ANALYSIS OF THE KMP ALGORITHM

BY:-

KRITIKA RAJ (180123024)

PRAGATI MAHAMUNE (180123032)

I. AIM:

-To present and discuss the Knuth-Morris-Pratt (KMP) algorithm, its optimization and other related work.

II. BACKGROUND

1. Introduction

Pattern matching is one of the basic operations of strings. To illustrate, assuming that pattern P is a given substring and text T is a much longer string to be looked up, it is required to find all substrings that are the same as P in string T.

Under normal circumstances, when talking about pattern matching, the first attempt is to match the substring P with string T starting from every single element in T, which uses the brute force method. By way of illustration, if we use f (T) for the length of T, supposing f (T) =n and f (P) =m, "length" means the number of all the letters in a string, when searching P from T, the number of letter matching operations it will take is to be m*(n-m+1). This is the naïve string-matching algorithm. Other algorithms for pattern searching include the Rabin-Karp algorithm, String matching with finite automata and the KMP algorithm. Now, we thoroughly look into the KMP or Knuth-Morris-Pratt algorithm.

2. State of art of the KMP algorithm

2.1 The Original KMP algorithm

KMP algorithm was invented by Donald Knuth and Vaughan Pratt together and independently by James H Morris in the year 1970. In the year 1977, all the three jointly published KMP Algorithm. The Knuth-Morris-Pratt (KMP) algorithm is one of the most efficient string-matching algorithms in theory. It has been considered as the first linear time string-matching algorithm with:

Time Complexity: O (m+n)

Preprocessing time: O (m)

Preprocessing algorithm of KMP- The KMP algorithm traverses the given pattern string from head to tail, trying to find the longest common elements between the prefix and the suffix of each substring of the pattern, and take down the length of the common part in a "Failure Table", and the table should be of the same length to the pattern. Each letter of the pattern in the Failure Table has a corresponding number to be calculated.

Searching algorithm of KMP- Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, in KMP we use a value from 'Failure Table' to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

2.2 An Example to illustrate the KMP Algorithm

Definition 1: The side of a string or a word w, on a q-ary alphabet is a non-empty subword which is a (strict) right factor and a (strict) left factor of w.

Example (a): 01201201 have two different sides: 01 and 01201.

In the following, P is a pattern that is searched in a text T. We note P[i] (resp. T[i] the i-th character of the pattern P (resp. The text T). The algorithm uses a text pointer tp and a pattern pointer pp that determine the next comparison to be performed: if tp =i and pp = j , then the i-th character of the text is to be compared to the j-th character of the pattern. After a comparison, the pointers are updated, as indicated by the result. After a match, both pointers are moved one step forward. Note that if pp=|P|=m, the pattern has been found, and pp will point again to the first character of the pattern. If a mismatch occurs, two choices appear. If pp=1, the text pointer is moved forward. In other cases, the text pointer is not updated. Never having to read backward is an advantage of the Knuth-Morris-Pratt algorithm over the naïve algorithm. To update the pattern pointer pp, a function next is defined. For any given j, let the largest side of the substring P[1]……..P[j-1] already read be P[1]………P[k-1]. Then next[j] =k and pp is moved to k. Remark that whenever the value of pp is 1, pp remains equal to 1 after a mismatch.

Example (b): Let P = 012012123 be the pattern and T = 32012012012123321 be the text. After the first comparison (a mismatch between T[1] = 3 and P[1] =0), the text pointer is moved to 2, compared to 0 again. After the next move of tp to T[3] = 0, six matches occur. The string 012012 is recognized, pp points to P[7] = 1 and tp to T[9] = 0. After this mismatch, the longest side of P[1]……P[6] = 012012 is 012, then pp is moved to 4. Now, T[9] matches to P[4] and the whole pattern is found.

2.3 Limitation of KMP

Although this algorithm is relatively outstanding, it still has a considerable margin of improvement. As the pattern slides forward, there are still many matches that are not necessary. In addition, when the pattern first appears in the second part of the text, more comparisons will be needed.

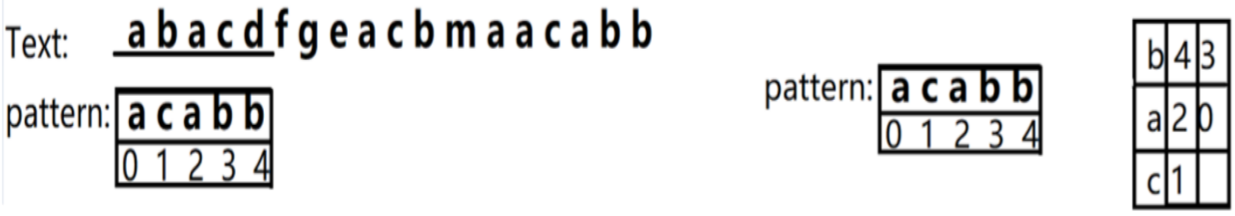
III. WORK FOCUSED

3. Proposed algorithm- L-I-KMP algorithm

This method uses a letter numbered table and a new data structure called last-identical array to decide the length of the move in some matching occasions. Similar to the KMP algorithm, the process is continued by referring to the data in the table when facing mismatches.

3.1. Preprocessing for L-I-KMP algorithm

Preprocessing is needed for this method. Before the match starts, it is necessary to analyze the data of the pattern, a letter table is used to take down the letters and their position in the pattern, and the next same letter from the data structure can be directly queried according to the appearing number. The order of the position starts from the tail of the pattern. This idea is being illustrated in more detail in Figure 1. In Figure 1.a), the given pattern is “acabb”, and the target text has been listed in the picture. It enables to browse from right to left and record each new character and its position during the process of preprocessing; this process is displayed in Figure 1.b).

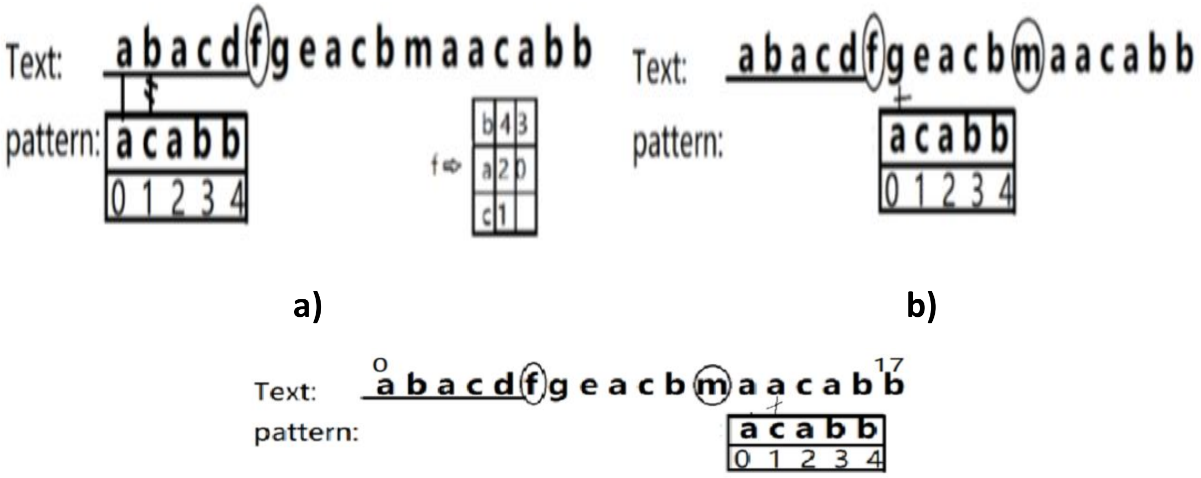


a) a pair of strings given **b) make a letter table when preprocessing.**

Figure 1. Example of L-I-KMP's preprocessing.

3.2. The flow of the algorithm

In L-I-KMP, the next step after mismatching of a character is to focus on the next digit of the last bit aligned with the entire pattern string, which is A1 in this context. If A1 is not in the letter table, the amount of movement is the length of itself plus one. Otherwise, looking up the letter table to memorize the first number after A1, and this number is k, corresponds to the position of one same character in the pattern, call it A2. Align the two characters, and start comparison from the left of the pattern. When mismatches occur, refer to the table again to locate the next number after k, and align the new location of the pattern with A1. The same logic continues until there is no corresponding number for A1. The next bit of the last aligned bit can be found at this point, and thereby to name a new A1. Figure 2 interprets the algorithm flow by using the same example as Figure 1.



a) **b)**

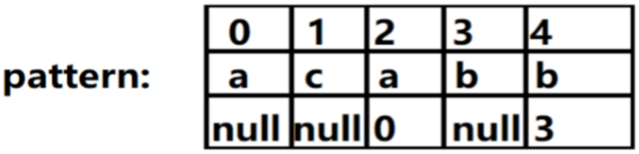
c)

Figure 2. Example to show the algorithm flow

In Figure 2.a), after preprocessing, the first step matches. However, b ≠ c, so pattern [1] ≠ text [1], the next digit we need to consider is the sixth letter text [5] = f, while f does not exist in the pattern, the pattern can jump over “f”, the result is in Figure 2.b). Then it is obvious that, g ≠ a, text [6] ≠ pattern [0], and text [6 + length of pattern] = text [11] = m is checked, and m does not appear in the letter table either. The whole pattern skips m. In Figure 2.c), the pattern and the text does not match at pattern [1], and the last digit, text [12 + length of pattern] = text [17] = b, of the text is one component of the letter table. The numbers that follow it are 4 and 3 in order. The initiate activity is to drag the pattern and make pattern [4] align with the text [17], and then begin comparing from the left of the pattern. Fortunately, the whole pattern matched. Thus, there is no need to use the number “3” of the letter table any longer.

3.3. Optimization for coding convenience

In the last-identical array, the character in the pattern not only corresponds to its position number but also points to the previous position of the same letter, if this letter appears the first time from left to right, then it points NULL. In Figure 3, given the pattern “acabb” and its position array in order is “01234”, its last-identical array is to be “NULL NULL 0 NULL 3”.

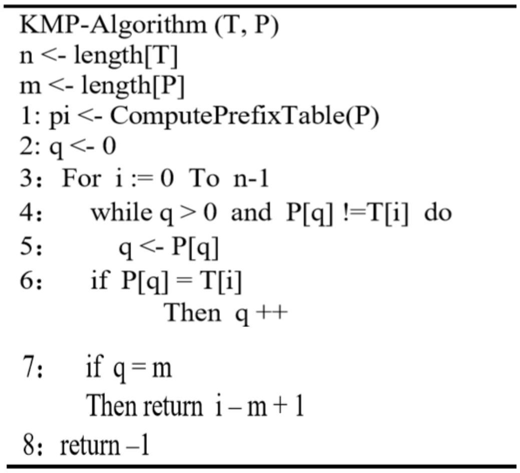


	0	1	2	3	4
pattern:	a	c	a	b	b
	null	null	0	null	3

Figure 3. The last-identical array of the given pattern

4. Performance Comparison of Algorithms

The times of attempt and the amount of letter’s match during the whole matching procedure are two essential elements to analyze the capability of the matching algorithm [16]. The pseudo-code of the KMP algorithm is expressed in Figure 4.

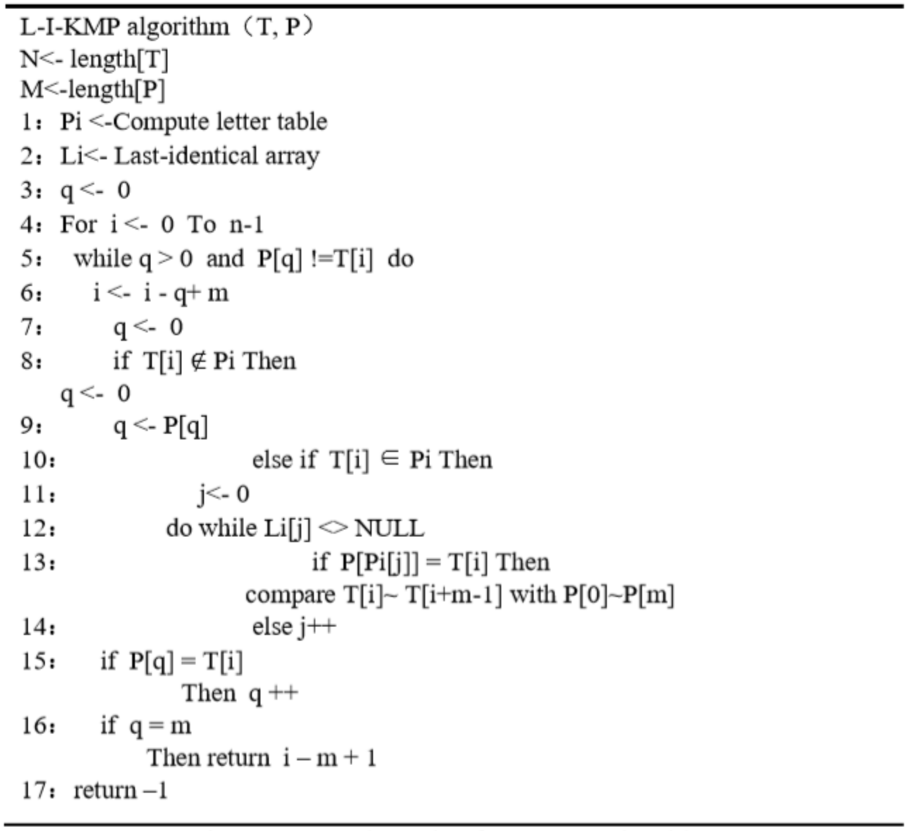


```
KMP-Algorithm (T, P)
n <- length[T]
m <- length[P]
pi <- ComputePrefixTable(P)
q <- 0
For i := 0 To n-1
    while q > 0 and P[q] !=T[i] do
        q <- P[q]
    if P[q] = T[i]
            Then q ++

    if q = m
    Then return i – m + 1
return –1
```

Figure 4. KMP Algorithm

When using the KMP algorithm, the first step is to preprocess the Prefix-Table of P, that is, to calculate the “Failure table” or the “next array”. Then step (2) initializes q to be the number of characters that matched, at the beginning of q=0. In step (3), it is time to start scanning the text from left to right. Then in step (4) (5) (6), the algorithm starts matching, if the two characters qualify the equation, the number of matched letters—q, is to plus one. Or if the two does not match, assign p a new value: q=P [q]. The pseudo-code of the new algorithm is displayed in Figure 5.

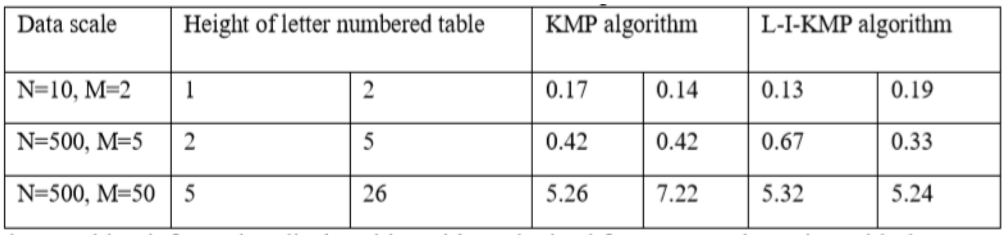


```
L-I-KMP algorithm（T, P）
N<- length[T]
M<-length[P]
1： Pi <-Compute letter table
2： Li<- Last-identical array
3： q <-  0
4： For  i <-  0  To  n-1
5：   while q > 0  and  P[q] !=T[i]  do
6：      i <-  i - q+ m
7：         q <-  0
8：         if  T[i] ∉ Pi Then
     q <-  0
9：         q <- P[q]
10：                      else if  T[i] ∈ Pi Then
11：                   j<- 0
12：              do while Li[j] <> NULL
13：                               if  P[Pi[j]] = T[i] Then
                         compare T[i]~ T[i+m-1] with P[0]~P[m]
14：                      else j++
15：     if  P[q] = T[i]
                   Then  q ++
16：     if  q = m
              Then return  i – m + 1
17： return –1
```

Figure 5. Pseudo-code of L-I-KMP algorithm

The table below illustrates the performance comparisons between the L-IKMP and KMP algorithm, in matching time.

Table 1. Performance comparison.



Data scale	Height of letter numbered table		KMP algorithm		L-I-KMP algorithm	
N=10, M=2	1	2	0.17	0.14	0.13	0.19
N=500, M=5	2	5	0.42	0.42	0.67	0.33
N=500, M=50	5	26	5.26	7.22	5.32	5.24

The matching information displayed in Table 1 obtained from repeated running with the same data generated randomly. It is noticeable that L-I-KMP algorithm performs well in practice when the types of letters are only a few in the pattern, and the same letters are further apart. If the pattern is not so long, it can achieve a good speed similar to the original KMP algorithm. Therefore, this is an appropriate algorithm.

IV. CONCLUSION

Through the experimental demonstration of string-matching processing, it is not uneasy to discover that the efficiency of KMP and L-I KMP algorithms are almost equal, assuming that the amount of data is small. However, when the data set is large, and the number of types of the pattern is relatively big, or the number is small but unevenly distributed, the L-I KMP algorithm is superior to the KMP algorithm. This study confirms that string matching problems can be solved in practical applications. Additionally, the new algorithm can be improved in many aspects by optimizing the last-identical array or completing the comparing process with the letter numbered table.

V. REFERENCES:

1] The Analysis of KMP Algorithm and its Optimization, Article by Xiangyu Lu 2019 J. Phys: Conf. Ser. 1345 042005.
https://www.researchgate.net/publication/337595769_The_Analysis_of_KMP_Algorithm_and_its_Optimization

2] Knuth-Morris-Pratt Algorithm: An Analysis, Conference Paper: August 1989 by Mireille Regnier

https://www.researchgate.net/publication/220975322_Knuth-Morris-Pratt_Algorithm_An_Analysis

3] Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.