String Matching

Suresh Jakka

Indiana State University

sjakka1@sycamores.indstate.edu

December 13, 2014

Abstract

This document explains string matching algorithms and shows the detailed description and the implementation of the Naive and the KMP string matching algorithms with an example. The efficiency and time complexities of the above algorithms are discussed along with the comparison of other available algorithms such as Rabin-Karp and Finite-State automaton are discussed.

1 Introduction

In computer science, string searching algorithms, sometimes called string matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

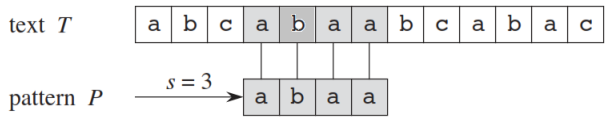
Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are vectors of elements of Σ. The Î£ may be a usual human alphabet (for example, the letters A through Z in the Latin alphabet). Other applications may use binary alphabet (Σ = 0,1) or DNA alphabet (Σ = A,C,G,T) in bioinformatics.

2 Definitions and Background

We formalize the String Matching problem as follows :

- The text is an array T[1..n] of length n
- The pattern is an array P[1...m] of length m $\leq$ n
- We assume further assume the elements of P and T are characters drawn from a finite alphabet Σ . P and T are called the strings of characters.

We say that P occurs with shift s in text T if 0 $\leq$ s $\leq$ n - m and T[s + 1...s + m] = p[1...m].If P occurs with shift s in T, then we call s a valid shift.



The above figure shows the string matching problem. The goal is to find all the occurences of P = abaa in the text T = abcabaabcabac. The pattern occurs at shift s=3.

3 The Algorithm

3.1 The Naive string-matching algorithm

The naive algorithm finds the shift using a loop that checks the condition P[1...m] = T[s+1...s+m] for each of the n-m+1 possible values of s.

```
Naive-String-Matcher(T,P)
   1  n <- length[T]
   2  m <- length[P]
   3  for s <- 0 to n-m
   4  do if P[1...m] = T[s+1...s+m]
        then print "Pattern occurs with shift s "
```

Procedure NAIVE-STRING-MATCHER takes time ((n-m+1)m), and this bound is tight in the worst case. The worst case running time is thus Θ((n-m+1)m), which is $\Theta(n^2)$ if m=[n\2]. The running time of NAIVE-STRING-MATCHER is equal to its matching time, since there is no preprocessing.

3.2 The Knuth-Morris-Pratt algorithm

This is linear-time string matching algorithm proposed by Knuth, Morris and Pratt. KMP spends a little time precomputing a table and then it uses that table to do an efficient search of the string in O(k). The difference is that KMP makes the use of previous match information that the naive algorithm does not.

```
KMP-Matcher(T,P)
 n <- length[T]
 m <- length[p]
 F <- Prefix Function (P)
 q <- 0
 for i <- 1 to n
    do while q > 0 and P[q+1] != T[i]
      do q <- F[q]
    if P[q+1] = T[i]
      then q <- q+1;
    if q = m
      then pattern occurs with shift i-m;
   q <- F[q]
```

```
Prefix Function(P)
m <- m
F[1] <- 0
k <- 0
for q <- 2 to m
  do while k > 0 and P[k+1] != P[q]
    do k <- F[k]
  if P[k+1] = P[q]
    then k <- k+1
  F[q] <= k
return F
```

Running Time The running time of Prefix-Function is Θ(m), where 'm' is the length of the pattern since the number of iterations is θ(m).Here we compute prefix function based on the pattern. Similarly the matching time for KMP-Matcher is θ(n) where n is the length of the text. The Complexity of the overall algorithm is O(m+n).

3.2.1 Example

Let us Consider an example dealing with both the algorithms mentioned above and find the difference.
Text T be **abaababaabacabaababaabaab**
Pattern P be **abaababaabaab**

Using Naive Algorithm
Every character of the pattern is compared with the text, so there is a mismatch at the position 12 between the characters **a and c**

abaababaaba c **abaababaabaab**
abaababaaba a **b**

Now the pattern is shifted by one position and then the comparison starts from the first character of the pattern. But the character does not match.

a b **baababaabacabaababaabaab**
  a **baababaabaab**

In this way when ever a mismatch occurs, the pattern is shifted by one position and comparison starts from the first character of the pattern until the pattern is found. If the pattern reaches the end of the text then there is a unsuccessful match and the algorithm terminates. In this example finally the match is found from position 13 in the text after doing 12 shifts.

abaababaabac**abaababaabaab**
            abaababaabaab

Using Knuth-Morris-Pratt Algorithm : As we know that KMP algorithm preprocess the pattern, the output of the prefix function is an array of values which gives the information of shifts to be done. The values for the pattern in our example :

pos	1	2	3	4	5	6	7	8	9	10	11	12	13
P	a	b	a	a	b	a	b	a	a	b	a	a	b
F(x)	0	0	1	0	0	1	2	3	4	5	6	0	0

Now the match operation gets started by comparing the characters of the pattern. But there is a mismatch at the position 13 as we have seen in the naive algorithm. Now the shift has to be calculated. Since there is a mismatch at the position 13; this indicates that the first 12 characters have been matched with the text. We take this information as our advantage in calculating the shift.
The value for the position 12 in the table above is 0. So we shift the pattern by 1, if the value for the position is >0 then shift = pos−f(x). In this case we shift the pattern by 1 position.

a b **aababaabacabaababaabaab**
a **baababaabaab**

Now there is a mismatch at position 1 in the pattern, since the f(x) for this position is 1. The shift is 1.

aba a **babaabacabaababaabaab**
a b **aababaabaab**

mismatch at pos=2; since f(2)=0 ; shift=1
abaa b **abaabacabaababaabaab**
a **baababaabaab**

mismatch at pos=1 ; f(1)=0; shift = 1;
abaababaaba c **abaababaabaab**
abaaba b **aabaab**

mismatch at pos=7; f(7)=2; shift=7-2=5; Here we shift the pattern by 5 which reduces the unnecessary shifts, unlike the naive algorithm.

abaababaaba c **abaababaabaab**
a b **aabaaabaab**

Similarly the shifts are calculated and the match is found by comparing the text and pattern until it reaches the end of text. Here, in the above example the pattern is found after 6 shifts. Hence there is a significant improvement in shifts when compared with the naive algorithm. This reduces the running time from O(mn) to O(m+n).

3.3 Comparing different String Matching Algorithms

Here m is the length of the pattern and n is the length of the text.

Algorithm	**PreprocessingTime**	**MatchingTime**
Naive Algorithm	0	O(mn)
KMP Algorithm	O(m)	O(m+n)
Rabin Karp	O(m)	O(n)
Finite-State Automaton	O(m$\mid \sum \mid$)	O(n)

4 Future Work

Longest Common Subsequence Problem In future, I would like to work on the longest common subsequence problem. LCS is the problem of finding the longest subsequence common to all sequences in a set of sequences. It differs from problems of finding common substring: unlike substrings, sequences are not required to occupy the consecutive portions within the original sequences. This problem is NP-hard for the arbitary number of input sequences. This problem can be solved in polynomial time by dynamic programming if the number of sequences is constant.
Assume N sequences of lengths n_1,.....n_N. Navie algorithms would search each 2^{n1} subsequences of the first sequence to determine whether they are also subsequences of the remaining sequences, so the time for this algorithm would be $\bigcirc(2^{n1} \sum n_i)$

5 References

1. Introduction to algorithms, 3rd edition by Thomas Cormen, Charles, Ronald and Clifford.
2. wikipedia reference