Exact String Matching

Jaak Vilo
2020 fall

Topics
• Exact matching of one pattern (string)
• Exact matching of multiple patterns

Find occurrences in text

Algorithms
One-pattern
• Naive (brute force)
• Knuth-Morris-Pratt
• Karp-Rabin
• Shift-OR, Shift-AND
• Boyer-Moore
• Factor searches

Multi-pattern
• Aho-Corasick
• Commentz-Walter

Later:
- Automata and regexps
- Approximate matching
- Indexing
- HMM / Transducers / Weight matrices ...

Exact pattern matching
• \( S = s_1 s_2 \ldots s_n \) (text) \( |S| = n \) (length)
• \( P = p_1 p_2 \ldots p_m \) (pattern) \( |P| = m \)
• \( \Sigma \) - alphabet \( |\Sigma| = c \)

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

Animations
• http://www.igm.univ-mlv.fr/~lecroq/strings/
• 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
Naïve: BAB in text?

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

5 MISMATCHES 10 COMPARISONS

Naïve

Algorithm Naive
Input: Text T[1..n] and pattern P[1..m]
Output: All positions i, where P occurs in T.

for i from 1 to n-m+1 do
  for j from 1 to m do
    if T[i+j-1] != P[j] then break;
  if j == m then print i;
return not found;

Brute force or NaïveSearch

function NaiveSearch(T[1..n], P[1..m])
for i from 1 to n-m+1 do
  for j from 1 to m do
    if T[i+j-1] != P[j] then break;
  if j == m then return i;
return not found;
C code

```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);
}
```

C code

```c
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 );
}
```

Simplest (Naive)

![Diagram showing Simplest (Naive) approach]

Question:

• Problems of this method? ☑️
• Ideas to improve the search? ☑️

Simplest (Naive / Brute force)

![Diagram showing Simplest (Naive / Brute force) approach]

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

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

Example: ABCABD

Automaton for ABCABD

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

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
**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
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 ($j=\text{fail}[j]$)
- $i$ 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 $O(m)$

---

**Karp-Rabin**


- Compare in $O(1)$ a hash of $P$ and $S[i..i+m-1]$
- Goal: $O(n)$.
  - $f(\text{h}(T[i..i+m-1]) \rightarrow \text{h}(T[i+1..i+m]) ) = O(1)$

**Karp-Rabin**


- Compare in $O(1)$ a hash of $P$ and $S[i..i+m-1]$
- Goal: $O(n)$.
  - $f(\text{h}(T[i..i+m-1]) \rightarrow \text{h}(T[i+1..i+m]) ) = O(1)$

---

**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, $h=125$)
- $\text{h}(T[1])=571$
- $\text{h}(T[2])=(571-5*100)*10 + 2 = 712$
- $\text{h}(T[3])=(\text{h}(T[2]) - \text{ord}(T[1])*10m)*10 + T[3+m-1]$
hash

- $c$ – size of alphabet
- $HS_i = H(S[i..i+m-1])$
- $H(S[i+1..i+m]) = (HS_i - \text{ord}(S[i]) \cdot c^{-1}) \cdot c + \text{ord}(S[i+m])$
- 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 $hp = (hp \cdot c + \text{ord}(p[i])) \mod q$ // $H(P)$
6. for $i = 1$ .. $m$ do $hs = (hp \cdot c + \text{ord}(s[i])) \mod q$ // $H(S[1..m])$
7. if $hp = hs$ and $P = S[1..m]$ report match at position
8. for $i=2$ .. $n-m+1$
9.    $hs = ((hs - \text{ord}(s[i-1]) \cdot cm) \cdot c + \text{ord}(s[i+m-1])) \mod q$
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/10.1145/155393.155406
  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? 
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 
  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 ...

- $T$=lasteael 
  l a s t e a e d
  0 1
  0 0
  0 0
  0 0
The Shift-OR algorithm uses bitonic technique. Let $B$ be an array of size $m$. Notice $B_i$ is the value of the array $B$ after the right shift $(i)$ has been processed has been done. It contains information about all positions of pattern. If $B_i(x)=1$, then $x$ is present in $B$. If $B_i(x)=0$, then $x$ is not in $B$.

**The C code**

```c
int compare(char a, char b) {
    return a == b;
}

int main() {
    char pattern[] = "lasteaed"
    char text[] = "lasteaed"
    int m = strlen(pattern);
    int n = strlen(text);
    int i, j, k;
    int shift = 0;
    int OR = 0;
    int BT = 0;

    for (i = 0; i < n - m; i++) {
        BT = 0;
        shift = 0;
        OR = 0;
        for (j = 0; j < m; j++) {
            if (compare(pattern[j], text[i + j])) {
                OR = OR | 1 << j;
            } else {
                OR = 0;
                shift = i;
                break;
            }
        }
        if (OR != 0) {
            if (shift == 0) {
                BT = 1;
            } else {
                BT = 0;
            }
        }
    }
    if (BT == 1) {
        printf("Pattern found at position: ");
        printf("%d\n", shift + 1);
    } else {
        printf("Pattern not found\n");
    }
    return 0;
}
```

**Shift – OR**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Shift 0</th>
<th>Shift 1</th>
<th>Shift 2</th>
<th>Shift 3</th>
<th>Shift 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

As $B_i(x)=0$ it means that an occurrence of $x$ has been found at position $12+1=13$. 

**The example**

The searching phase:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Shift 0</th>
<th>Shift 1</th>
<th>Shift 2</th>
<th>Shift 3</th>
<th>Shift 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Shift OR 0 to indicate match, 1 mismatch

- OR: 1
- BT: 1

Description:
The Shift-OR algorithm uses bitonic technique. Let $B$ be an array of size $m$. Notice $B_i$ is the value of the array $B$ after the right shift $(i)$ has been processed has been done. It contains information about all positions of pattern. If $B_i(x)=1$, then $x$ is present in $B$. If $B_i(x)=0$, then $x$ is not in $B$. The vector $B_i$ can be computed after $B_i$ is done. For each $B_i(x)$:

\[ B_i(x) = \begin{cases} 
1 & \text{if } x \equiv \frac{j}{P_i} \text{ (mod } 2^i) \\
0 & \text{else} \end{cases} \]

If $B_i(x)$ is $1$, then a complete match can be reported.
Summary so far

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

Exact String Matching
Boyer-Moore style

Jaak Vilo
2020 fall

Find occurrences in text

- Compare starting from right....
- If P does not contain X, jump ahead by |P| positions
- Or, align with next X in P

• R. Boyer, S. Moore: A fast string searching algorithm. CACM 20 (1977), 762-772 [PDF]
• What have we learned if we test for a potential match from the end?

Our search algorithm may be specified as follows:

```c
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;
    }
}
```

If the above algorithm returns false, then pat does not occur in string. If the algorithm returns a number, then it is the position of the left end of the first occurrence of pat in string.

Bad character heuristics

maximal shift on S[i]

Good suffix heuristics

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
while $i <= n$
  for ($j=m$; $j>0$ and $P[j]=S[i-m+j]$; $j--$) ;
  if $j==0$ report match at position $i-m+1$
  $i = i + \max(\delta_1[S[i]], \delta_2[j])$

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?

| Text | | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|------|
|      |      |      |      |      |      |      |      |      |      |      |

- From right to left suffix search

• Which is the next position of the window?

| Text | | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|------|
|      |      |      |      |      |      |      |      |      |      |      |

It depends on 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 BMH (Boyer-Moore-Horspool)

• RN Horspool: Practical Fast Algorithm in String
  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. $d = \delta[P[m]]$;
4. for ($i=m$ ; $i<=n$ ; $i = i+d$) ;
5. repeat // skip loop
6. $t = \delta[S[i]]$ ; $i = i + t$
7. until $t==0$
8. for ($j=m-1$ ; $j>0$ and $P[j]=S[i-m+j]$ ; $j = j-1$) ;
9. if $j==0$ 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
  $\delta[a] = m$
2. for $j=1..m-1$
do
  $\delta[P[j]] = m-j$
3. $d = \delta[P[m]]$;
4. for ($i=m$ ; $i<=n$ ; $i = i+d$) ;
5. repeat // skip loop
6. $t = \delta[S[i]]$ ; $i = i + t$
7. until $t==0$
8. if ($j==0$ and $d==m-j$) report match at $i-m+1$
9. $i = i + \delta[P[j]]$
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!)

- http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
  Boyer-Moore_original-p762.boyer.pdf
- Animation: http://www.igm.univ-mlv.fr/~lecroq/string/

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

- http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
  Boyer-Moore_original-p762.boyer.pdf
• 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. \( i += \text{delta}[S[i]]; \)
  \( i += \text{delta}[S[i]]; \)
  \( i += (t = \text{delta}[S[i]]); \)

2.1.9 Regularity

Although providing a high performance, the degree of success for these
substring matching problems is subject to the problem's structure. The main
approach differ in the number of operations required, which is more
sensitive in the regularity of the text

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. \( i += \text{delta}[S[i]]; \)
  \( i += \text{delta}[S[i]]; \)
  \( i += (t = \text{delta}[S[i]]); \)

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 Hertz, Quick Search, 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

2.5 Factor based approach

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

  • Factor — a substring of a pattern
    • Any substring
    • (how many?)

 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 $O(n)$

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

BNDM matches an NDA

NDA on the suffixes of ‘announce’

Deterministic version of the same
Backward Factor Oracle
Pattern matching problem and strategies

Alphabet \( \Sigma \)
Test \( S \) of length \( n \)
Pattern \( P \) of length \( m \)

Strategy 1: prefix
Strategy 2: suffix search
Strategy 3: Factor search

Exact String Matching
Multiple patterns

Jaak Vilo
2020 fall

String Matching of one pattern

Multiple patterns

Why?

- Multiple patterns
- 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.

**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 test 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 test 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.

Idea

1. Create an automaton from all patterns
2. Match the automaton
   - Use the PATRICIA trie for creating the main structure of the automaton

**References**
H. Knuth, D. Morris, J. W. and Paten, V. R. For patterns. (1977) [PDF]

**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 retrieval 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 storage of text. It has been applied to several large information-retrieval problems and will be applied to others.

**Note:**
The name comes from retrieval and is pronounced, "tree."
Trie for $P=\{\text{he},\ \text{she},\ \text{his},\ \text{hers}\}$

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. 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 $i$
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, ... , a_m) /* \( P_i = a_1, \ldots , a_m \) */

begin

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

Preprocessing II for AC (FAIL)

queue = \( \emptyset \)

for \( a \in \Sigma \) do

if \( \text{goto}[0,a] \neq \emptyset \) then

\( \text{enqueue}(\text{queue}, \text{goto}[0,a]) \)

fail[\text{goto}[0,a]] = 0

while queue \( \neq \emptyset \) do

\( r = \text{take}(\text{queue}) \)

for \( a \in \Sigma \) do

if \( \text{goto}[r,a] \neq \emptyset \) then

\( s = \text{goto}[r,a]; \)

\( \text{enqueue}(\text{queue}, s) \) // breadth first search

state = fail[r]

while \( \text{goto}[\text{state},a] = \emptyset \) do

\( \text{state} = \text{fail}[\text{state}]; \)

\( \text{fail}[s] = \text{goto}[\text{state},a]; \)

\( \text{output}[s] = \text{output}[s] + \text{output}[\text{fail}[s]] \)
end

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[i,a] are indexed in an array, then the size is \( |M_P|^{|\Sigma|} \), 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

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 fgrep by the GNU project [Hu93] 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. $l_{\text{min}} = 4$
3. Table of shifts
4. Start the search
Horspool for many patterns
Search for ATGTAG, TATG, ATAAAT, ATGTG

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
• 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 the algorithm was part of the second version of agrep (WM92a, WM92b), although the algorithm has not been discussed in (WM92a) and only briefly in (WM92b). 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

• Instead of looking at characters from the text one by one, we consider them in blocks of size B.
• $\log_2(M)$; 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.

• 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, ATAAAT, ATGTG

\[
\begin{array}{c|c|c}
\text{1 symbol} & \text{2 symbols} \\
\hline
A & 1 & \text{AA (LMIN-L+1)} \\
C & 4 (LMIN) & \text{AC} \\
G & 2 & \text{AT} \\
T & 1 & \text{CG} \\
\hline
\end{array}
\]

Experimental length: \( \log_2 2^{2\text{m}} \text{n} \)

Wu-Manber algorithm

Search for ATGTATG, TATG, ATAAAT, ATGTG

SBOM & Lmin

5 patterns

10 patterns

100 patterns

1000 patterns
5 strings

10 strings

100 strings

1000 strings

Factor Oracle

Factor Oracle: safe shift
Factor Oracle:

**Construction of factor Oracle**

2. Factor oracle

2.1 Construction algorithm

1. Build Oracle $\mathcal{O}$ on $p_1, \ldots, p_k$.
2. $T \leftarrow \emptyset$
3. $v \in \bigcup_{i=1}^{k} \{p_i, \overline{p_i}\}$
4. $E \leftarrow \emptyset$
5. Add edge $(v, w)$ to $E$.
6. If $v = p_i$ or $v = \overline{p_i}$, $w = v_i$.
7. If $v = p_i$ or $v = \overline{p_i}$, $w = v_i$.
8. For each $i = 1, \ldots, k$.
9. Add edge $(w, v_i)$ to $E$.
10. Add edge $(w, \overline{v_i})$ to $E$.

Figure 1: 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=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 =