Text Algorithms (4AP)
Lecture 3.2: Multiple pattern matching

Jaak Vilo
2008 fall

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
  - Map many DNA primers on the 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

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.

- Generalization of KMP for many patterns
- Text S like before.
- Set of patterns $P = \{ P_1, ..., P_k \}$
- 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

• Create an automaton from all patterns
• Match 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.
• 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?

- We create an automaton $M_0$ for a set of strings $P$.
- Finite state machine: read a character from text, and change the state of the automaton based on the state transitions...
- Main links: $\text{goto}[j,c]$ - read a character $c$ from text and go from a state $j$ to state $\text{goto}[j,c]$.
- If there are no $\text{goto}[j,c]$ links on character $c$ from state $j$, use $\text{fail}[j]$.
- Report the output. Report all words that have been found in state $j$.

How to search for words like he, sheila, hi. Do these occur in the trie?

Output table

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hi</td>
</tr>
<tr>
<td>1</td>
<td>he</td>
</tr>
<tr>
<td>2</td>
<td>he</td>
</tr>
<tr>
<td>3</td>
<td>his</td>
</tr>
<tr>
<td>4</td>
<td>hers</td>
</tr>
</tbody>
</table>

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

Algorithm Aho-Corabick preprocessing I (TRIE)

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

begin
1. $i = 0$; $r = 0$;
2. while $\text{goto}(i,a) \neq \emptyset$ do // follow existing path
3. $s = \text{goto}(i,a)$;
4. $i = s$;
5. for $a_1$ to $a_n$ do // add new path (state)
6. $s = s + a_1$;
7. $\text{goto}(i,a_1) = s$;
8. $s = s + a_1$;
9. $\text{output}(i) = x_1, ..., x_n$;
end
begin
10. $\text{news} = 0$;
11. for $i = 1$ to $k$ do enter($P_i$);
12. for $a \in \Sigma$ do
13. if $\text{goto}(0,a) = \emptyset$ then $\text{goto}(0,a) = 0$;
end

Preprocessing II for AC (FAIL)

queue = $\emptyset$
for $a \in \Sigma$ do
if $\text{goto}[0,a] \neq \emptyset$ then
enqueue(queue, $\text{goto}[0,a]$);
fail($\text{goto}[0,a]$) = 0

while queue $\neq \emptyset$ do
$r = \text{take(queue)}$;
for $a \in \Sigma$ do
if $\text{goto}[r,a] \neq \emptyset$ then
$s = \text{goto}[r,a]$
enqueue(queue, $s$); // breadth first search
state = $\text{fail}[s]$
while $\text{goto}[state,a] \neq \emptyset$ do state = $\text{fail}[state]$
fail[s] = $\text{goto}[state,a]$
$\text{output}(s) = \text{output}(s) + \text{output fail[s]}$
Correctness
- AC algorithm is correct...
- 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[,] are indexed in an array, then the size is $|M_j|^{|E|}$, 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

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 [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

What are the possible limitations for C-W?

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

Wu-Manber

- Cloneer: http://clozeer.iat.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 \( B \).
• \( \log_2 M \); 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.

Multiple Shift-AND
• \( P = \{P_1, P_2, P_3, P_4\} \). Generalize Shift-AND
  - Bits
  - Start
  - Match

What about other possibilities?
• Generalization of Factor search (?)
  – A possibility for a project?

10 strings

100 strings
1000 strings

Factor Oracle

Factor Oracle: safe shift

Factor Oracle:

Shift to match prefix of P2?

Factor oracle

Construction of factor 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/softum.ps

2 Factor oracle

2.1 Construction algorithm

---

Definition 1. The factor oracle of a word $w = p_1p_2 \ldots p_n$ is the automaton built by the algorithm Build-Oracle (Figure 1) on the word $w$, where all the states are terminal. It is denoted by Oracle($w$).

To each state $i$ of Oracle($w$), there is an edge with a label $p_i$.

The factor oracle of the word $w$ is obtained in given as an example Figure 1. In the example, it can be noticed that the word $a$ is recognized whereas it is not a factor of $w$.

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

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

Lemma 1. Let $i \in S$. If a maximal length word among the words recognized in state $i$ of Oracle($w$), then $w \in \text{Oracle}(w)$.