Fundamental Algorithms, String Algorithms, HW - 9

Tulsi Jain

1. Different Pattern

Let's say length of Pattern, P is m
Given all characters in p are different.
Algorithms would be as following
First, we would iterate the text, T and look for index, i in T such that it matches the first character of P. then we would increase both T and P index by one if it matches we would repeat this step until we complete and if it did not matches then we would set P's index to first element and T to next index. In the question, it is mentioned that characters of P are different hence when we found a mismatch we do not have to start from very next element of which P's first index matches. As characters occurred in between can not first character of P.

Algorithm 1 DifferentPattern

```
function DifferentPattern(P, T)                ▷ where P pattern, T text
    m = P.length;
    n = T.length;
    matchCount = 0;
    j = 1;
    for i = 1 to n do
        if T[i] == P[j] then
            j++;
            if j == m+1 then
                matchCount++;
                Match found at shift i - m
                j = 1;
            end if
        else
            j = 1;
        end if
        i++;
    end for
end function
```

2. Gap Character

First step: We would split the Pattern, P with gap character. Let's say of gap character is n. then we would get n + 1 smaller pattern with no occurrence on gap character.
Now, we would try to find first occurrence of first pattern (out of n + 1)in text, T. If this is match then we would move to match next pattern without starting from beginning in Text, T. Doing this, in single iteration of text, T we can find the first occurrence of the pattern, P. Total running time of this algorithms is O(mn). Where n is the length of text, T and m is the length of pattern, P after removing gap characters.

Algorithm 2 GapCharacter

```
function GapCharacter(T, P)                          ▷ where T text, P pattern
    SubPattern [] = P.split("<>");
    j = 0;
    n = T.length;
    for i = 1 to n do
        if  j == SubPattern.length  then
            return true;
        end if
        if SubPattern[j][0] == T[i] then
            try matching SubPattern[j] in from T[i+1]
            k = SubPattern[j].length - 1;
            if SubPattern[j] == T[i..i + k]  then
                j = j + 1;
                i = i + k;
            end if
        end if
    end for
    return false;
end function
```

3. Working Modulo

T = 3141592653589793, q = 11, P = 26:

Pattern	Mod
31	9
14	3
41	8
15	4
59	4
92	4
26	4
65	10
53	9
35	2
58	3
89	1
97	9
79	2
93	5

Spurious hit is = 15, 59, 92

26 is correct

4. String-matching Automaton

P = aabab, T = aaababaabaababaab State-transition

State	a	b
0	1	0
1	2	0
2	2	3
3	4	0
4	2	5
5	1	0

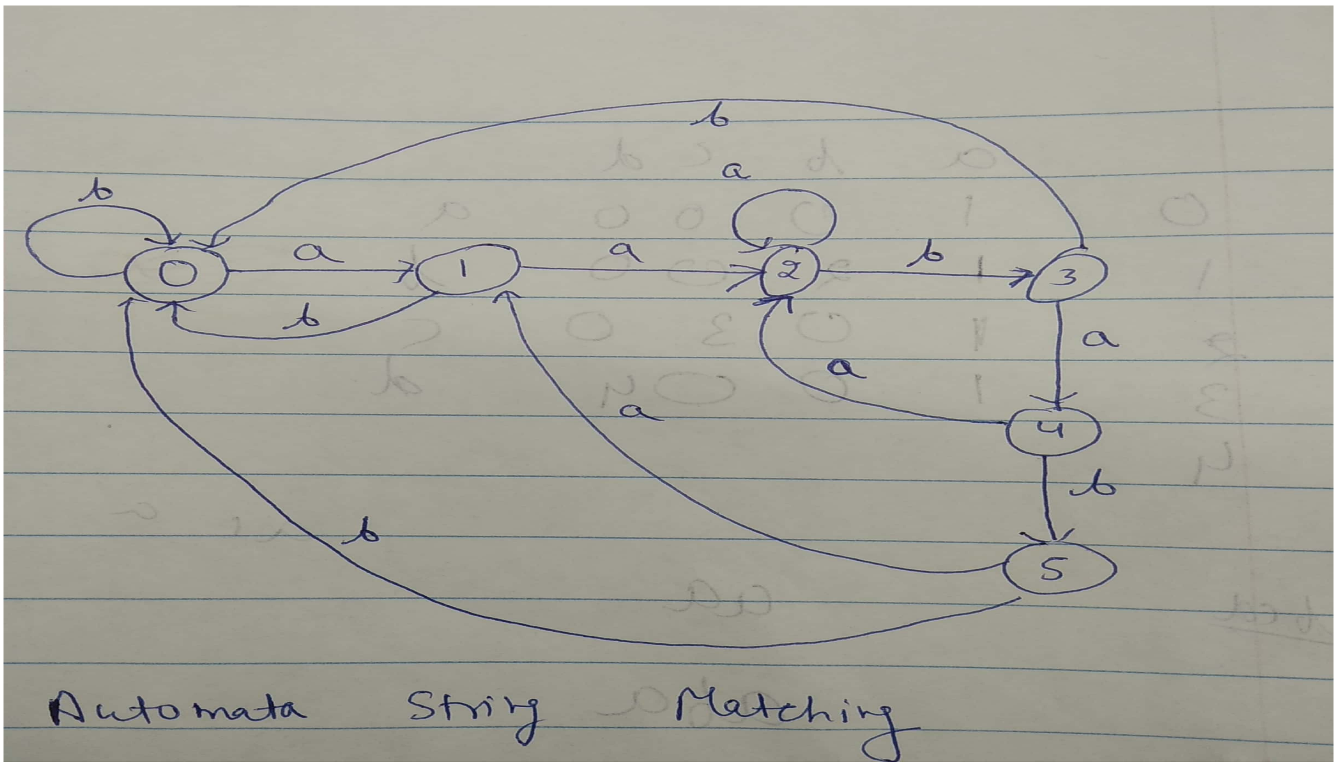

Figure 4.1: Second Approach

State-transition diagram

0 → 1 → 2 → 2 → 3 → 4 → 5 (match) → 1 → 2 → 3 → 4 → 2 → 3 → 4 → 5 (match) → 1 → 2 → 3

5. Non-overlappable Pattern

Given pattern is non-overlappable. Given its conditions to be non-overlappable we can infer that all the characters are different in Pattern, P. This state transition function would go back to either next stage or go back to either initial vertex or first element. It we go to next stage it means suffix of what we have traversed it prefix of what we are trying to find out.

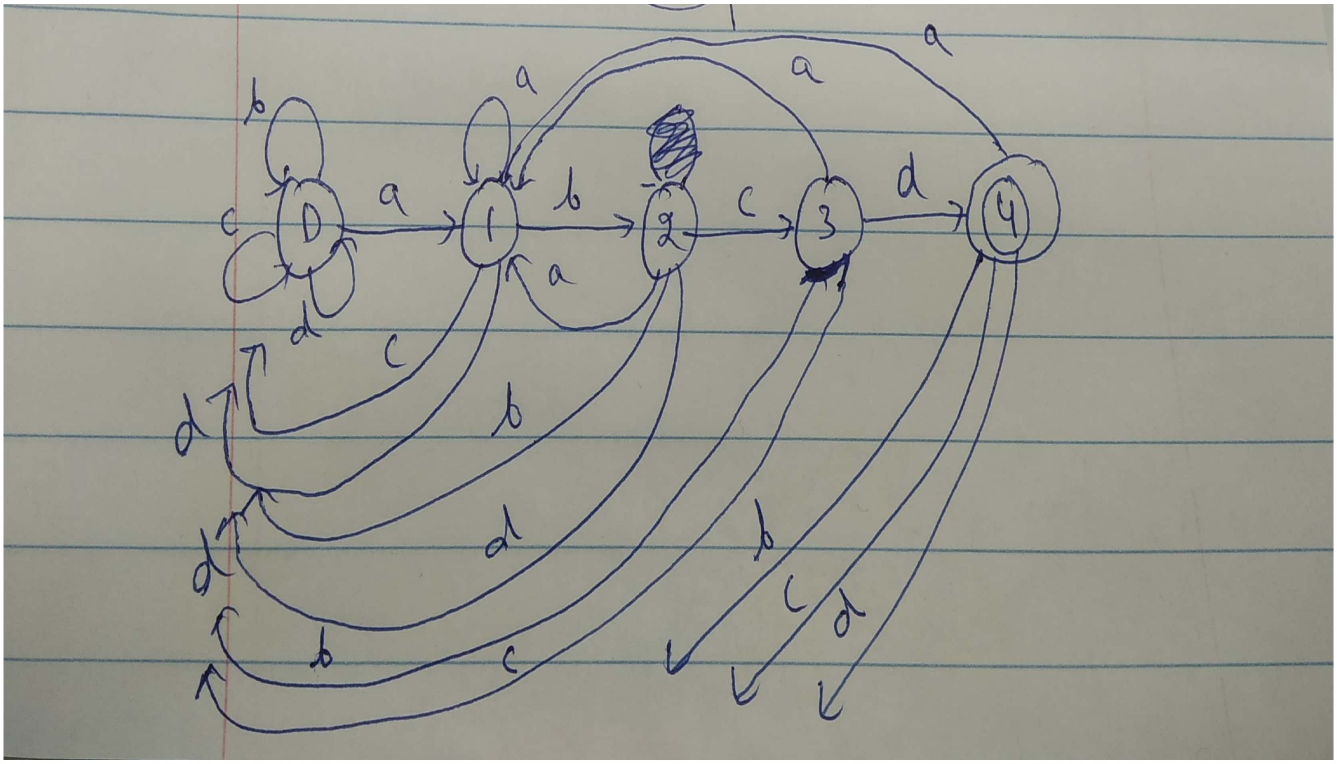

Figure 5.1: Second Approach

6. Cyclic Rotation

If length of T is not equal to T' it means T' is not cyclic rotation of T. If length is equal then lets make a new string, TT by concatenating T with T. If T' is cyclic rotation is should occur in TT. This can be done in linear time using **Knuth-Morris-Pratt algorithm**.

Prepossessing would take O(n) to build suffix prefix matching of Pattern, T'. That is 1 $<= q <= n$ suffix of T'_q is also a prefix of T'_q. Then, we can linearly traverse the text, T when we found a mismatch at T_k, P_q then we would find maximum suffix, M_s in P[1..q] which is also a prefix. Next we would again start matching T_k P[$|M_s|$+1] and soon. This only takes a linear traversal.

Below mentioned are algorithms are referenced from CLRS, chapter 32. Arguments to the KMP-MATCHER is TT and T'.

Algorithm 3 COMPUTE-PREFIX-FUNCTION

```
function COMPUTE-PREFIX-FUNCTION(P)                ▷ where P pattern
    m = P.length
    let π [1....m] be a new array
    π[1] = 0;
    k = 0;
    for q = 2 to m do
        while k > 0 and P[k+1] != P[q] do
            k = π[k];
            if P[k+1] == P[q] then
                k = k + 1;
            end if
            π[q] = k;
        end while
    end for
end function
```

Algorithm 4 KMP-MATCHER

```
function KMP-MATCHER(T, P)                              ▷ where T text, P pattern
    m = P.length;
    n = T:length;
    π = COMPUTE-PREFIX-FUNCTION(P);
    q = 0;
    k = 0;
    for i = 1 to n do
        while q > 0 and P[k+1] != T[i] do
            q = π[q];
            if P[q+1] == T[i] then
                q = q + 1;
            end if
            if q == m then
                return true;
                q = π[2];
            end if
        end while
    end for
    return false;
end function
```