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

Jaak Vilo
2008 fall

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
  - 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.*

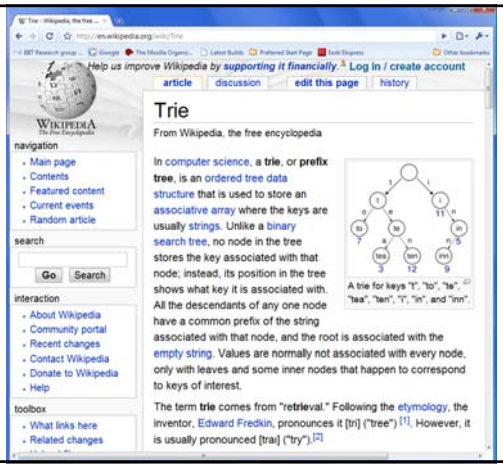
- Generalization of KMP for many patterns
- Text S like before.
- Set of patterns $\mathbf{P} = \{ P_1 , .., P_k \}$
- Total length $|\mathbf{P}| = m = \Sigma_{i=1..k} m_i$
- Problem: find all occurrences of any of the $P_i \in \mathbf{P}$ from S

Idea

- Create an automaton from all patterns
- Match the automaton

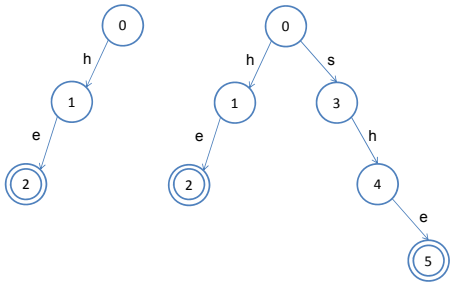
PATRICIA trie

- D. R. Morrison, "PATRICIA: Practical Algorithm To Retrieve Information Coded In Alphanumeric", Journal of the ACM 15 (1968) 514-534.
- **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

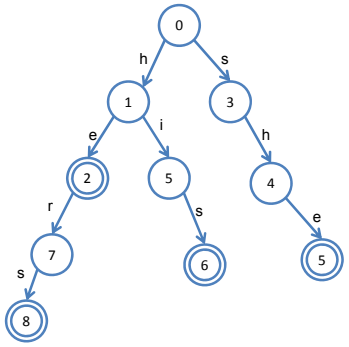


- **Word trie** - a good data structure to represent a set of words (e.g. a dictionary).
- **trie** (data structure)
- **Definition:** A tree for storing strings in which there is one node for every common preffix. The strings are stored in extra leaf nodes.
- See also digital tree, digital search tree, directed acyclic word graph, compact DAWG, Patricia tree, suffix tree.
- **Note:** The name comes from re**trie**val and is pronounced, "tree."
- To test for a word p, only O(|p|) time is used no matter how many words are in the dictionary...

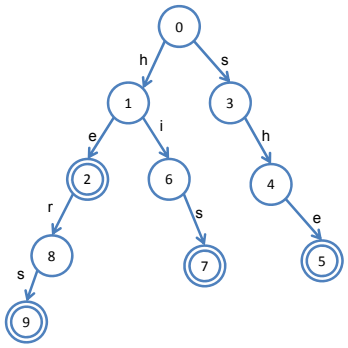
Trie for P={he, she, his, hers}



Trie for P={he, she, his, hers}



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



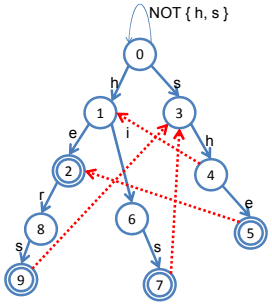
- We create an automaton M_P for a set of strings P.
- Finite state machine: read a character from text, and change the state of the automaton based on the state transitions...
- Main links: goto[j,c] - read a character c from text and go from a state j to state goto[j,c].
- If there are no goto[j,c] links on character c from state j, use 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?

goto[1,i] = 6. ;

fail[7] = 3,
fail[8] = 0 ,
fail[5]=2.

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



AC - matching

Input: Text S[1..n] and an AC automaton M for pattern set P
Output: Occurrences of patterns from P in S (last position)

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

Algorithm Aho-Corasick preprocessing I (TRIE)

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

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

begin
10. news = 0
11. **for** i=1 to k **do** enter(P_i)
12. **for** a ∈ Σ **do**

13. **if** goto[0,a] = ∅ **then** goto[0,a] = 0 ;
end

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

- 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[j,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

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

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

- Wu S., and U. Manber, "A Fast Algorithm for Multi-Pattern Searching," Technical Report TR-94-17, Department of Computer Science, University of Arizona (May **1993).**
- Citeseer: http://citeseer.ist.psu.edu/wu94fast.html [Postscript]
- We present a different approach that also uses the ideas of Boyer and Moore. Our algorithm is quite simple, and the main engine of it is given later in the paper. An earlier version of this algorithm was part of the second version of agrep [WM92a, WM92b], although the algorithm has not been discussed in [WM92b] and only briefly in [WM92a]. The current version is used in glimpse [MW94]. **The design of the algorithm concentrates on typical searches** rather than on worst-case behavior. This allows us to **make some engineering decisions** that we believe are crucial to making the algorithm significantly faster than other algorithms in practice.

Key idea

- Main problem with Boyer-Moore and many patterns is that, the more there are patterns, the shorter become the possible shifts...
- Wu and Manber: check several characters simultaneously, i.e. increase the alphabet.

- Instead of looking at characters from the text one by one, we consider them in blocks of size *B*.
- *$\log_c 2M$; in practice, we use either B = 2 or B = 3.*
- *The SHIFT table* plays the same role as in the regular Boyer-Moore algorithm, except that it determines the shift based on the last *B characters rather than just one character.*

Instead of looking at characters from the text one by one, we consider them in blocks of size B. Let M be the total size of all patterns, $M = k*m$, and let c be the size of the alphabet. As we show in Section 3.1, a good value of B is in the order of $\log_c 2M$; in practice, we use either $B = 2$ or $B = 3$. The SHIFT table plays the same role as in the regular Boyer-Moore algorithm, except that it determines the shift based on the last B characters rather than just one character. For example, if the string of B characters in the text do not appear in any of the patterns, then we can shift by $m - B + 1$. Let's assume for now that the SHIFT table

Multiple Shift-AND

- P={P1, P2, P3, P4}. Generalize Shift-AND
- Bits =

	P4	P3	P2	P1

- Start =

	1	1	1	1

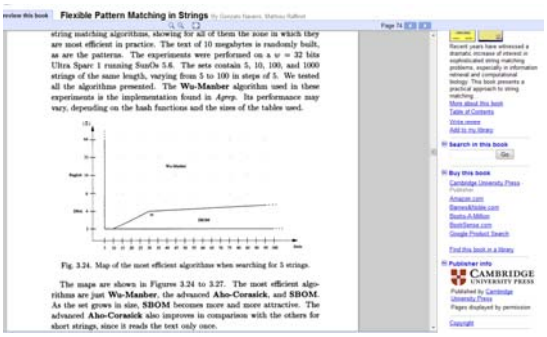
- Match=

	1	1	1	1

What about other possibilities?

- Generalization of Factor search (?)
  - A possibility for a project?

5 strings



10 strings

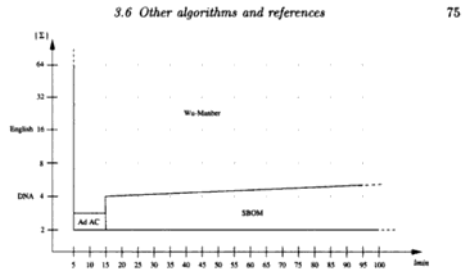


Fig. 3.25. Map of the most efficient algorithms when searching for 10 strings.

100 strings

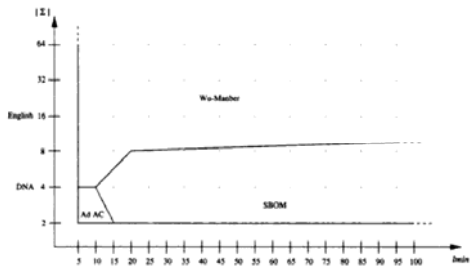
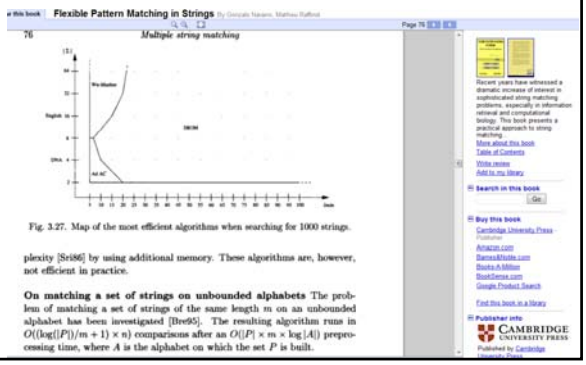


Fig. 3.26. Map of the most efficient algorithms when searching for 100 strings.

1000 strings


Flexible Pattern Matching in Strings

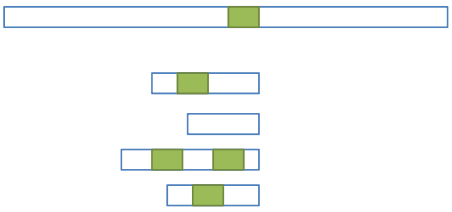
76 Multiple string matching

Fig. 3.27. Map of the most efficient algorithms when searching for 1000 strings.

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

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

Factor Oracle

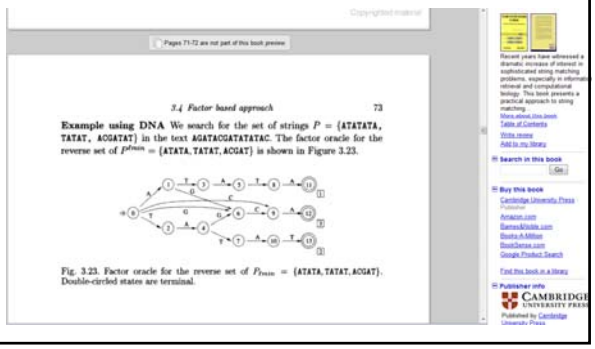


Factor Oracle: safe shift

Factor Oracle:

Shift to match prefix of P2?

Factor oracle

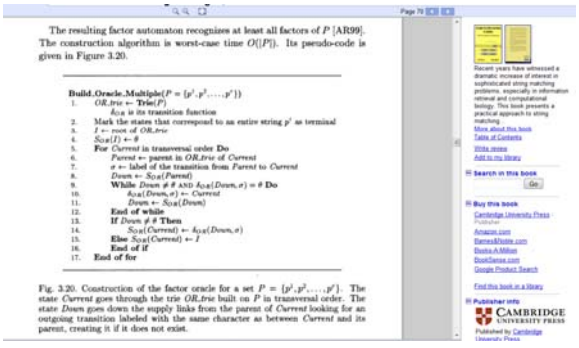


3.4 Factor based approach 73

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

Fig. 3.23. Factor oracle for the reverse set of $P_{rev} = \{$ATATA, TATAT, ACGAT$\}$. Double-circled states are terminal.

Construction of factor Oracle



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

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

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

Factor oracle

- Allauzen, C., Crochemore, M., and Raffinot, M. 1999. Factor Oracle: A New Structure for Pattern Matching. In *Proceedings of the 26th Conference on Current Trends in theory and Practice of informatics on theory and Practice of informatics* (November 27 - December 04, 1999). J. Pavelka, G. Tel, and M. Bartosek, Eds. Lecture Notes In Computer Science, vol. 1725. Springer-Verlag, London, 295-310.
- 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

2 Factor oracle

2.1 Construction algorithm

Build_Oracle($p = p_1 p_2 \dots p_m$)
1. For i from 0 to m
2. Create a new state i
3. For i from 0 to $m-1$
4. Build a new transition from i to $i+1$ by p_{i+1}
5. For i from 0 to $m-1$
6. Let u be a minimal length word in state i
7. For all $\sigma \in \Sigma, \sigma \neq p_{i+1}$
8. If $u\sigma \in \mathrm{Fact}(p_{i-|u|+1} \dots p_m)$
9. Build a new transition from i to $i + \mathrm{poccur}(u\sigma, p_{i-|u|+1} \dots p_m)$ by σ

Figure1. High-level construction algorithm of the Oracle

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

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

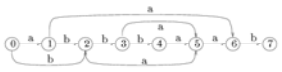


Figure2. Factor oracle of *abbbaab*. The word *aba* is recognizes whereas it is not a factor.

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

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