Department of
Computer Science and Engineering

Title: Implement String and Pattern Matching Problems using KMP Algorithm

Algorithms Lab

CSE 206

Green University of Bangladesh

1 Objective(s)

- To understand the concepts of string and pattern matching.
- To use KMP algorithm and its components to implement and analyse string and pattern matching problems.

2 Problem analysis

Given a text **txt[0...n-1]** and a pattern **pat[0...m-1]**, a function **search(char pat[], char txt[])** prints all occurrences of **pat[]** in **txt[]**. It is always true that $n > m$. Example input outputs of pattern matching algorithm are given below:

- **Input:** txt[] = "THIS IS A TEST TEXT" pat[] = "TEST"
  Output: Pattern found at index 10
- **Input:** txt[] = "AABAACAADAABAABA" pat[] = "AABA"
  Output: Pattern found at index 0
  Pattern found at index 9
  Pattern found at index 12

2.1 Discussions and simulation steps

2.1.1 KMP Algorithm

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to **O(n)**. The basic idea behind KMP algorithm is: 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.

Let us consider below example to understand the solution steps of string matching using backtracking and KMP.

1. Input **txt[]** = AAAAABAAABA and **pat[]** = AAAA
2. We compare first window of txt with pat
   txt = AAAAABAAABA
   pat = AAAA [Initial position]
   We find a match. This is a simple naïve approach.
3. In the next step, we compare next window of txt with pat.
   txt = AAAAABAAABA
   pat = AAAA [Pattern shifted one position]
   This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

2.1.2 Prefix function, lps

From the above steps, a question might be raised that how to know how many characters are to be skipped during a second trial of matching or not with the **txt[]**. KMP algorithm achieves this using a pre-processing step, by constructing a separate function **lps[]**, longest proper prefix which is also a suffix. This function is also simply known as the prefix function. A proper prefix is prefix with whole string not allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC". We search for **lps** in sub-patterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.

For each sub-pattern **pat[0..i]** where $i = 0$ to $m - 1$, **lps[i]** stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern **pat[0..i]**. Few examples of **lps[]** for given patterns are given below:

- for **pat[]** = AAAA, **lps[]** = [0, 1, 2, 3]
- for **pat[]** = ABCDE, **lps[]** = [0, 0, 0, 0, 0]
- for **pat[]** = AABAACAABAA, **lps[]** = [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
- for **pat[]** = AAACAAAAAC, **lps[]** = [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

We now simulate an example of the construction of **lps[]** or prefix function pre-processing with a given pattern below.

1. **pat[]** = AAACAAAA
2. len = 0, i = 0.
   lps[0] is always 0, we move to i = 1
3. len = 0, i = 1.
   Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 1, **lps[1] = 1**, i = 2
4. len = 1, i = 2.
   Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 2, **lps[2] = 2**, i = 3
5. len = 2, i = 3.
   Since pat[len] and pat[i] do not match, and len > 0, set **len = lps[len-1] = lps[1] = 1**
6. len = 1, i = 3.
   Since pat[len] and pat[i] do not match and len > 0, **len = lps[len-1] = lps[0] = 0**
7. len = 0, i = 3.
   Since pat[len] and pat[i] do not match and len = 0, Set **lps[3] = 0** and i = 4.
8. len = 0, i = 4.
   Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 1, **lps[4] = 1**, i = 5
9. len = 1, i = 5.
   Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 2, **lps[5] = 2**, i = 6
10. len = 2, i = 6.
    Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 3, **lps[6] = 3**, i = 7
11. len = 3, i = 7.
    Since pat[len] and pat[i] do not match and len > 0, set **len = lps[len-1] = lps[2] = 2**
12. len = 2, i = 7.
    Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. len = 3, **lps[7] = 3**, i = 8
13. We stop here as we have constructed the whole lps[]. so the final **lps[] = [0, 1, 2, 0, 1, 2, 3, 3]**

2.1.3 Time Complexity

Time complexity of the **lps[]** construction is $\mathbf{O(m)}$. Time complexity of the KMP function is $\mathbf{O(n)}$. So entire together is $\mathbf{O(n + m)}$. Since we always know $n > m$, so final time complexity of KMP can be termed as $\mathbf{O(n)}$.

3 Algorithm

Unlike simple search algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from **lps[]** to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

Algorithm 1: KMP algorithm using **lps[]** and backtracking

Input: the text, **txt[]**, the pattern **pat[]**

`/* KMP (string txt[], pattern pat[]) */`

1 Compare **pat[j]** with $j = 0$ with characters of current window of **txt[]**
2 Matching characters **txt[i]** and **pat[j]**
3 Keep incrementing i and j while **pat[j]** and **txt[i]** keep matching.
4 **if** *we see a mismatch* **then**
5 | We already know 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).
6 | We know that **lps[j-1]** is count of characters of **pat[0, ..., j-1]** that are both proper prefix and suffix
7 | We 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.
8 **end**

4 Implementation in Java

```
1  // JAVA program for implementation of KMP pattern
2  // searching algorithm
3
4  class KMP_String_Matching {
5    void KMPSearch(String pat, String txt)
6    {
7      int M = pat.length();
8      int N = txt.length();
9
10     // create lps[] that will hold the longest
11     // prefix suffix values for pattern
12     int lps[] = new int[M];
13     int j = 0; // index for pat[]
14
15     // Preprocess the pattern (calculate lps[]
16     // array)
17     computeLPSArray(pat, M, lps);
18
19     int i = 0; // index for txt[]
20     while (i < N) {
21       if (pat.charAt(j) == txt.charAt(i)) {
22         j++;
23         i++;
24       }
25       if (j == M) {
26         System.out.println("Found pattern "
27             + "at index " + (i - j));
28         j = lps[j - 1];
29       }
30
31       // mismatch after j matches
32       else if (i < N && pat.charAt(j) != txt.charAt(i)) {
33       // Do not match lps[0..lps[j-1]] characters,
34       // they will match anyway
35         if (j != 0)
```

```
36            j = lps[j - 1];
37          else
38            i = i + 1;
39        }
40      }
41    }
42
43    void computeLPSArray(String pat, int M, int lps[])
44    {
45      // length of the previous longest prefix suffix
46      int len = 0;
47      int i = 1;
48      lps[0] = 0; // lps[0] is always 0
49
50      // the loop calculates lps[i] for i = 1 to M-1
51      while (i < M) {
52        if (pat.charAt(i) == pat.charAt(len)) {
53          len++;
54          lps[i] = len;
55          i++;
56        }
57        else // (pat[i] != pat[len])
58        {
59          // This is tricky. Consider the example.
60          // AAACAAAA and i = 7. The idea is similar
61          // to search step.
62          if (len != 0) {
63            len = lps[len - 1];
64
65          // Also, note that we do not increment
66          // i here
67          }
68          else // if (len == 0)
69          {
70            lps[i] = len;
71            i++;
72          }
73        }
74      }
75    }
76
77    // Driver program to test above function
78    public static void main(String args[])
79    {
80      String txt = "ABABDABACDABABCABAB";
81      String pat = "ABABCABAB";
82      new KMP_String_Matching().KMPSearch(pat, txt);
83    }
84  }
```

5 Sample Input/Output (Compilation, Debugging & Testing)

Input:
AABAACAADAABAABA
AABA
(The entire string and the pattern to be matched are the inputs.)
Output:

Found pattern at index 0
Found pattern at index 9
Found pattern at index 12
(Output is the positions of the main string where the pattern is matched.)

6 Discussion & Conclusion

Based on the focused objective(s) to understand the backtracking solution of string matching problem using KMP algorithm, the additional lab exercise will increase confidence towards the fulfilment of the objectives(s).

7 Lab Task (Please implement yourself and show the output to the instructor)

1. Write a Java implementation using KMP algorithm to find out the indexes where a given pattern gets matched to a given array. Repeat the algorithm for multiple pairs of given strings and patterns.

8 Lab Exercise (Submit as a report)

- Implement KMP in case of integers or others

9 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.