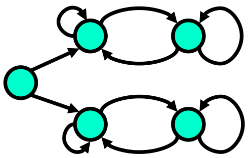
Algorithm Design and Analysis

Victor Adamchik — CS 15-451 — Spring 2014

Lecture 39 — Apr 25, 2014 — Carnegie Mellon University

String Matching - I

J.Morris



D.Knuth

Algorithms on Strings

Pattern Matching
Wild-card Matching
Compute a distance between two strings
Compute a longest substring
Compute a cheapest tree connecting all given strings
Compute a shortest superstring of all strings

Pattern Matching

Let T be a string of length N over a finite alphabet Σ and P be a string of length M over Σ

In a pattern matching problem we search for all occurrences of a pattern P in a text T.

Brute-Force Algorithm

It runs in time O(n m)

Example of worst case:

- T = aaa … ah
- P = aaah
- may occur in images and DNA sequences
- unlikely in English text

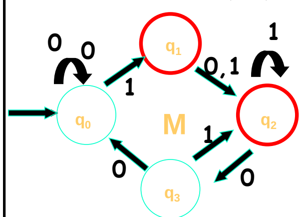
Deterministic Finite Automaton

A finite automaton M is defined as a 5-tuple
$M = (\Sigma, Q, q_0, A, \delta)$

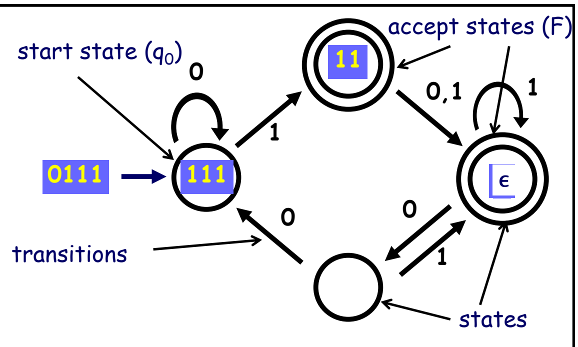
Σ is the alphabet
Q is the set of states
$q_0 \in Q$ is the start state
$A \subseteq Q$ is the set of accept states
$\delta : Q \times \Sigma \to Q$ is the transition function
L(M) = the language of machine M
= set of all strings machine M accepts

$M = (Q, \Sigma, \delta, q_0, F)$

Q = {q0, q1, q2, q3}
δ: $Q \times \Sigma \to Q$ transition function
$\Sigma = \{0,1\}$
$q_0 \in Q$ is start state
A = {q1, q2} $\subseteq$ Q accept states



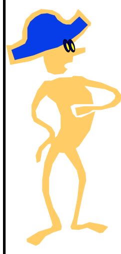
δ	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2



The machine accepts a string if the process ends in an accept state (double circle)

The Knuth-Morris-Pratt Algorithm

(1976)



Build DFA from pattern
Run DFA on multiple texts

Build DFA from pattern

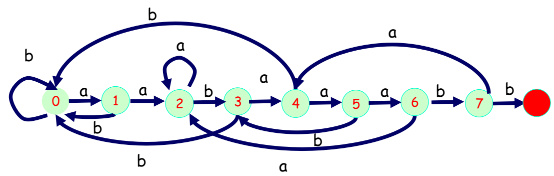
The alphabet is {a, b}.
The pattern is a a b a a a b b.

To create a DFA we consider all prefixes
ε, a, aa, aab, aaba, aabaa, aabaaa, aabaaab, aabaaabb

These prefixes are states. The initial state is ε (empty string). The pattern is the accept state.

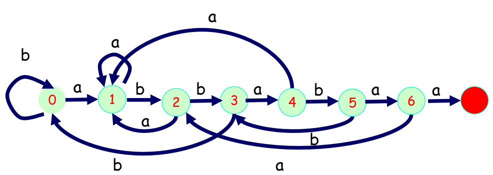
DFA Construction

a a b a a a b b



Matched: a a b a a a b b

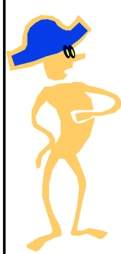
read the string
ababbbabbabaabbab
and print the sequence of states.



a-b-a-b-b-b-a-b-b-a-b-a-a-b-b-a-b
1-2-1-2-3-0-1-2-3-4-5-6-7

The Knuth-Morris-Pratt Algorithm

(1976)

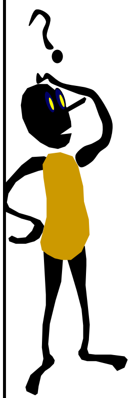


1970 Cook published a paper about a possibility of existence of such algorithm

Knuth and Pratt developed an algorithm

Morris discovered the same algorithm

Building a DFA



What is the worst-case runtime of building a DFA?

$O(M^3 \Sigma)$

M = pattern.length();

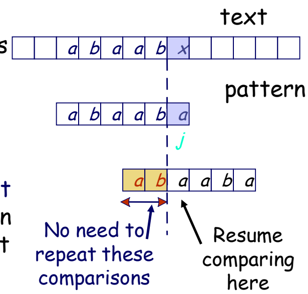
Σ = alphabet.size();

KMP eliminates the need to compute the entire transition function.

The KMP Algorithm - Motivation

Algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?



KMP would say, "but we already had seen this"

a	b	b	a	b							

a	b	b	a	b	b

We were comparing index 5 within T, vs index 5 within P.

"so we could have resumed like this:"

a	b	b	a	b							

a	b	b	a	b	b

We now compare index 5 within T, vs index 2 within P.

So when a match fails, KMP tells us that we should stay at the SAME index within T as we were before, and we just change our index within P (we move back within W).

we need to go back as far as possible in order to guarantee that we don't miss anything.

a	b	a	b	a	b						

a	b	a	b	a	b	a

It would we dangerous to move back to

a	b	a	b	a	b						

a	b	a	b	a	b	a

We could miss something!

So we need to take the LONGEST suffix of P which is a prefix of P

a	b	**a**	**b**	**a**	**b**						

KMP

How much can a string overlap with itself at each position?

a	b	a	b	b
0	0	1	2	0

Compute the length of the longest prefix of P that is a proper suffix of P.

It determines where to go whenever there is a mismatch in the next letter.

Matching

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b

a b a b b

KMP

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b
a b a b b

KMP

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b
a b a b b

KMP

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b
a b a b b

KMP

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b
a b a b b

Mismatch

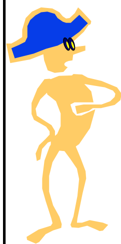
KMP

a	b	a	b	b
0	0	1	2	0

a b a b a b b a b b a b a b a b b
    a b a b b

The KMP Algorithm

Implementation



Failure Function

π[k] = max(j < k | pattern[j] is a suffix of pattern[k])

π[k] is called a failure function, since it represents only backward transitions, in other words, it determines where to go whenever there is a mismatch in the next letter.

"aabaaab", π = {0, 1, 0, 1, 2, 3}

Failure Function

```
int[] pi = new int[pattern.length()];
int x = 0;
for(int p = 1; p < pattern.length(); p++)
{
  while(x > 0 &&
        pattern.charAt(x) != pattern.charAt(p))
  x = pi[x-1];

  if(pattern.charAt(x) == pattern.charAt(p)) x++;
  pi[p] = x;
}
```

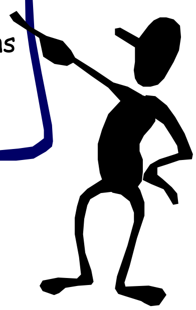
Matching

```
x = 0;
for(int k = 0; k < text.length(); k++)
{
   while(x>0 &&
         pattern.charAt(x) != text.charAt(k))
   x = pi[x-1];

   if(pattern.charAt(x) == text.charAt(k))  x++;

  if(x == pattern.length()) return true;
}
```

The KMP Algorithm

Theorem:

At most 2N comparisons in total



Applications

DNA matching:

DNA consists of small molecules called nucleotides. There are four of them Adenine, Cytosine, Guanine and Thymine. Therefore, {A, C, G, T} creates an alphabet.

Protein matching:

Proteins are composed of amino acids. There are basically 20 amino acids. Hence, a protein can be represented as a string over 20 letters.