المركز الوطني للمتميزين
NATIONAL CENTER FOR THE DISTINGUISHED
/NCDSyria

String matching algorithms

تقديم الطالب: سليمان ضاهر

اشراف المدرس: علي جنيدي

للعام الدراسي:2017/2016

The Introduction

The introduction to information theory is quite simple. The invention of writing occurred 5000 years ago, but no other culture thought of manipulating the written data more than the people of IT revolution. Coding makes our life easier, creates huge hopes in domains of security, compression and data transmission. No other introduction to coding would be more decent than the string matching theory. A variety of algorithms were discussed during this paper

Abstract

We formalize the string-matching problem as follows. We assume that the text is an array T [1 . . n] of length n and that the pattern is an array P[1 . . m] of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet . For example, we may have = {0,1} or = {a, b,..., z}. The character arrays P and T are often called strings of characters. We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position s + 1 in text T) if $0 \leq s \leq n - m$ and T [s + 1 . .s + m] = P[1 . . m] (that is, if T [s + j] = P[j], for $1 \leq j \leq m$). If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T. and in this research we will study the most important algorithms that do this mission

Contents

Table of figures

The string matching algorithms and its importance

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem can greatly aid the responsiveness of the text-editing program. String-matching algorithms are also used, for example, to search for particular patterns in DNA sequences.

The naive string-matching [1]

The brute force algorithm consists in checking, at all positions in the text between 0 and n-m

The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the

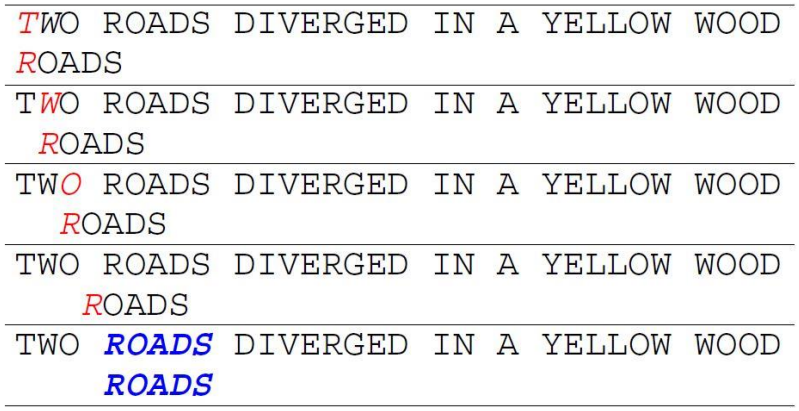

Figure 1 an example of the naïve string matcher

template equal the corresponding characters in the text, as illustrated

Matching steps:

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]
5. **then** print "Pattern occurs with shift" s

First there is a for loop that considers each possible shift explicitly. Then there is a test to determine whether the current shift is valid or not(4); this test involves an implicit loop(4) to check corresponding character positions

Complexity:

Procedure NAIVE-STRING-MATCHER takes time O((n − m + 1)m), and this bound is tight in the worst case. For example, consider the text string a^n (a string of n a's) and the pattern a^m. For each of the n−m+1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus O((n − m + 1)m), which is $O(n^2)$ if m = [n/2]. The running time of NAIVE-STRING-MATCHER is equal to its matching time, since there is no preprocessing.

As we shall see, NAIVE-STRING-MATCHER is the simplest way to match a pattern with a text because it just depends on one loop and it doesn't require any preprocessing functions for the pattern and the text, but it is inefficient because information gained about the text for one value of s is entirely ignored in considering other values of s. Such information can be very valuable, however. For example, if P = aaab and we find that s = 0 is valid (where (s) is the first index of the matching text), then none of the shifts 1, 2, or 3 are valid, since T [4] = b. In the following sections, we examine several ways to make effective use of this sort of information. The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.

Karp-Rabin algorithm

Let's view a string of k consecutive characters as representing a length-k decimal number. The character string 31415 thus corresponds to the decimal number 31,415. Given the dual interpretation of the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

String to integer converter (**Horner's rule**) [1]

Given a pattern P [1 . .m], we let p denote its corresponding decimal value. In a similar manner, given a text T [1 . . n], we let t_s denote the decimal value of the length-m substring T [s + 1 . . s + m], for s = 0, 1, . . . , n − m. Certainly, t_s = p if and only if T [s + 1 . . s + m] = P[1 . .m]; thus, s is a valid shift if and only if t_s = p.

We can compute p in time O(m) using Horner's rule

$$p = P[m] + 10\,P[m-1] + 10^2 P[m-2] + \cdots + 10^{m-1} P[2] + 10^m P[1]$$

The value t0 can be similarly computed from T [1 . . m] in time O(m).

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time O(n−m), it suffices to observe that ts+1 can be computed from t_s in constant time, since

$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$.

For example, if m = 5 and t_s = 31415, then we wish to remove the high-order digit T [s +1] = 3 and bring in the new low-order digit (suppose it is T [s +5+g1] = 2)

to obtain t_{s+1}= 10(31415 − 10000 · 3) + 2 = 14152 .

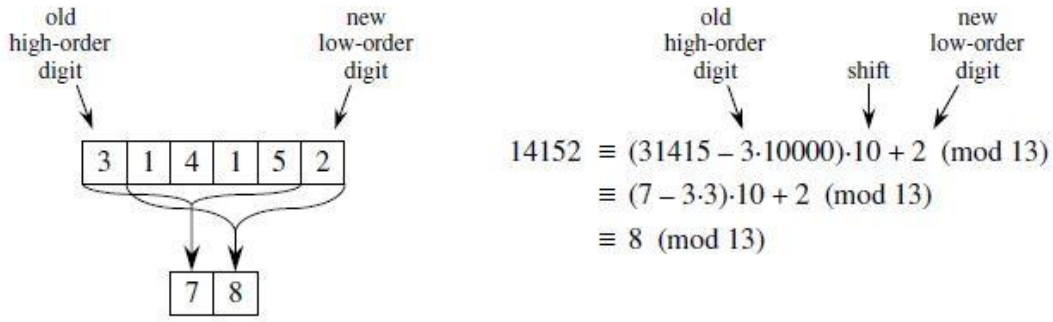

Figure 2 calculating t(s+1) using t(s)

Subtracting 10m−1T [s +1] removes the high-order digit from t_s , multiplying the

result by 10 shifts the number left one position, and adding T [s +m +1] brings in

the appropriate low-order digit. The only difficulty with this procedure is that p and t_s may be too large to work with conveniently. If P contains m characters, then assuming that each arithmetic operation on p (which is m digits long) takes "constant time" is unreasonable. Fortunately, there is a simple cure for this problem, compute p and thet_s's modulo a suitable modulus q

The modulus q is typically chosen as a prime such that 10q just fits within one computer word, which allows all the necessary computations to be performed with single-precision arithmetic. In general, with a d-array alphabet {0, 1, . . . , d − 1}, we choose q so that dq fits within a computer word and adjust the recurrence equation to work modulo q, so that it becomes

t_{s+1}= (d (t_s− T [s + 1]h) + T [s + m + 1]) mod q,

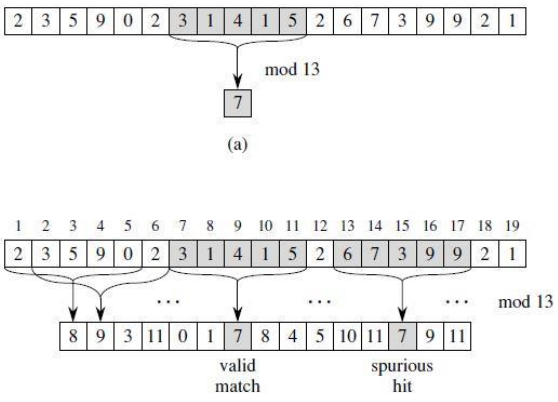

Figure 3an example of Karp-Rabin matcher and the spurios hit

The solution of working modulo q is not perfect, however, since $t_s \equiv$ p (mod q) does not imply that t_s = p. On the other hand, if $t_s \equiv$ p (mod q), then we definitely have that t_s= p, so that shift s is invalid. We can thus use the test $t_s \equiv$ p (mod q) as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv$ p (mod q) must be tested further to see if s is really valid or we just have a **spurious hit**. This testing can be done by explicitly checking the condition P[1 . .m] = T [s + 1 . . s + m]. If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

Matching steps [2]

1 n ← length[T]

2 m ← length[P]

3 h ← dm−1 mod q

4 p ← 0

5 t0 ← 0

6 **for** i ← 1 **to** m do **(Preprocessing).**

7 p ← (dp + P[i]) mod q

8 t0 ← (dt0 + T [i]) mod q end for

9 **for** s ← 0 **to** n – m do **(Matching**)

10 **if** p = t_s then

11 **if** P[1 . .m] = T [s + 1 . . s + m] then

12 print s end if end if

13 **if** s < n – m then

14 t_{s+1} ← (d(t_s − T [s + 1]h) + T [s + m + 1]) mod q end if end for

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high order digit position of an m-digit window. Lines 4–8 compute p as the value of P[1 . .m]

mod q and t0 as the value of T [1 . .m] mod q. The **for** loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant: Whenever line 10 is executed, t_s = T [s + 1 . . s + m] mod q. If p = t_s in line 10 (a "hit"), then we check to see if P[1 . . m] = T [s +1 . . s +m] in line 11 to rule out the possibility of a spurious hit. Any valid shifts found are printed out on line 12. If s < n − m (checked in line 13), then the **for** loop is to

be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of t_{s+1} mod q from the value of t_s mod q in constant time.

Complexity [1]

RABIN-KARP-MATCHER takes O(m) preprocessing time, and its matching time

is O((n − m + 1)m) in the worst case, since (like the naive string-matching algorithm)

the Rabin-Karp algorithm explicitly verifies every valid shift. If P = a^m

and T = a^n, then the verifications take time O((n − m + 1)m), since each of the

n − m + 1 possible shifts is valid.

In many applications, we expect few valid shifts (perhaps some constant c of them); in these applications, the expected matching time of the algorithm is only O((n − m + 1) + cm) = O(n+m), plus the time required to process spurious hits.

Although the O ((n − m + 1)m) worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems.

Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. Their algorithm depends on an auxiliary function called The prefix function π. It encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the naive pattern-matching algorithm

The prefix(π)function for a pattern [3]

Consider the operation of the naive string matcher. That uses a particular shift s of a template containing the pattern P = ababaca against a text T for this example, q = 5 of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift s + 1 is

necessarily invalid, since the first pattern character (a) would be aligned with a text

character that is known to match with the second pattern character (b). The shift s= s + 2 shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match.

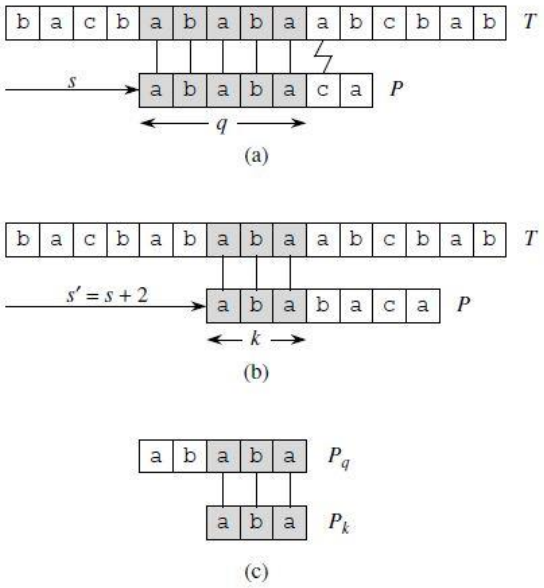

Figure 4 skipping the shifts that must necessarily match the characters of text the using the prefix compute function

In the figure below, for the pattern P = abababababca and q = 8.
(**a**) The π function for the given pattern. Since π[8] = 6, π[6] = 4, π[4] = 2, and π[2] = 0, by iterating π we obtain π∗[8] = {6, 4, 2, 0}. (**b**) We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P8; this happens for k = 6, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P8. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters.

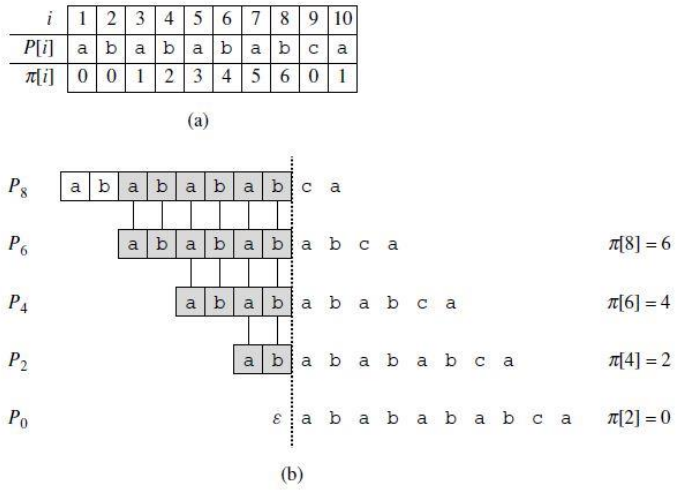

Figure 5 an example of KMP matcher

portion of the text, it is a suffix of the string P_q . Equation (32.5) can therefore be

interpreted as asking for the largest k < q Then, s'= s+(q−k) is the next potentially valid shift.

This information can be used to speed up both the naive string-matching algorithm and the finite-automaton matcher.

Matching steps [1]

KMP-MATCHER(T, P)

1. m ← length[P]
2. n ←length [T]
3. π ←COMPUTE-PREFIX-FUNCTION(p)
3. i←0, j←0
4. while(i+m<=n)do
5. while(t[i+j]==p[j])do
6. j←j+1
7. if(j>=m)
8. return i end while
9. i←i+ max(j-π [j-1],0)
10. j = π [j-1]end while
11. return -1

COMPUTE-PREFIX-FUNCTION(P)

1. m ← length[P]
2. π[1] ← 0
3. k ← 0
4. for q ← 2 to m do
5. while k > 0 and P[k + 1] ~= P[q] do
6. k ← π[k] end while
7. if P[k + 1] ==P[q] then
8. k ← k + 1 end if
9. π[q] ← k end for
10. return π

max(a,b)

1. if a>b
2. return a
3. else
4. return a

Complexity [3]

The running time of COMPUTE-PREFIX-FUNCTION is O (m).

and the average run time of the KMP matcher is O(n).

but when we search for a short pattern the KMP algorithm is not very good because it depends on the repetition of characters in the pattern and when the pattern is short the chance of repeating characters is very low.

Boyer-Moore algorithm

The Boyer-Moore algorithm is designed to skip the highest number of useless shifts using the right to left scan, the bad character rule and the good suffix rule. These three ideas can make the matching process faster and more convenient while searching in a long text because they help to skip a lot of failing matching attempts.

Right-to-left scan:[4]

Instead of scanning the pattern from the left to the right, this algorithm starts scanning from the right.

Bad character rule:[4]

we use this rule when a mismatch occurs, so we use the knowledge of the mismatched character to skip alignments.

Let character (b) be the mismatched character in text (T). so we skip alignments until (b) matches its opposite in pattern(P) or (P) moves past (b).

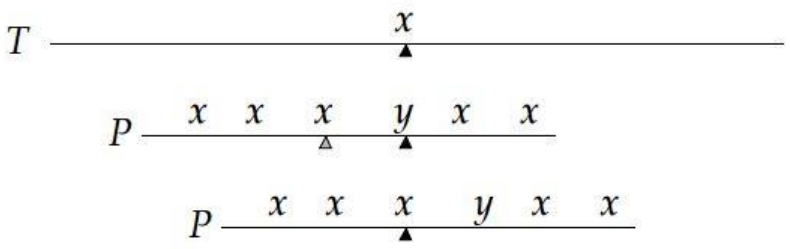

Figure 6 skipping shifts using the bad character rule

Good suffix rule:[4]

When some characters are matched, we can use the knowledge of the matched characters to skip alignments.

Suppose that for some alignment of (P) and (T), substring (t) of (T) matches a suffix of (P), but a mismatch occurs at the next position. Find the rightmost copy (t') of (t) in (P) such that (t') is not a suffix of (P) and the character to the left of (t') in (P) differs from the

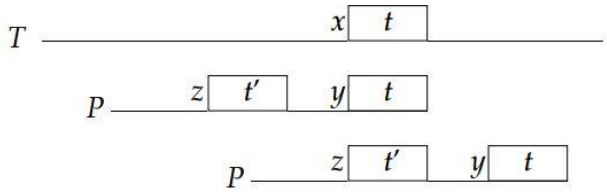

Figure 7 skipping useless shifts using the good suffix rule

character to the left of (t) in (P). Shift (P) so that (t') in (P) is aligned with(t) in (T).

If there is no such t', shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T.

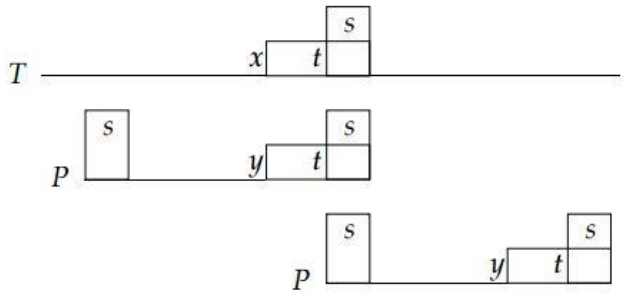

Figure 9 using the good suffix rule to shift the pattern to the right place

If no such shift is possible, shift P by n places to the right.

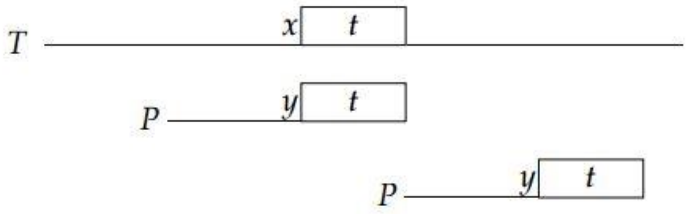

Figure 8using the good suffix rule to shift the pattern to the right place

If an occurrence of P is found, shift P by the least amount so that a proper prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift is possible, shift P by n places to the right

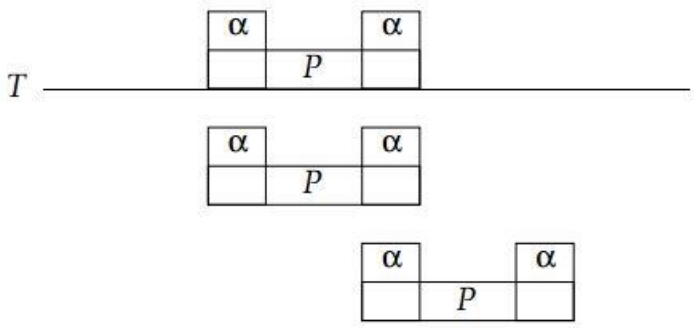

Figure 10 using the good suffix rule to shift the pattern to the right place

Putting it together [4]

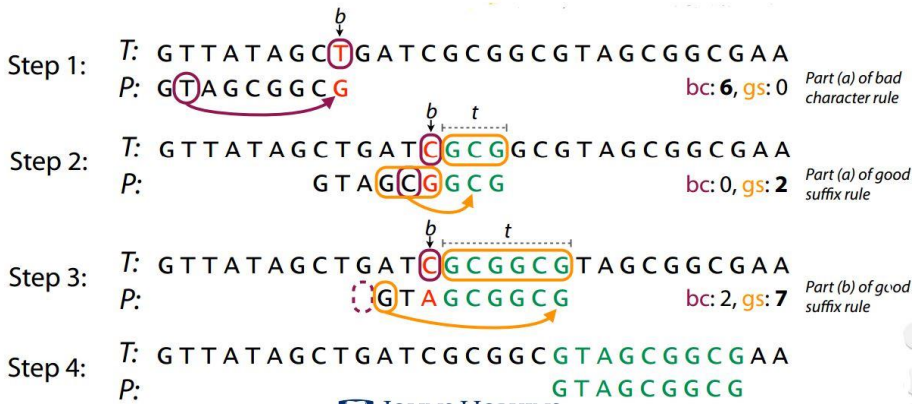

Figure 11 an example of the Boyer–Moore matcher

After each alignment, use bad character or good suffix rule, whichever skips more

Matching steps:[2]

bmInitocc()

1. char a; int j;
2. for a←0 to alphabetsize
3. occ[a]=-1;
4. for j←0 to m-1
5. do a=p[j]
6. occ[a]=j end for

bmPreprocess1()

1. i←m, j←m+1
2. f[i] ←j;
3. while i>0 do
4. while j<=m and p[i-1]~= p[j-1]) do
5. If s[j]==0
6. s[j]=j-i
7. j=f[j] end while
8. i=i-1
9. j=j-1
10. f[i]=j; end while

bmPreprocess2

1. j←f[0]
2. for i←0 to m do
3. if s[i]==0
4. then s[i] ←j
5. if i==j
6. then j=f[j]
7. end for

max(**a**,**b**)

1. if a>b
2. return a
3. else
4. return a

bmSearch()

1. i←0
2. while i<=n-m do
3. j←m-1
4. while j>=0 and p[j]==t[i+j] do
5. j--
6. if j<0 then
7. return i
8. i←i+s[0] end if
9. else
10. i←i+max(s[j+1], j-occ[t[i+j]]) end while
11. end while

complexity:{Cormen, 2002 #8}

the worst case of Boyer-Moore algorithm is O((n − m + 1)m)

but in general this algorithm has sub liner run time which is O(n/m) so we can say that it is the fastest algorithm between the ones we studied. but when we search for a short pattern the Boyer- Moore algorithm is not very good because it depends on the repetition of characters in the pattern and when the pattern is short the chance of repeating characters is very low.

Results and Conclusion

As we saw in this research, we have a lot of string matching algorithms that can do the same mission but with different specification.

The naïve algorithm is the simplest algorithm, but it takes more time than the others so it is very good in the simple applications that don't have a long pattern to search for.

But for the real applications that needs very fast processing and has a long pattern to search for in a very long text, the Boyer-Moore and the KMP algorithms are the best ones, especially if we have a small alphabet (as when we search for a pattern in a sequence of DNA the alphabet is only (G, T, C, A)) because we have a lot of repeated chars in the pattern that gives us higher chance to skip useless shifts, but when we want to search for a short pattern, the Boyer-Moore and the KMP algorithms won't be efficient enough, because their preprocessing functions depend on repeating characters in the pattern but for a short pattern the probability of repeating characters will be very low especially if we have a big alphabet. So in this case the best algorithm of the ones we study in this research is the Rabin-Karp algorithm.

The algorithm	Preprocessing time	matching time
Naïve	0	$O((n - m + 1)m)$
Rabin-Karp	$O(m)$	Worst case $O((n - m + 1)m)$ Best case $O(n)$
KMP	$O(m)$	$O(n)$
Boyer Moore	$O(m)$	Worst case $O((n - m + 1)m)$ best case $O(n/m)$

References

1. Cormen, T.H., C. E.leiserson, and C. Stein, *String Matching*, in *Introduction To Algorithms*. 2002, The MIT Press. p. 906-933.
2. Crochemore, M., W. Rytter, and M. Crochemore, *Text algorithms*. Vol. 698. 1994: World Scientific.
3. Aho, A.V. and M.J. Corasick, *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 1975. **18**(6): p. 333-340.
4. Boyer, R.S. and J.S. Moore, *A fast string searching algorithm.* Communications of the ACM, 1977. **20**(10): p. 762-772.