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

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

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

\[ \Sigma - alphabet \quad |\Sigma| = c \]

- Does S contain P?
  - Does S = S' P S'' fo some strings S' ja S''?
  - Usually \( m \ll n \) and \( n \) can be (very) large

Find occurrences in text

```
P
S
```

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 A B B A A B A B
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 $S[i+j-1] \neq P[j]$ 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 $s[i+j-1] \neq sub[j]$
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_2( char* pat, char* text )
{
  int m :
  int count = 0 :
  char *tp:
  m = strlen(pat):
  tp=pat :
  for( ; *tp ; tsp++ ) { 
    if( strcmp( pat, tsp, 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 $S[i] <> P[j]$ 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

D. Knuth, J. Morris, V. Pratt:
Fast Pattern Matching in strings.

Automaton for ABCABD

D. Knuth, J. Morris, V. Pratt:
Fast Pattern Matching in strings.

Fail:

Pattern:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
KMP matching

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

```plaintext
\begin{align*}
i &= 1; 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++, 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 $P$

```plaintext
\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] \\
& \quad \text{until } i \geq m \\
\end{align*}
```

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 $nr$ 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)

```plaintext
\begin{align*}
i &= 1; 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++, j++, \text{fail}(i) = j \\
& \quad \text{else } j = \text{fail}[j] \\
& \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

ABCABD

 Fail: 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

ABCABD

0 1 1 1 1

Karp-Rabin


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

```
\begin{center}
\begin{tikzpicture}
\fill[green!50, line width=3pt] (0,0) rectangle (5,1); \\
\draw[black, line width=2pt] (0,0) rectangle (5,1); \\
\fill[green!50, line width=3pt] (0,0) rectangle (1,1); \\
\draw[black, line width=2pt] (0,0) rectangle (1,1); \\
\end{tikzpicture}
\end{center}
```

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


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

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

Hash

- “Remove” the effect of $T[i]$ and “Introduce” the effect of $T[i+m]$ – in $O(1)$

- Use base $|Σ|$ arithmetics and treat 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, $h=125$)

- $H(T[1]) = 571$
- $H(T[2]) = (571 - 5\cdot100)*10 + 2 = 712$
- $H(T[3]) = (H(T[2]) - \text{ord}(T[1])*10m)*10 + T[3+m-1]$

- $c$ – size of alphabet

- $HSi = H(S[i..i+m-1])$

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

- Modulo arithmetic – to fit value in a word!

Hash

- $c$ – size of alphabet

- $HSi = H(S[i..i+m-1])$

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

- 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. $q = 33554393$ /* $q$ 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 $P == S[i..i+m-1]$ 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

State: which prefixes match?

Move to next:
shift 1, introduce 1, bitwise and

Track positions of prefix matches

Pattern[i][j]
Shift left $<<$
Bitwise AND

Mask on char T[i]
Vectors for every char in $\Sigma$

- $P$ = 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 ...

- $T$ = lastead

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

- $T$ = lastead

  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

- $T$ = lastead

  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

http://www.igm.univ-mlv.fr/~lecroq/string/node6.html
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(n</td>
<td>σ</td>
<td>)</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>O(n/</td>
<td>σ</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>

• 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:

\[ \text{stringlen} \leftarrow \text{length of string}; \]
\[ i \leftarrow \text{pattern}; \]
\[ \text{loop: if } i > \text{stringlen} \text{ then return false.} \]
\[ j \leftarrow \text{pattern}; \]
\[ \text{loop: if } j = 0 \text{ then return } j + 1. \]
\[ \text{if string}(i) = \text{pat}(j) \]
\[ \text{then} \]
\[ j \leftarrow j - 1; \]
\[ i \leftarrow i - 1; \]
\[ \text{goto loop.} \]
\[ \text{done; \hspace{1cm} goto \text{loop.}} \]

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}.

void bmInittocc() {
  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

\[ \text{delta}( \text{S}[i] ) = \text{minimal shift so that matched region is fully covered} \]

or that the suffix of match is also a prefix of \text{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 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)).

String Matching: Horspool algorithm

- How the comparison is made?
- Which is the next position of the window?

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 $\Sigma$ 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] then // match
6. d = delta[P[m]]; // memorize d on P[m]
7. delta[P[m]] = 0; // ensure delta on match of last char is 0
8. i = i + d; // skip loop
9. until i = n
10. if i==0 report match at i+m-1

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 $\Sigma$ do delta[a] = n
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 + delta[S[i]])
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 $P$: $S[n+1..n+m] = P$
(in order for the algorithm to finish correctly – at least one occurrence!)

• http://www.itf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
• http://biit.cs.ut.ee/~vilo/edu/2005-06/Text_Algorithms/Articles/Exact/
• http://www.igm.univ-mlv.fr/~lecroq/string/
• Animation: http://www.igm.univ-mlv.fr/~lecroq/string/
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]]);
```

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.


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?)

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
String Matching of one pattern

CTACTACTACGCTATACGATCGTAGC
TACTACGTTAGACTAA

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

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 \( |P| = m = \sum_{i=1}^{k} m_i \)
  • 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 indexing 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.
   - 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 $p$, only $O(|p|)$ time is used no matter how many words are in the dictionary.
How to search for words like he, sheila, hi. Do these occur in the trie?

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)

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. $\text{state} = 0$
2. for $i = 1..n$ do
3. while $(\text{goto}[\text{state}, S[i]] \neq \emptyset$ and $(\text{fail}[\text{state}] \neq \text{state})$ do
4. $\text{state} = \text{fail}[\text{state}]$
5. $\text{state} = \text{goto}[\text{state}, S[i]]$
6. if $(\text{output}[\text{state}] \neq \emptyset)$ then report matches $\text{output}[\text{state}]$ at position $i$

Algorithm Aho-Corasick preprocessing I (TRIE)

Preprocessing II for AC (FAIL)

Input: $P = \{ P_1, \ldots, P_k \}$
Output: $\text{goto}[s]$ and partial output $\text{output}[s]$
Assume: output(s) is empty when a state s is created; $\text{goto}[s,a]$ 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. $s = \text{goto}[s, a_j]$;
4. $j = j + 1$;
5. for $j = 1$ to $m$ do // add new path (states)
6. $\text{news} = \text{news} + 1$;
7. $\text{goto}[\text{news}, a_j] = s$;
8. $s = \text{news}$;
9. $\text{output}[s] = a_1, \ldots, a_m$
end

queue = $\emptyset$
for $a \in \Sigma$ do
if $\text{goto}[0, a] \neq \emptyset$ then
enqueue( queue, $\text{goto}[0, a]$ )
$\text{fail}[\text{goto}[0, a]] = 0$
while queue $\neq \emptyset$
$r = \text{take}(\text{queue})$
for $a \in \Sigma$ do
if $\text{goto}[r, a] \neq \emptyset$ then
enqueue( queue, $\text{goto}[r, a]$ )
$\text{state} = \text{fail}[\text{state}]$
while $\text{goto}[\text{state}, a] = \emptyset$ do $\text{state} = \text{fail}[\text{state}]$
$\text{fail}[\text{state}] = \text{goto}[\text{state}, a]$
$\text{output}[s] = \text{output}[s] + \text{output}[\text{fail}[s]]$
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| * |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

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


  You can download here my algorithm StringMatchingFastOnTheAverage (PDF, ~17,2 MB) or
  here StringMatchingFastOnTheAverageExtendedAbstract (PDF, ~3 MB)

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
da 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.

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.

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. Imin=4

3. Table of shifts

4. Start the search

Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG
Horspool for many patterns
Search for ATGTATG, TATG, ATAAAT, ATGTG

The text ACATGCTATGTGACA...
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 [WM93b] and only briefly in [WM93a]. The current version is used in glimpse [WM94]. 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.

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 simbol</th>
<th>2 simbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>4 (min)</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>

ATT: 1
AG: 1
AT: 1
CA: 3
CG: 3
CC: 3
LMIN - L + 1

... 2 symbols

Wu-Manber algorithm

Search for ATGTATG, TATG, ATAAT, ATGTG into the text: ACATGCTATGTGACATAATA

Experimental length: $\log_{\Sigma} 2^{*\text{min}^{*}}$
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
SBDM. We slide a window of size \( \sigma \) in \( P \) 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

- Wu-Manber
- SBOM
- 5 strings
- 10 strings

5 strings

10 strings

Fig. 3.35. Map of the most efficient algorithms when searching for 10 strings.
Factor Oracle

Shift to match prefix of P2?

Factor Oracle: safe shift

Factor Oracle:

Factor oracle
Construction of factor Oracle

2 Factor oracle
2.1 Construction algorithm

Build Oracle($p_0, p_1, \ldots , p_n$)
1. For $i$ from 0 to $n$
2. Create a new state $i$
3. For $i$ from 0 to $n - 1$
4. Build a new transition from $i$ to $i + 1$ by $p_{i+1}$
5. For $i$ from 0 to $n - 1$
6. Let $n$ be a minimal length word in state $i$
7. For all $x \in \Sigma$, $x \neq p_n$
8. If set $I(x) = \{ p_0, p_1, \ldots , p_n \}$
9. Build a new transition from $i$ to $i$-permuter($x$, $p_0, p_1, \ldots , p_n$) by $\#$

Figure 2: High-level construction algorithm of the Oracle

Factor oracle

- http://portal.acm.org/citation.cfm?id=647009.712672&coll=GUIDE&dl=GUIDE&CFID=31549541&CFTOKEN=61816419
- 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 =

<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 |