Abstract

KMP is a string searching algorithm. The problem is to find the occurrence of $P$ in $S$, where $S$ is the given string and $P$ is the Pattern string. Firstly, we must compare each character of $P$ with $S$, if it is matched then we must continue else we must shift $P$ one position to the right and repeat the matching procedure. The running time of the algorithm is $O(m + n)$ where $m$ is the length of pattern $P$ and $n$ is the length of the string $S$. Since we know that $m < n$.

Therefore running time is $O(n)$.

1 Statement of Problem

This problem is to find a string inside an another string. In a pattern $P$ for each position $i$, $sp_i(P)$ is said to be the length of longest suffix of $P[1, 2i]$, which matches to the prefix $P$. It is similar to that of naive algorithm which performs its comparisons from left to right. It also calculates the maximum possible shifts from left to right for the pattern $P$.

2 History

The main reason behind the slowness of the naive algorithm is that after every mismatch the pattern is shifted towards right by one character position to the text so that when we again compare those pairs of characters which has already been scanned. But Knuth-Morris-Pratt observes that it is not necessary to side the pattern by one character position only. They used the information that is gained during previously compared characters and developed a algorithm knows after the names as Knuth-Morris-Pratt algorithm.

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naive algorithm. Knuth-Morris-Pratt algorithm keeps the information that naive approach wasted gathered during
the scan of the text. By avoiding this waste of information, it achieves a running time of $O(n + m)$, which is optimal in the worst case sense. That is, in the worst case Knuth-Morris-Pratt algorithm, we have to examine all the characters in the text and pattern at least once. It uses a minimum of comparison through the use of a pre-computed table. The implementation of the Knuth-Morris-Pratt is efficient because it minimizes the total number of comparisons of the pattern against the input string, while consuming characters is a predictable way from the buffer. There are string matching algorithms that have better average-case running times than Knuth-Morris-Pratt algorithm, but there is no single comparator algorithm with better worst-case characteristics. The worst case performance of the consistent consumption of input characters makes a buffered implementation possible.

3 Time Complexity

The prefix-function values can be computed in $O(m)$. Number of iterations for finding the match takes $2n$ iterations. Totally, the running time of the knuth-morris-pratt algorithm will be $O(m + n)$. In preprocessing phase it is $O(m)$. In searching phase it would be $O(n + m)$ (since it is independent from the alphabet size).

4 Comparison with other string algorithms

4.1 Karp-Rabin Algorithm

4.1.1 Statement of the problem

The Rabin-Karp algorithm is a string searching algorithm that uses hashing function to find the set of pattern strings in a text. The key to performance of Rabin-Karp algorithm is the efficient computations of hash values of the successive substrings of the text. Compared to other algorithms it is inferior to single pattern matching because of its slow worst case behaviour. It is best suits for multiple pattern search.

4.1.2 Complexity of the algorithm

Best running Time: $O(n + m)$.
Worst-case Time: $O(nm)$.

4.2 Boyer-Moore Algorithm

4.2.1 Statement of the problem

It is an efficient string searching algorithm and it has been the standard benchmark for the practical string search literature. This algorithm’s execution time
can be sublinear as not every character of the string to be searched needs to be checked. The algorithm gets faster as the target string becomes larger.

4.2.2 Complexity of the algorithm

Worst-case Time: $O(nm)$.
Best running time: $O(n)$.

4.3 Naive String search algorithm

4.3.1 Statement of the problem

This algorithm is said to be the simplest and least efficient way to see where one string occurs inside another or not. In this algorithm we only have to look at one or two characters for each wrong positions to see that it is wrong position.

4.3.2 Complexity of the algorithm

Average case: $O(n + m)$.
Worst case: $O(nm)$.

5 $O(mn)$ Approach

One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched ‘p’, with the first element of the string ‘S’, in which to locate ‘$p$’. If the element of ‘$p$’ matches the first element of ‘S’, compare the second element of ‘$p$’ with the second element of ‘S’. If match is found proceed likewise until entire ‘$p$’ is found. If a mismatch is found at any position, shift ‘$p$’ one position to the right and repeat comparison beginning from first element of ‘$p$’.

5.1 Working of $O(mn)$

Let us describe with an illustration of how $O(mn)$ approach works. Let us consider a String $S$ and Pattern $p$, where String $S$ is given as “ a b c a b a b a b a c a b a e ” and pattern $p$ is given by “ a b a a ”.

5.1.1 Steps for $O(mn)$ approach for the above example


String $a\ b\ c\ a\ b\ a\ c\ a\ b\ c\ a\ b\ a\ c$

Pattern $a\ b\ a\ a$


String $a\ b\ c\ a\ b\ a\ a\ b\ c\ a\ b\ c\ a\ b\ a\ c$

Pattern $a\ b\ a\ a$

Since a mismatch is detected, shift ‘$p$’ one position to the right and repeat the steps from step 1 to step 3 and repeat matching procedure.

String $a\ b\ c\ a\ b\ a\ a\ b\ c\ a\ b\ c\ a\ b\ a\ c$

Pattern $a\ b\ a\ a$

So, we can see that finally a match is found after shifting ‘$p$’ three times to the right.

5.1.2 Drawbacks of this approach

The running time of the algorithm is very slow, since the matching time is of the order $O(mn)$. The main reason for the slowness of this approach is its repetative comparisons.

6 General Algorithm for all the String searching algorithm
place the pattern at left;
    while
        pattern not fully matched
        and text not exhausted do
        begin
            while
                pattern character differs from
                current text character
            do shift pattern appropriately;
            advance to next character of text
        end;

7 Components of KMP Algorithm

* The Prefix Function, $\pi$:  
   The prefix function, $\pi$ preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself. It is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$. It is used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.

\[
m \leftarrow \text{length}[p] \\
a[1] \leftarrow 0 \\
k \leftarrow 0 \\
\text{for } q \leftarrow 2 \text{ to } m \text{ do} \\
    \text{while } k > 0 \text{ and } p[k+1] \neq p[q] \text{ do} \\
        k \leftarrow a[k] \\
    \text{end while} \\
    \text{if } p[k+1] = p[q] \text{ then} \\
        k \leftarrow k + 1 \\
    \text{end if} \\
    a[q] \leftarrow k \\
\text{end for}
\]

* The KMP Matcher:  
   With string 'S', pattern 'p' and prefix function '\pi' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

Computation of Prefix-function:  
   Let us consider an example of how to compute $\pi$ for the pattern 'p' below:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
</table>

Initially:  
$m = \text{length}[p] = 7$  
$\pi[1] = 0$
where m, □[1], and k are the length of the pattern, prefix function and initial potential value respectively.

Step 1: q = 2, k = 0
□[2] = 0

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>□</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: q = 3, k = 0
□[3] = 1

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>□</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 3: q = 4, k = 1
□[4] = 2

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>□</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 4: q = 5, k = 2
□[5] = 3

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>□</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 5: q = 6, k = 3
□[6] = 1

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>□</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Step 6: $q = 7$, $k = 1$

$\sqcap[7] = 1$

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

After iterating 6 times, the prefix function computations is complete:

<table>
<thead>
<tr>
<th>q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>a</td>
<td>b</td>
<td>A</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

7.1 Run-time complexity

Prefix function can be computed in $O(m)$ time.

8 Algorithm

Step 1: Initialize the input variables:
   $n =$ Length of the Text.
   $m =$ Length of the Pattern.
   $\sqcap =$ Prefix-function of pattern (p).
   $q =$ Number of characters matched.

Step 2: Define the variable:
   $q=0$, the beginning of the match.

Step 3: Compare the first character of the pattern with first character of text.
   If match is not found, substitute the value of $\sqcap$ to $q$.
   If match is found, then increment the value of $q$ by 1.

Step 4: Check whether all the pattern elements are matched with the text elements.
   If not, repeat the search process.
   If yes, print the number of shifts taken by the pattern.

Step 5: look for the next match.
Pseudo-code for KMP algorithm:

\[
\begin{align*}
& n \leftarrow \text{length}[S] \\
& m \leftarrow \text{length}[p] \\
& a \leftarrow \text{Compute Prefix function} \\
& q \leftarrow 0 \\
& \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
& \quad \text{while } q > 0 \text{ and } p[q+1] \neq S[i] \text{ do} \\
& \quad \quad q \leftarrow a[q] \\
& \quad \quad \text{if } p[q+1] = S[i] \text{ then} \\
& \quad \quad \quad q \leftarrow q + 1 \\
& \quad \quad \text{end if} \\
& \quad \quad \text{if } q == m \text{ then} \\
& \quad \quad \quad q \leftarrow a[q] \\
& \quad \quad \text{end if} \\
& \quad \text{end while} \\
& \text{end for}
\end{align*}
\]

8.1 Example of the String Pattern Matching

Now let us consider an example so that the algorithm can be clearly understood. Let text be “y x y z x y z y z x y z x y z x y z x x y z x y z x y z x y z x x y z x x y z x y z x x” and Pattern be “x y z x y z x x”. The space after each character is shown only for clarity. Assume that these strings do not have any other characters in them. The matching process would start by aligning the first character of pattern (from left) on the first character of the text. Initially this situation would be as shown below.

```
Text: y x y z y x y z x x y z x y z x y z x x y z x x
Pattern: x y z x x y z x x
```

We start matching characters of the pattern from left to right. It mismatches at the first character itself so we slide the pattern towards right by one position. We try as shown below.

```
Text: y x y z y x y z x x y z x y z x x y z x x
Pattern: x y z x y z x x
```

```
Text: y x y z y x y z x x y z x x y z x x y z x x
Pattern: x y z x x y z x x
```
Now first three characters of the pattern match with the three characters of the text starting at position ie the fourth character of the pattern does not match with the corresponding character of the text where we know that \( x \ y \ z \ r \) where \( r \neq x \). It is some character other than \( x \). Without making any further tests with the Text, we can safely conclude that it is not worth shifting pattern right by one, two or three positions, as such a slide would not result in a proper alignment. Now lets us move four characters from the current position. The situation would be

\[
\text{Text: } \text{y} \ x \ y \ z \ y \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x
\]

\[
\text{Pattern: } x \ y \ z \ x \ y \ z \ x \ y \ z
\]

Now eighth character of Pattern mismatches. The first seven characters now matches. It means that the last eight characters of Text tested are \( x \ y \ z \ x \ y \ z \ x \ r \) where \( r \neq z \). Shifting Pattern one or two places would not be correct but a three place shift might be as follows

\[
\text{Text: } \text{y} \ x \ y \ z \ y \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x
\]

\[
\text{Pattern: } x \ y \ z \ x \ y \ z \ x \ y \ z
\]

As we have shifted it by appropriately choosing the shift mount so that the characters prior to mismatch location of the Text do match. In this case we find the mismatch has occurred at the same location of the Text. Since the time three position shift is not enough we attempt a four position shift.

\[
\text{Text: } \text{y} \ x \ y \ z \ y \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x
\]

\[
\text{Pattern: } x \ y \ z \ x \ y \ z \ x \ y \ z
\]

Still we get a mismatch. This time a shift of three positions would be essential.

\[
\text{Text: } \text{y} \ x \ y \ z \ y \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x \ y \ z \ x
\]

\[
\text{Pattern: } x \ y \ z \ x \ y \ z \ x \ y \ z
\]
This completes the matching of the entire pattern. It is known from the above example that it takes 14 shifts to complete the matching process for the above substring of the text.

9 KMP as a Finite State Automata

We can use finite automata to do string matching by having an automaton set up to match just one word, and when we get to the final state, we know that we have found the substring in the text. This technique is very efficient because a finite automaton functions by looking at each point symbol once. This means that we can do the matching with a finite automaton in no more than \( T \) comparisons. This is not an easy task, and although algorithms are available that can do this, they take a lot of time. Because of this, finite automata are not a good general purpose solution to string matching.

When constructing a finite automaton to look for a substring in a piece of text, it is easy to build the links that move us from the start state to the final accepting state, because they are just labeled with the characters of the sub-string. The problem occurs when we begin to add additional links for the other characters that do not get us to the final state. The Knuth-Morris-Pratt algorithm is based on finite automata but uses a simpler method of handling the situation in which the characters do not match. In Knuth-Morris-Pratt, we label the state with the symbols that should match at that point. We then need only two links from each state one for a successful match and the other for a failure. The success link will take us to the next node in the chain, and the fail link will take us back to a previous node based on the word pattern.

For a pattern \( P[1...m] \), its string matching automaton can be constructed as follows.
1. The state set \( Q \) is \([0, 1, ..., m]\). The start state \( q_0 \) is state 0, and state \( m \) is the only accepting state.
2. The transition function \( \delta \) is defined by the following equation, for any state \( q \) and character \( a \).

Let us consider an example:
\[ P=ababaca \]
Let us consider an example to show string matching by using the finite automaton.

Let $P=\text{ababaca}$ $T=\text{ababacaba}$

Step 1: $q = 0$, $T[1] = a$. Go into the state $q = 1$.
Step 2: $q = 1$, $T[2] = b$. Go into the state $q = 2$.

Therefore from the above example we can see that the matched string is described in the form of steps and finally the string matching gets halted after all the strings in the pattern are matched from the given text.

10 Advantages and Disadvantages of KMP

* Advantages:

→ It runs in optimal time: $O(m+n)$. Which is very fast.
→ The algorithm never needs to move backwards in the input text $T$. This makes the algorithm good for processing very large files.

* Disadvantages:

→ Doesn’t work so well as the size of the alphabets increases. Due to which the chances of mismatch is more.
References

