Group Members: Chaitanya Darade, Tanvi Rasam

Project 3: Pattern Matching Algorithms

Project Overview :

In this project, the following algorithms are implemented in Python language:

- Boyer-Moore-Horspool algorithm
- Knuth-Morris-Pratt algorithm
- Boyer-Moore algorithm

Data Structures Used:

1. Lists
2. Numpy Arrays
3. Strings

Pattern matching in computer science is the checking and locating of specific sequences of data of some pattern among raw data or a sequence of tokens. Unlike pattern recognition, the match has to be exact in the case of pattern matching. Many applications make use of pattern matching as a major part of their tasks. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

Algorithms-

KMP

Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm. When a mismatch occurs, we can shift the pattern at most by the largest prefix of $\boldsymbol{P[0..j]}$ **that is a suffix of** $\boldsymbol{P[1..j]}$ so as to avoid redundant comparisons.

The basic idea behind KMP's 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. (value of j) This reduces the no. of comparisons required in searching.

Also, due to this, in cases where we see many matching characters followed by a mismatching character, KMP works better than other approaches as the matching starts from where comparison is yet not done

How to decide upon how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array lps[] (longest proper prefix which is also suffix) of size m that tells us the count of characters to be skipped.

The failure function (lps) can be represented by an array and can be computed in $\boldsymbol{O(m)}$ time

Preprocessing :
In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1].

At each iteration of the while-loop, either

- *i* increases by one, or
- the shift amount *i* - *j* increases by at least one ($F(j - 1) < j$)

There are no more than **2*n*** iterations of the while-loop, thus, KMP's algorithm runs in optimal time $\boldsymbol{O(m + n)}$

Boyer-Moore-Horspool

A simplified version of Boyer-Moore algorithm: preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs

Preprocesses pattern right to left

Upon mismatch, we shift the pattern until –
1) The mismatch becomes a match
2) Pattern P move past the mismatched character.

Shift Table-

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

Where, c is the last character in the pattern and the character in the text where mismatch occurs

Time to **build the table is O(S+m)**, where S= alphabet

It may take **O(mn) time in worst case**. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = "AAAAAAAAAAAAAAAAAA" and pat[] = "AAAAA". This is equivalent to Brute force approach

O(n/m) in Best Case , when the largest shift occurs of size m.

The algorithm gets faster as the substring being searched for becomes longer. This is because with each unsuccessful attempt to find a match between the substring and the string, the algorithm uses the shift table to rule out positions where the substring cannot match.

The average time is O(n).

Boyer Moore

Boyer Moore is a combination of following two approaches.
1) Bad Character Heuristic- It indicates how much to shift based on text's character, c, causing a mismatch

2) Good Suffix Heuristic- It indicates how much to shift based on matched part (suffix) of the pattern

Preprocesses pattern right to left

The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the max of the slides suggested by the two heuristics. **So it uses best of the two heuristics at every step**.

For Bad Character table-

If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's

The two algorithms (i.e. BM and Horspool) act differently, after some positive number k $(0 < k < m)$ of the pattern's characters are matched successfully before a mismatch is encountered.

As, BM chooses the mismatched character of the text to determine the shift and not the rightmost character of the text

we do not want to shift the pattern by 0 or a negative number of positions. Thus, $d_1 = \max\{ t_1(c) - k, 1\}$

For Good Suffix Table-

It is applied after 0 < *k* < *m* last characters were matched.

Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :
1) Another occurrence of t in P matched with t in T.
2) A prefix of P, which matches with suffix of t
3) P moves past t

d2(k)- The distance between matched suffix of size *k* and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

If there is no such occurrence, match the longest part of the *k*-character suffix with corresponding prefix.

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \begin{cases} d_1 & if\ k = 0 \\ \max\{d_1, d_2\} & if\ k > 0 \end{cases}$$

Runtime:

The Boyer-Moore algorithm (original paper) has worst-case running time of $O(n + m)$.

- $\boldsymbol{O(n+m)}$ only if the pattern does not appear in the text, proved by Knuth, Morris, and Pratt in 1977

- Followed by Guibas and Odlyzko in 1980 with an upper bound of $5n$ comparisons in the worst case.

- Richard Cole gave a proof with an upper bound of $3n$ comparisons in the worst case in 1991.

When the pattern does occur in the text, running time of the original algorithm is $\boldsymbol{O(nm)}$ **in the worst case**.

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear.