

KMP ALGORITHM FOR SEARCHING TOOLS IN ANY APPLICATION

Brandon Oktavian Pardede/13518043
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): brandonpardede25@gmail.com

Abstract—Searching tools have contributed so much in helping user to find certain words that would like to be found in a web page or application. It also return all matching words in the page on relatively very fast time, in fact less than a second. The algorithm used for this tools are vary, but here explained all we need to know about what's behind the searching tools, especially the KMP Algorithm.

Keywords—KMP Algorithm, Searching Tools, Technology

I. INTRODUCTION

Have you ever wonder how the searching tools work in browser? Or do you ever wonder how can small computers that we use in today's life can process things so fast that even we can't do it as fast as they do? That's the magic of the technology, the way they solve problems that given to them can even outcome our abilities to do it.

Searching tools in web browser or any other applications is one of the example. They can process our input so fast and return all the possible matching word in just a second. But how do they process it? The answer is there must be an algorithm behind it, the algorithm that is best suit for matching characters in a big data text that contains so many words in it.

In the world of science computing, there are many algorithm that can be used to matching strings with pattern that we would like to match, such as Booyer-Moore Algorithm (BM), Brute Force Algorithm, and Knuth-Morris-Pratt Algorithm (KMP). These three algorithm vary in method to match strings, as well as the time complexity that have to be taken in order to process the input until the output. They also have their own advantages and disadvantages, and thus make them can only be suitable for certain problem-solving that requires or still in their own boundaries.

In this problem we discuss, we will dive in to the world of KMP Algorithm and explains why this algorithm is used in so many field of string matching, and why searching tools use this algorithm to help them find the matching strings in just a blink of a second.

II. THE KNUTH-MORRIS-PRATH ALGORITHM (KMP)

The Knuth-Morris Prath Algorithm (KMP) is an algorithm for string matching that use *lps* (longest prefix suffix) as the function boundaries. This algorithm match strings from the left of the text, and then process the mismatch with the algorithm as below:

whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

When processing the whole text with a pattern that we would like to search the match, the first thing to do is find the *lps* for each index of the pattern. For example:

Suppose we have a pattern that consists of "ABCDE". We divide this pattern to each index, so the boundaries (mark as $b(k)$) would be like this:

j	0	1	2	3	4
P[j]	A	B	C	D	E
k	-	0	1	2	3
b(k)	-	0	0	0	0

The word "j" is the index of the pattern starting from 0, the word "k" is the value of j subtracted by 1, and $b(k)$ is the *lps* (longest prefix suffix) from the index (0...k) for the prefix and index (1...k) for the suffix. From the word "ABCDE", we know that there are no prefix that also suffix, so the value of all of the boundaries function for each index is 0.

The next step is for the searching algorithm. After we found all of the boundaries function value of each index, we will continue to processing all the text with the pattern we would like to match. We use a value from $b(k)$ to decide the next characters to be matched. The idea is to not match a character that we know will anyway match. How to use $b(k)$ to decide next positions (or to know a number of characters to be skipped)? Here is the step by step:

1. We start comparison of pat[j] with j = 0 with characters of current window of text.
2. We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
3. When we see a **mismatch**:
 - We know that characters pat[0..j-1] match with txt[i-j...i-1] (Note that j starts with 0 and increment it only when there is a match).
 - We also know (from above definition) that lps[j-1] is count of characters of pat[0...j-1] that are both proper prefix and suffix.
 - From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j...i-1] because we know that these characters will anyway match.

Here below is shown the implementation of KMP Algorithm (source code) of the algorithm in Java, this source code is taken from geeksforgeeks.com and contributed by Amit Khandewall.

```
void computeLPSArray(char* pat, int M, int* lps);
```

```
// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);
```

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];
            }
        }
    }
}
```

```

// Also, note that we do not increment
// i here
}
else // if (len == 0)
{
lps[i] = 0;
i++;
}
}
}
}

// Driver program to test above function
int main()
{
char txt[] = "ABABDABACDABABCABAB";
char pat[] = "ABABCABAB";
KMPSearch(pat, txt);
return 0;
}

```

The function “computeLPSArray” is used to defined the boundaries function of each index of the pattern. It is the same as b(k) function that writer have explained before. So assume we have array of LPS that contains [0,0,0,1],so the boundaries function(b(k)) of each index(starting from 0) is respectively 0,0,0,1.

Example of Knuth-Morris-Pratt(KMP) Algorithm using simple text:

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
k	-	0	1	2	3	4
b(k)	-	0	0	1	0	1

The first step is we have to find the boundaries function of each index of the pattern. The pattern that we would like to search in this case is “abacab”. From the left-down side of the picture above,we can see the boundaries function of the pattern. Here is the step by step to determine the boundaries function(b(k)):

At pattern index 0, the boundaries function is not defined since there are no k value that is -1,thus the b(k) is not defined and we can apply the Brute-Force algorithm when we found the mismatch at pattern index 0.

Next,at pattern index 1,the boundaries function for k == 0 is 0 because there are no longest prefix suffix between “a” (prefix) and not defined suffix.

At pattern index 2,the boundaries function for k == 1 is 0 since there are no longest prefix suffix between “ab” and “b”.

At pattern index 3,the boundaries function for k == 2 is 1 since the longest prefix suffix between “aba” and “ba” is “a” where the length is 1.

At pattern index 4,the boundaries function for k == 3 is 0 since there are no longest prefix suffix between “abac” and “bac”.

Last but not least,at pattern index 5,the boundaries function for k == 4 is 1 since the longest prefix suffix between “abaca” and “baca” is 1.

After we found the boundaries function for each index of the pattern,now we process to the searching.

The first mismatch occurs at pattern index 5,where the boundaries function is 1. So,we start the next comparison from

pattern index 1, which means we do not have to check the pattern index 0 again (see the picture above).

The mismatch occurs again when we comparing the pattern[1] with the text index before. Because the value of boundaries function of $k = 0$ is 0, so we proceed to compare the pattern index 0.

The mismatch occurs again at pattern[4], so we check the $b(k)$ which is 0. We compare the pattern[0], but the mismatch immediately occurs, it means we can apply the Brute Force Algorithm because the mismatch occurs and boundaries function is not defined.

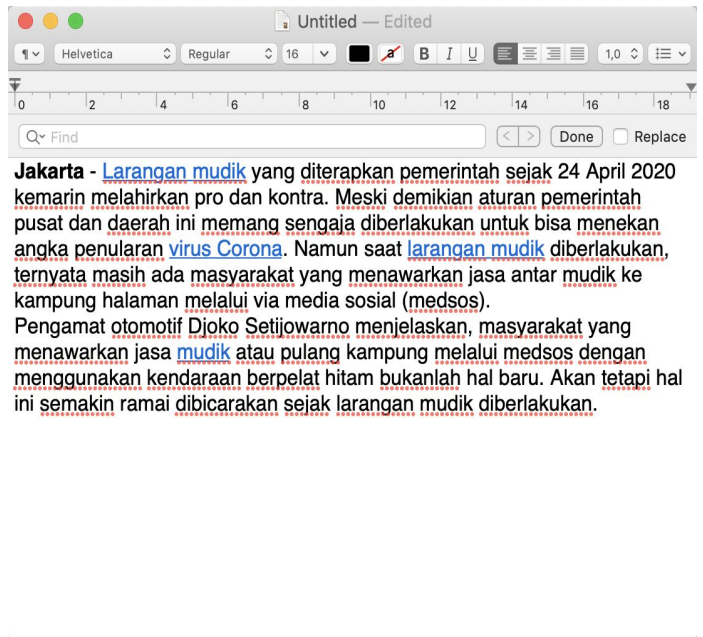
Finally, all the words are matching so we can get the result, where the match occurs from index 10.

The time complexity for Knuth-Morris-Pratt (KMP) Algorithm is $O(n)$ where n is the text size.

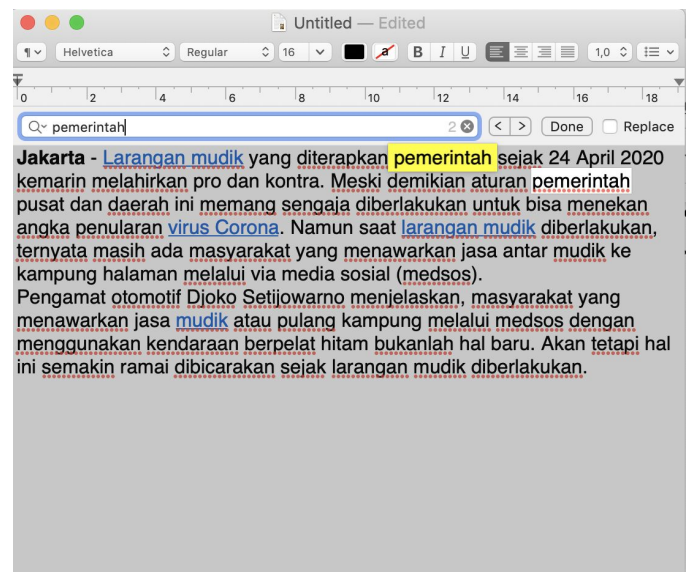
III. SEARCHING TOOLS APPLICATION

Jumping to the real problem, now we try to apply these algorithm to find the matching string with the pattern we would like to put. To be noted, the application of this problem may vary in many ways, because there are many words that can be put as a pattern, as well as there are also many source text that can be used for the test case, so we just explain one of the examples regarding this.

Suppose we have a window that consists of large text. Now we have an input of pattern that we would like to match with the whole text in the window. In this example, the application used would be TextEditor in MacOS.



The picture above is the text that we would like to match with the pattern. As shown above, the pattern is not yet input, so there will be no return match. If we try to input something to the "find" table in the TextEdit, the TextEditor will bold certain words that match with the input we put in the "find" table. Here is the example:



When words “pemerintah” is put in the “find” table,the TextEditor application will bold certain words that match the input text we put in the “find” table before. This apply the KMP Algorithm that have explained before,where the application check all the words in the file,then return words or index that match with the input string.

Now,we try to break it down from the beginning.Using the KMP Algorithm,we create a boundaries function for the pattern we input,in this case is “pemerintah”

```

j   0 1 2 3 4 5 6 7 8 9
P[j] p e m e r i n t a h
k   - 0 1 2 3 4 5 6 7 8
b(k) - 0 0 0 0 0 0 0 0 0

```

We can see that the boundaries function of each index of the pattern is 0,so when we proceed to the next searching step,when we found a mismatch in the text,it will only continue to the next index of text where the mismatch occurs.Example:

In the index 0 where the text[0] is “J” and pattern[0] is “p”,the mismatch occurs,because the mismatch occurs at pattern index 0,the boundaries function is not defined,so we use the brute-force algorithm-like for processing the next word.Now we compare text[1] with pattern[0], the mismatch occurs again at index 0 in the pattern,so we check again the boundaries function(which is not defined for -1),we apply like the previous way. We process to the next index until we found “p” at text index 36(in word “diterapkan”),the mismatch don’t occur,so we check next text index which is 37(“k”),and compare to pattern[1],where mismatch occurs. Because the mismatch occurs,so we start again from index 37,and so on until we find the matching words in index 41. The KMP Algorithm also search for the next words even after the matching patterns occur,basically saying that they check all the index in the whole text,so the return result can be more than 1 matching words.

When we found boundaries function of a pattern where the boundaries function of each index is not 0 at all,so we can easily apply this algorithm more effectively,because the algorithm do not check again the word that do not need to be checked. Example when we have pattern “abaaba”,the b(k) when k == 4 is 2. So when the mismatch occurs at pattern[5] which the value of k is 4,we start again the next comparison from pattern index 2(the value of boundaries function),this

means the words before pattern[2] do not need to be check again because it must be the same words.

IV. CONCLUSION

The Knuth-Morris-Pratt Algorithm is very effective in processing string-matching problems.

The pre-processing(finding boundaries function) have time complexity of O(n) and the searching time complexity of O(m) where n is the pattern size and m is the text size.

When using KMP Algorithm,we do not have to be worry because of the text size which can be very large and big. This algorithm have guaranteed that the worst case is also efficient (O(m) also),compared to other algorithm(Brute Force,Booyer-Moore,and Rabin-Karp) where it takes O(mn) to process in the worst case.

VIDEO LINK AT YOUTUBE (*Heading 5*)

<https://www.youtube.com/watch?v=7VsX4AQYJSs>

ACKNOWLEDGMENT

The author would like first to give thanks to God because without Him,author would not even be able to finish this project and also have the chance to still live.Author would also like to give the biggest appreciation and respect to all of the people who contributed from the beginning of the Strategi Algoritma study progress until now.To Dr. Ir. Rinaldi Munir, MT and Dr. Masayu Leylia Khodra,author would like to give the biggest appreciation and respect because Mr.Rinaldi have been a good and caring teacher,which makes author enjoy the studying progress and learn so much from their teaching. Author also would like to thank his parents,that have provided the best life and support while doing his academic progress in Bandung Institute of Technology.

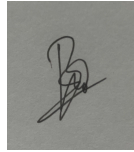
REFERENCES

- [1] [http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Pencocokan-String-\(2018\).pdf](http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Pencocokan-String-(2018).pdf)
- [2] https://oto.detik.com/berita/d-4999554/jasa-antar-mudik-via-medsos-gunakan-mobil-pelat-hitam-sudah-berlangsung-lama?tag_from=wp_beritautama&_ga=2.151850105.1722798357.1588406165-865129897.1587489839
- [3] <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bekasi, 2 Mei 2020

A square box containing a handwritten signature in black ink. The signature is stylized and appears to be the name 'Brandon Oktavian Pardede'.

Brandon Oktavian Pardede