2$ or $B = 3$.

• The SHIFT table plays the same role as in the regular Boyer-Moore algorithm, except that it determines the shift based on the last $B$ characters rather than just one character.
Horspool to Wu-Manber

How do we can increase the length of the shifts?

With a table shift of l-mers with the patterns ATGTATG,TATG,ATAAT,ATGTG

<table>
<thead>
<tr>
<th>1 símbol</th>
<th>2 símbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1</td>
<td>AA 1</td>
</tr>
<tr>
<td>C 4 (lmin)</td>
<td>AC 3 (L\text{MIN}-L+1)</td>
</tr>
<tr>
<td>G 2</td>
<td>AG 3</td>
</tr>
<tr>
<td>T 1</td>
<td>AT 1</td>
</tr>
</tbody>
</table>

\[ \text{AA} = 1, \text{AC} = 3, \text{AG} = 3, \text{AT} = 2, \text{TA} = 2, \text{TT} = 1, \text{TC} = 1, \text{CG} = 3, \text{TA} = 2, \text{TG} = 2 \]

Slides courtesy of: Xavier Messeguer PeyPOCH (http://www.lsi.upc.es/~alggen)
Wu-Manber algorithm

Search for ATGTATG, TATG, ATAAT, ATGTG

into the text: ACATGCTATGTGACATAATA

Experimental length: $\log_{|\Sigma|} 2^{lmin*r}$

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Backward Oracle

• Set Backwards oracle SBDM, SBOM

• Pages 68-72
3.4.3 Set Backward Oracle Matching algorithm

The Set Backward Oracle Matching algorithm (SBOM) [AR99] uses a factor oracle of the set of strings. The factor oracle of $P$ recognizes at least all the factors of the strings in $P$. The search algorithm is similar to SDBM. We slide a window of size $l_{min}$ along the text, reading backward a suffix of the window in the factor oracle. If we fail on a letter $\sigma$, we can safely shift the window past $\sigma$. If not, we reach the beginning of the window and verify a subset of $P$ against the text.

3.4.3.1 Factor oracle of a set of strings

The factor oracle construction on a set of strings resembles the Aho-Corasick automaton construction. The only difference appears when going down the supply path looking for an outgoing transition labeled by $\sigma$. In the Aho-Corasick automaton construction, if this transition does not exist, we just jump to the next state on the supply path (Section 3.2.2). In the factor oracle construction, we create in addition a transition labeled by $\sigma$ from each state on the supply path to the state where the original transition leads.
String matching of many patterns

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
String matching of many patterns

Wu-Manber

SBOM

L_{\text{min}}

(5 patterns)

(10 patterns)

(1000 patterns)

(100 patterns)

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
5 strings

string matching algorithms, showing for all of them the zone in which they are most efficient in practice. The text of 10 megabytes is randomly built, as are the patterns. The experiments were performed on a $w = 32$ bits Ultra Sparc 1 running SunOS 5.6. The sets contain 5, 10, 100, and 1000 strings of the same length, varying from 5 to 100 in steps of 5. We tested all the algorithms presented. The **Wu-Manber** algorithm used in these experiments is the implementation found in *Agrep*. Its performance may vary, depending on the hash functions and the sizes of the tables used.

![Graph](image)

Fig. 3.24. Map of the most efficient algorithms when searching for 5 strings.

The maps are shown in Figures 3.24 to 3.27. The most efficient algorithms are just **Wu-Manber**, the advanced **Aho-Corasick**, and SBOM. As the set grows in size, SBOM becomes more and more attractive. The advanced **Aho-Corasick** also improves in comparison with the others for short strings, since it reads the text only once.
3.6 Other algorithms and references

Fig. 3.25. Map of the most efficient algorithms when searching for 10 strings.
Fig. 3.26. Map of the most efficient algorithms when searching for 100 strings.
Fig. 3.27. Map of the most efficient algorithms when searching for 1000 strings.

Complexity [Sri86] by using additional memory. These algorithms are, however, not efficient in practice.

On matching a set of strings on unbounded alphabets The problem of matching a set of strings of the same length $m$ on an unbounded alphabet has been investigated [Bre95]. The resulting algorithm runs in $O((\log(|P|)/m + 1) \times n)$ comparisons after an $O(|P| \times m \times \log |A|)$ preprocessing time, where $A$ is the alphabet on which the set $P$ is built.
Factor Oracle
Factor Oracle: safe shift
Factor Oracle:

Shift to match prefix of P2?
3.4 Factor based approach

Example using DNA: We search for the set of strings $P = \{\text{ATATAT}, \text{TATAT}, \text{ACGAT}\}$ in the text $\text{AGATACGATATATAACGAT}$. The factor oracle for the reverse set of $P^\text{tmin} = \{\text{ATATA}, \text{TATAT}, \text{ACGAT}\}$ is shown in Figure 3.23.

Fig. 3.23. Factor oracle for the reverse set of $P^\text{tmin} = \{\text{ATATA}, \text{TATAT}, \text{ACGAT}\}$. Double-circled states are terminal.
Construction of factor Oracle

The resulting factor automaton recognizes at least all factors of $P$ [AR99]. The construction algorithm is worst-case time $O(|P|)$. Its pseudo-code is given in Figure 3.20.

---

```
Build_Oracle_Multiple(P = \{p^1, p^2, \ldots, p^r\})
1. OR_trie ← Trie(P)
   δ_OR is its transition function
2. Mark the states that correspond to an entire string $p^i$ as terminal
3. I ← root of OR_trie
4. SO_R(I) ← θ
5. For Current in transversal order Do
   Parent ← parent in OR_trie of Current
   σ ← label of the transition from Parent to Current
   Down ← SO_R(Parent)
9. While Down ≠ θ AND δ_OR(Down, σ) = θ Do
10. δ_OR(Down, σ) ← Current
   Down ← SO_R(Down)
12. End of while
13. If Down ≠ θ Then
14. SO_R(Current) ← δ_OR(Down, σ)
15. Else SO_R(Current) ← I
16. End of if
17. End of for
```

Fig. 3.20. Construction of the factor oracle for a set $P = \{p^1, p^2, \ldots, p^r\}$. The state Current goes through the trie OR_trie built on $P$ in transversal order. The state Down goes down the supply links from the parent of Current looking for an outgoing transition labeled with the same character as between Current and its parent, creating it if it does not exist.
Factor oracle


- [http://www-igm.univ-mlv.fr/~allauzen/work/sofsem.ps](http://www-igm.univ-mlv.fr/~allauzen/work/sofsem.ps)
2 Factor oracle

2.1 Construction algorithm

**Figure 1.** High-level construction algorithm of the Oracle
Definition 1  The factor oracle of a word $p = p_1 p_2 \ldots p_m$ is the automaton build by the algorithm Build_Oracle (figure 1) on the word $p$, where all the states are terminal. It is denoted by Oracle($p$).

The factor oracle of the word $p = abbaab$ is given as an example figure 2. On this example, it can be noticed that the word $aba$ is recognized whereas it is not a factor of $p$.

![Factor oracle of abbaab. The word aba is recognizes whereas it is not a factor.](image)

Figure 2. Factor oracle of abbaab. The word aba is recognized whereas it is not a factor.

Note: all the transitions that reach state $i$ of Oracle($p$) are labeled by $p_i$.

Lemma 1  Let $u \in \Sigma^*$ be a minimal length word among the words recognized in state $i$ of Oracle($p$). Then, $u \in \text{Fact}(p)$ and $i = \text{poccurs}(u, p)$. 
So far

- Generalised KMP -> AhoCorasick
- Generalised Horspool -> CommentzWalter, WuManber
- BDM, BOM
  -> Set Backward Oracle Matching...

- Other generalisations?
Multiple Shift-AND

- $P=\{P_1, P_2, P_3, P_4\}$. Generalize Shift-AND

- **Bits** = 
  
<table>
<thead>
<tr>
<th>P4</th>
<th>P3</th>
<th>P2</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Start** = 
  
  | 1   | 1   | 1   | 1   |

- **Match** = 
  
  | 1   | 1   | 1   | 1   | 1   |