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

Multi-pattern
• Aho Corasick
• Commentz-Walter

Indexing
• Trie (and suffix trie)
• Suffix tree

Exact pattern matching

\[ S = s_1 s_2 ... s_n \quad (text) \quad |S| = n \quad (length) \]

\[ P = p_1 p_2 ... p_m \quad (pattern) \quad |P| = m \]

\[ \Sigma - \text{alphabet} \quad |\Sigma| = c \]

• Does S contain P?
  — Does S = S' P S'' for some strings S' ja S''?
  — Usually m << n and n can be (very) large

Find occurrences in text

Animations

• http://www.igm.univ-mlv.fr/~lecroq/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,
  Thierry.Lecroq@laposte.net
Brute force: BAB in text?

A B A C A B A B A 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;

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 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( 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 $S[i] \not= P[j]$ 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: ABCABD

**Automaton for ABCABD**

**Automaton for ABCABD**

Fail links:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 2 & 3 & 1 \\
\end{array}
\]

Pattern:

\[
\begin{array}{cccccccc}
A & B & C & A & B & D \\
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]
KMP matching

Input: Text \(S[1..n]\) and pattern \(P[1..m]\)
Output: First occurrence of \(P\) in \(S\) (if exists)

\[\begin{align*}
i &= 1; \quad j = 1; \\
&\text{initfail}(P) \quad // \text{Prepare fail links} \\
&\text{repeat} \\
&\quad \text{if } j = 0 \text{ or } S[i] == P[j] \\
&\quad \quad \text{then } i++; \quad j++; \quad // \text{advance in text and in pattern} \\
&\quad \text{else } j = \text{fail}[j] \quad // \text{use fail link} \\
&\quad \text{until } j > m \text{ or } i > n \\
&\quad \text{if } j > m \text{ then report match at } i-m
\end{align*}\]

Initialization of fail links

Algorithm: KMP_Initfail
Input: Pattern \(P[1..m]\)
Output: \(\text{fail}[]\) for pattern \(P\)

\[\begin{align*}
i &= 1, \quad j = 0, \quad \text{fail}[1] = 0 \\
&\text{repeat} \\
&\quad \text{if } j = 0 \text{ or } P[i] == P[j] \\
&\quad \quad \text{then } i++; \quad j++; \quad \text{fail}[i] = j \\
&\quad \text{else } j = \text{fail}[j] \\
&\quad \text{until } i > m
\end{align*}\]

Initialization of fail links diagram

Analysis of time complexity

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

Time complexity of KMP matching?

Input: Text \(S[1..n]\) and pattern \(P[1..m]\)
Output: First occurrence of \(P\) in \(S\) (if exists)

\[\begin{align*}
i &= 1; \quad j = 1; \\
&\text{initfail}(P) \quad // \text{Prepare fail links} \\
&\text{repeat} \\
&\quad \text{if } j = 0 \text{ or } S[i] == P[j] \\
&\quad \quad \text{then } i++; \quad j++; \quad \text{fail}[i] = j \\
&\quad \text{else } j = \text{fail}[j] \quad // \text{use fail link} \\
&\quad \text{until } j > m \text{ or } i > n \\
&\quad \text{if } j > m \text{ then report match at } i-m
\end{align*}\]

Karp-Rabin


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

\[\begin{align*}
\text{h}(P) &\text{ hash of } P \\
\text{h}(T[i..i+m-1]) &\text{ hash of substring of } S
\end{align*}\]

- Goal: \(O(n)\).
- \(f(\text{h}(T[i..i+m-1]) \to \text{h}(T[i+1..i+m])) = O(1)\)
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 => Worst case $O(nm)$

Let’s use numbers

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

Hash

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

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. $q = 33554433$ /* $q$ is a prime */
3. $cm = c^m \mod q$
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 $hs = (hs - ord(s[i-1])*cm) * c + ord(s[i+m-1]) \mod q$ // $H(S[i..i+m])$
7. if $hp == hs$ and $P == S[i..i+m]$ 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 $P == S[i..i+m-1]$ report match at position
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
  [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

Matching in linear time (shift-OR)

State: which prefixes match?
Shift-AND ; shift-OR

Move to next: shift-AND
shift 1, introduce 1, bitwise and
Track positions of prefix matches

Vectors for every char in $\Sigma$

- $P=$aste

- $T=\text{lasteaed}$

- $T=\text{lasteaed}$
The C code

```c
void search(char *text, char *pattern, int m, int n, long *fail, long *state)
{
    int i, j, k;
    unsigned long w;
    unsigned long x;

    // Preprocessing
    for (i = 0; i < m; i++)
    {
        w = DigitChar(pattern[i]);
        state[i] = DigitChar(pattern[m - 1 - i]);
    }

    // Searching
    for (i = 0; i < n; i++)
    {
        w = DigitChar(text[i]);
        int pi = fail[i];
        int qi = state[i];
        int s = qi;
        while (s >= 0 && (qi != w) && (pi != w))
        {
            qi = state[s];
            s = (pi == w) ? pi : fail[s]
        }
        if (s == -1)
            s = fail[s];
        if (qi == w)
            printf("%s matched at position %d\n", pattern, i);
    }
}
```

### Summary

- **Algorithm**
  - **Worst case**
  - **Ave. Case**
  - **Preprocess**

<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>O(n/m) ?</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>BM-Horspool</td>
<td>O(n/m)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Factor search</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(m*</td>
</tr>
</tbody>
</table>

- R. Boyer, S. Moore: A fast string searching algorithm. CACM 20 (1977), 762-772 [PDF]

---

**Fig. 3.72.** Map of experimental efficiency for different string matching algorithms.
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:

\[
\text{stringslen = length of string.}
\]

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

\[
\text{top: if } i > \text{stringslen then return false.}
\]

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

\[
\text{loop: if } j = 0 \text{ then return } j + 1.
\]

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

\[
j = j - 1.
\]

\[
i = i - 1.
\]

\[
\text{goto loop.}
\]

\[
\text{else:}
\]

\[
j = \max(\text{delta}_{\text{string}}(i), \text{delta}_{\text{pat}}(j)).
\]

\[
\text{goto top.}
\]

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}.

Bad character heuristics

maximal shift on \text{S}_i

\[
\text{void bmInitocc()}
\]

\[
\text{char a; int j;}
\]

\[
\text{for(a=0; a<alphabetsize; a++)}
\]

\[
\text{occ[a]=-1;}
\]

\[
\text{for (j=0; j<m; j++)}
\]

\[
\text{a=p[j];}
\]

\[
\text{occ[a]=j;}
\]

}
Good suffix heuristics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>s</td>
</tr>
<tr>
<td>1. s</td>
<td>p</td>
</tr>
<tr>
<td>2. s</td>
<td>p</td>
</tr>
</tbody>
</table>

delta[ 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] )

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

Text: |
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
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++)
6. repeat // skip loop
7. t = delta[S[i]]; i += t
8. until t==0
9. for (j=m-1; j>0 and P[j] == S[i-m+j]; j--);
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 += t

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 credited to a certain extent by dependence distribution in the preprocessing, particularly with the match function.

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).

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.

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?)

Forward-Fast-Search: Another Fast Variant of the Boyer-Moore String Matching Algorithm

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

- PS.gz (local copy)

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 σ. This means that the string σω is no longer a factor of p, so no occurrence of p can contain σω, and we can safely shift the window to after σ.

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 σ, then σω is not a factor of the pattern and the window can be safely shifted after σ.
Factor searches

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 \(O(n)\)

BNDM — simulate using bitparallelism

Bits — show where the factors have occurred so far

BNDM matches an NDA

NDA on the suffixes of ‘announce’

Deterministic version of the same Backward Factor Oracle
Multiple patterns

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)
- ACM:DOI PDF
- 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.

References:

• Generalization of KMP for many patterns
• Text $S$ like before.
• Set of patterns $P = \{ P_1, \ldots, P_k \}$
• Total length $|P| = m = \Sigma_{i=1..k} m_i$
• Problem: find all occurrences of any of the $P_i \in P$ 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 re-arrangement 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.
- [ACM:DOI PDF]

**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 prefix. 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 $w$, only $O(|w|)$ time is used no matter how many words are in the dictionary...
Aho-Corasick

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 $j$.

AC Automaton (vs KMP)

goto[1, i] = 6 ;
fail[7] = 5,
fail[8] = 0 ,

Output table
state output[j]
2 he
5 she, ha
7 his
9 hers

Algorithm Aho-Corasick preprocessing I (TRIE)

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 $\text{goto}[\text{state},S[i]] = \emptyset$ and $\text{fail}[\text{state}] = \text{state}$ do
4. state = $\text{fail}[\text{state}]$
5. state = $\text{goto}[\text{state},S[i]]$
6. if $\text{output}[\text{state}]$ not empty
7. then report matches $\text{output}[\text{state}]$ at position $i$
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 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.

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||Σ||Σ|, 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

Individual node (goto)

• Full table
• List
• Binary search tree(?)
• Some other index?
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
Proceedings of the 6th Colloquium, on Automata,
Languages and Programming. Lecture Notes In
Computer Science; Vol. 71, 1979. pp. 118 - 132,
Springer-Verlag

• http://www.fh-albsig.de/de/personen/professoren.php?RID=36
• You can download here my algorithm StringMatchingFastOnTheAverage [PDF, ~17.2 MB] or here StringMatchingFastOnTheAverage (extended abstract) [PDF, ~3 MB]

C-W description

• 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
grep 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,ATAAT,ATGTG

1. Build the trie of the inverted patterns

2. Inmin=4

3. Table of shifts

4. Start the search

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)
What are the possible limitations for C-W?

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

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.

Horspool to Wu-Manber

How do we can increase the length of the shifts?

With a table shift of 1-mers with the patterns ATGTATG, TATG, ATAAT, ATGTG

\[
\begin{array}{c|c|c}
\text{1 symbol} & \text{2 symbols} \\
\hline
A & 1 & \text{AA} \\
C & 1 & \text{AC} \\
G & 3 & \text{(LMIN-L+1)} \\
T & 1 & \text{AG} \\
& & \text{AT} \\
& & \text{DA} \\
& & \text{CC} \\
& & \text{CG} \\
& & \text{TA} \\
& & \text{TG} \\
\end{array}
\]
**Wu-Manber algorithm**

Search for ATGTATG, TATG, ATAT, ATGTG

into the text: ACATGCTATGTGACATAATA

Experimental length: \( \log_{21} 2 \times \text{min}^* \)

---

**Backward Oracle**

- Set Backwards oracle SBDM, SBOM
- Pages 68-72

---

3.1.3 Set Backward Oracle Matching algorithm

The Set Backward Oracle Matching algorithm (SBOM) [AB99] 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 SBDM. We slide a window of size \( \text{min}^* \) along the text, reading backward a suffix of the window in the factor oracle. If we fall 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.

3.1.3.1 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**

- 5 strings
- 10 patterns
- 100 patterns
- 1000 patterns

---

**5 strings**
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.

On searching a set of strings on unbounded alphabets, the problem of searching a set of strings of the same length is even unbounded without additional memory. Three algorithms can, however, meet the objective in practice.

Factor Oracle: safe shift

Shift to match prefix of P2?
30.11.17

**Factor oracle**

| Example using DNA: | For search of the set of patterns \( P = \{\text{ATGCA, GCAGA, TTTTT}\} \) in the text \( W = \text{ATGCA, GCAGA, TTTTTA, ACGAT, ATGCA}\). The factor oracle of the pattern set \( P \) is shown in Figure 3.3.

Fig 3.3. Factor oracle of the pattern set \( P \).

| Construction of factor Oracle |

| The resulting factor automaton recognizes all factors of \( |P|\) \( \text{KMP} \) in the text \( W \). The construction algorithm is presented in \( O(n) \). In pseudo-code is given in Figure 3.3.

**Factor oracle**

- http://portal.acm.org/citation.cfm?id=647009.712672&coll=GUIDE&dl=GUIDE&CFID=31549541&CFTOKEN=61811641#
- http://www.igm.univ-mlv.fr/~allauzen/work/sofsem.ps

**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 =

- Start =

- Match =

<table>
<thead>
<tr>
<th></th>
<th>$P_4$</th>
<th>$P_3$</th>
<th>$P_2$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

|   | 1     | 1     | 1     | 1     |

|   | 1     | 1     | 1     | 1     |