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 $\Sigma$ and $P$ be a string of length $M$ over $\Sigma$

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 $O(n \cdot m)$

Example of worst case:
- $T = \text{aaa ... ah}$
- $P = \text{aaah}$
- may occur in images and DNA sequences
- unlikely in English text

Deterministic Finite Automaton

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

$\Sigma$ 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
$\delta : Q \times \Sigma \rightarrow Q$ is the transition function
$L(M)$ = the language of machine $M$
= set of all strings machine $M$ accepts
The machine accepts a string if the process ends in an accept state (double circle).

**Build DFA from pattern**

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

To create a DFA, we consider all prefixes \(\epsilon, a, aa, aab, aaba, aabaa, aabaaab, aabaaabb\). These prefixes are states. The initial state is \(\epsilon\) (empty string). The pattern is the accept state.

**DFA Construction**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

Matched: a b a a a b b

read the string
ababbbabababaabbab
and print the sequence of states.

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

**The Knuth-Morris-Pratt Algorithm**

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?

\[ O(M^3 \Sigma) \]

\( M = \text{pattern.length}() \);
\( \Sigma = \text{alphabet.size}() \);

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"

\[
\begin{array}{cccc}
  a & b & b & a & b \\
  a & b & b & a & b \\
  a & b & b & a & b \\
  a & b & a & b & b \\
\end{array}
\]

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

"so we could have resumed like this:"

\[
\begin{array}{cccc}
  a & b & b & a & b \\
  a & b & b & a & b \\
  a & b & a & b & b \\
  a & b & a & b & b \\
\end{array}
\]

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

KMP

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

\[
\begin{array}{cccc}
  a & b & a & b \\
  0 & 0 & 1 & 2 \\
\end{array}
\]

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

Matching

\[
\begin{array}{cccc}
  a & b & a & b \\
  0 & 0 & 1 & 2 \\
\end{array}
\]
The KMP Algorithm

Implementation
**Failure Function**

\[ \pi[k] = \max\{j < k \mid \text{pattern}[j] \text{ is a suffix of pattern}[k]\} \]

\( \pi[k] \) 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", \( \pi = \{0, 1, 0, 1, 2, 3\} \)

**Matching**

\[ x = 0; \]
\[ \text{for(int } k = 0; k < \text{text.length(); } k++; \} \]
\[ \{ \]
\[ \text{while} (x > 0 \text{ && pattern.charAt}(x) \neq \text{text.charAt}(k)) \]
\[ x = \pi[x-1]; \]
\[ \text{if} (\text{pattern.charAt}(x) == \text{text.charAt}(k)) x++; \]
\[ \text{if} (x == \text{pattern.length()}) \text{return true;} \]

**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.

**The KMP Algorithm**

Theorem: At most 2N comparisons in total

```
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;
}
```