String matching

Knuth-Morris-Pratt algorithm

Idea

After a shift of the pattern, the naive algorithm has forgotten all information about previously matched symbols. So it is possible that it re-compares a text symbol with different pattern symbols again and again. This leads to its worst case complexity of $\Theta(nm)$ (n: length of the text, m: length of the pattern).

The algorithm of Knuth, Morris and Pratt [KMP 77] makes use of the information gained by previous symbol comparisons. It never re-compares a text symbol that has matched a pattern symbol. As a result, the complexity of the searching phase of the Knuth-Morris-Pratt algorithm is in $O(n)$.

However, a preprocessing of the pattern is necessary in order to analyze its structure. The preprocessing phase has a complexity of $O(m)$. Since $m \leq n$, the overall complexity of the Knuth-Morris-Pratt algorithm is in $O(n)$.

Basic definitions

Definition: Let A be an alphabet and $x = x_0 \ldots x_{k-1}$, $k \in \mathbb{N}$ a string of length k over A.

A prefix of x is a substring u with

$u = x_0 \ldots x_{b-1}$ where $b \in \{0, \ldots, k\}$

i.e. x starts with u.

A suffix of x is a substring u with

$u = x_{k-b} \ldots x_{k-1}$ where $b \in \{0, \ldots, k\}$

i.e. x ends with u.

A prefix u of x or a suffix u of x is called a proper prefix or suffix, respectively, if $u \neq x$, i.e. if its length b is less than k.

A border of x is a substring r with

$r = x_0 \ldots x_{b-1}$ and $r = x_{k-b} \ldots x_{k-1}$ where $b \in \{0, \ldots, k-1\}$

A border of x is a substring that is both proper prefix and proper suffix of x. We call its length b the width of the border.

Example: Let x = abacab. The proper prefixes of x are

ε, a, ab, aba, abac, abaca

The proper suffixes of x are

ε, b, ab, cab, acab, bacab

The borders of x are

ε, ab

The border ε has width 0, the border ab has width 2.

The empty string ε is always a border of x, for all $x \in A^+$. The empty string ε itself has no border.

The following example illustrates how the shift distance in the Knuth-Morris-Pratt algorithm is determined using the notion of the border of a string.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b c a b c a b d
a b c a b d
        a b c a b d
```

The symbols at positions 0, ..., 4 have matched. Comparison c-d at position 5 yields a mismatch. The pattern can be shifted by 3 positions, and comparisons are resumed at position 5.

The shift distance is determined by the widest border of the matching prefix of p. In this example, the matching prefix is abcab, its length is j = 5. Its widest border is ab of width b = 2. The shift distance is $j - b = 5 - 2 = 3$.

In the preprocessing phase, the width of the widest border of each prefix of the pattern is determined. Then in the search phase, the shift distance can be computed according to the prefix that has matched.

Preprocessing

Theorem: Let r, s be borders of a string x, where $|r| < |s|$. Then r is a border of s.

Proof: Figure 1 shows a string x with borders r and s. Since r is a prefix of x, it is also a proper prefix of s, because it is shorter than s. But r is also a suffix of x and, therefore, proper suffix of s. Thus r is a border of s.

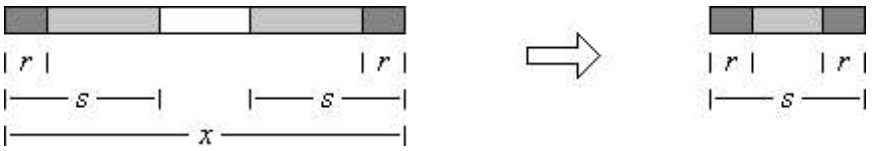

Figure 1: Borders r, s of a string x

If s is the widest border of x, the next-widest border r of x is obtained as the widest border of s etc.

Definition: Let x be a string and $a \in A$ a symbol. A border r of x can be extended by a, if ra is a border of xa.

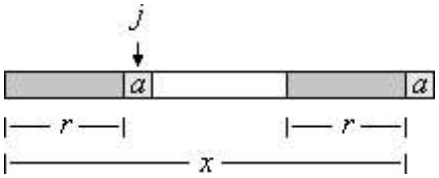

Figure 2: Extension of a border

Figure 2 shows that a border r of width j of x can be extended by a, if $x_j = a$.

In the preprocessing phase an array b of length m+1 is computed. Each entry $b[i]$ contains the width of the widest border of the prefix of length i of the pattern (i = 0, ..., m). Since the prefix ε of length i = 0 has no border, we set $b[0]$ = -1.

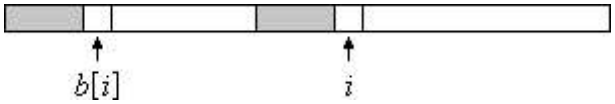

Figure 3: Prefix of length i of the pattern with border of width b[i]

Provided that the values $b[0]$, ..., $b[i]$ are already known, the value of $b[i+1]$ is computed by checking if a border of the prefix p_0 ... p_{i-1} can be extended by symbol p_i. This is the case if $p_{b[i]} = p_i$ (Figure 3). The borders to be examined are obtained in decreasing order from the values $b[i]$, $b[b[i]]$ etc.

The preprocessing algorithm comprises a loop with a variable j taking these values. A border of width j can be extended by p_i, if $p_j = p_i$. If not, the next-widest border is examined by setting $j = b[j]$. The loop terminates at the latest if no border can be extended (j = -1).

After increasing j by the statement `j++` in each case j is the width of the widest border of p_0 ... p_i. This value is written to $b[i+1]$ (to $b[i]$ after increasing i by the statement `i++`).

Preprocessing algorithm

```
void kmpPreprocess()
{
    int i=0, j=-1;
    b[i]=j;
    while (i<m)
    {
        while (j>=0 && p[i]!=p[j]) j=b[j];
        i++; j++;
        b[i]=j;
    }
}
```

Example: For pattern p = ababaa the widths of the borders in array b have the following values. For instance we have $b[5] = 3$, since the prefix ababa of length 5 has a border of width 3.

j:	0	1	2	3	4	5	6
$p[j]$:	a	b	a	b	a	a	
$b[j]$:	-1	0	0	1	2	3	1

Searching algorithm

Conceptually, the above preprocessing algorithm could be applied to the string pt instead of p. If borders up to a width of m are computed only, then a border of width m of some prefix x of pt corresponds to a match of the pattern in t (provided that the border is not self-overlapping) (Figure 4).

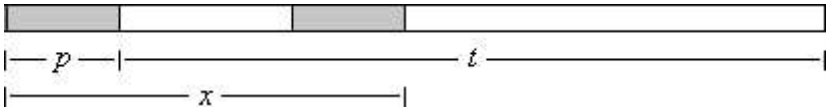

Figure 4: Border of length m of a prefix x of pt

This explains the similarity between the preprocessing algorithm and the following searching algorithm.

Searching algorithm

```
void kmpSearch()
{
    int i=0, j=0;
    while (i<n)
    {
        while (j>=0 && t[i]!=p[j]) j=b[j];
        i++; j++;
        if (j==m)
        {
            report(i-j);
            j=b[j];
        }
    }
}
```

When in the inner while loop a mismatch at position j occurs, the widest border of the matching prefix of length j of the pattern is considered (Figure 5). Resuming comparisons at position $b[j]$, the width of the border, yields a shift of the pattern such that the border matches. If again a mismatch occurs, the next-widest border is considered, and so on, until there is no border left (j = -1) or the next symbol matches. Then we have a new matching prefix of the pattern and continue with the outer while loop.

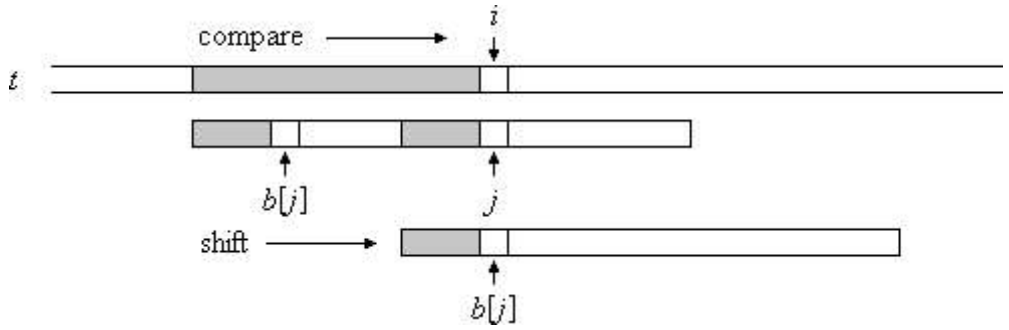

Figure 5: Shift of the pattern when a mismatch at position j occurs

If all m symbols of the pattern have matched the corresponding text window ($j = m$), a function *report* is called for reporting the match at position i-j. Afterwards, the pattern is shifted as far as its widest border allows.

In the following example the comparisons performed by the searching algorithm are shown, where matches are drawn in green and mismatches in red.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b a b b a b a a
a b a b a c
    a b a b a c
        a b a b a c
          a b a b a c
              a b a b a c
```

Analysis

The inner while loop of the preprocessing algorithm decreases the value of j by at least 1, since $b[j] < j$. The loop terminates at the latest when $j = -1$, therefore it can decrease the value of j at most as often as it has been increased previously by `j++`. Since `j++` is executed in the outer loop exactly m times, the overall number of executions of the inner while loop is limited to m. The preprocessing algorithm therefore requires $O(m)$ steps.

From similar arguments it follows that the searching algorithm requires $O(n)$ steps. The above example illustrates this: the comparisons (green and red symbols) form "stairs". The whole staircase is at most as wide as it is high, therefore at most $2n$ comparisons are performed.

Since $m \leq n$ the overall complexity of the Knuth-Morris-Pratt algorithm is in $O(n)$.

References

[KMP 77] D.E. KNUTH, J.H. MORRIS, V.R. PRATT: Fast Pattern Matching in Strings. SIAM Journal of Computing 6, 2, 323-350 (1977)

[Web 1] http://www-igm.univ-mlv.fr/~lecroq/string/

[Web 2] http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/stringmatchingclasses/KmpStringMatcher.java
Knuth-Morris-Pratt algorithm as a Java class source file

Next: [Boyer-Moore algorithm] or ▲

H.W. Lang Hochschule Flensburg lang@hs-flensburg.de Impressum