KNUTH-MORRIS-PRATT

Pattern Matching Algorithm

MAY 2, 2022

DHRUV MAHESHWARI – 2019A7PS0020U
ADAYA NEERAJ – 2019A7PS0045U
KEERTHI PACHALA – 2019A7PS0076U

Acknowledgements

We owe a huge debt of gratitude to everyone who assisted us with this endeavor.

Our sincere gratitude to the project supervisors, Dr. Siddhaling Urolagin and Dr. Razia Sulthana, for providing us with the opportunity to work on such an educational project and for believing in us and supporting us throughout the project, motivating and inspiring us to be more creative.

We also wish to express our heartfelt gratitude to Prof. Srinivasan Madapusi, Director of BITS Pilani, Dubai Campus, who has always steered us in the correct direction.

Index

Abstract

String matching is one of the most difficult, delicate, and time-consuming tasks because it is used in so many contemporary applications, such as text editing, graphics, literature retrieval, biochemistry and so on.

The aim of this paper is to conduct an extensive literature survey and compare all the available approaches related to pattern matching algorithms. The benefits of an efficient KMP algorithm are vast both in terms of both technical and biological world. For example, it can help in detection of tumor cells in a given genome. It can also be used to detect spam emails.

By reducing additional comparisons of letters in text with pattern, KMP improved the native algorithm. Many studies have attempted to increase the serial version's matching efficiency, but recently, some researchers presented a more effective technique to improve the performance of the KMP algorithm by employing the concept of parallel processing. It uses a greedy approach to reduce the number of comparisons between the pattern (input) and text (dataset).

We want to identify all the real life situations where the KMP algorithm can be used and optimize it further.

Problem Definition and Introduction

String searching algorithm, also known as string pattern match algorithm is one of the most important and fundamental algorithms in computer science. The algorithm has a number of different applications in our day-to-day life. Even the internet as we know it today uses string matching algorithms to better the user experience. With rapid speeds of data being processed online, there is a huge need for efficient and fast pattern matching algorithms. Apart from this, the algorithm is used for binary matching, DNA sequence matching, etc.

We aim to find the best, most-efficient string searching algorithms by analysing different pre-existing algorithms like BF algorithm algorithm, KMP algorithm, Rabin-Karp Algorithm etc. By analysing these algorithms using different parameters, we can determine the best suited algorithm with least time complexity and space complexity. We aim to use these algorithms in above-mentioned real-life applications and observe the results.

Objectives

- To study and analyse previous works done in pattern matching algorithms.
- To reduce time complexity and increase the accuracy of KMP algorithm
- To implement algorithm and compare its performance to other algorithms available

Literature Table

Sr no	Objective	Problem statement	Methodology	Dataset	Algorithm	Advantage	Disadvantage	Performance Measure value
1	Make parallel KMP; design a spam filter	Use enhanced version of KMP algorithm with GPU computation to outperform traditional serial versions on CPUs and use it to make spam filter	Divides the dataset into smaller parts and searches in parallel using the GPU's capabilities. Dynamic thread size is used. When pattern size is huge, thread size is 15x pattern size. When pattern size is small, thread size is 30x pattern size. Data cleaning and stop words are considered in data pre-processing	A) String of 194 MB generated of English alphabets. Random substrings taken of size 94.36 MB and 5.38 MB. Pattern length consisted	We must first compute the number of matches that match with the pattern's final 1 to m-1 characters and the number of matches with the pattern's beginning 1 to m-1 characters. An array, called a pre-allocated look up table, is designed to store the number of matches found to the left and right of the chunk. The stored values are compared at the end of execution, and the number of matches at	Reduces time complexity significantly as highlighted.	There's an decrease in accuracy of predicting mail is spam. Takes into account only keywords and not a group of words to detect spam. Sometimes a group of words also help in classifying an email as spam.	Runtime and accuracy of model. A) Speed of 12x remained constant and parallel KMP outperformed B) Speed increased by 100% (from 27.72 to 13.56 s). However accuracy falls from 92% to 85%.

				of 7, 34, 134, 495. B) Enron Dataset; 5172 email consisting of 3672 ham emails and 1500 spam emails	the borders is determined. This entire procedure is carried out in a single loop.				For the text size, n, and pattern size, m, Serial Version comes out to be O(m + n). Its parallel run is O(n / Tn + c), where Tn is the total no of threads.
2	Propose a word matching technique for locating a word's valid shifts in a text stream.	To select in what sequence the letters of the letter/word in pattern P should be equated to the letters in the words in dataset T. Non-matching terms of T can be eliminated in fewer	The algorithm considers the frequency and location of letters or a pair of letters in a word. This knowledge aids in determining the order in which the letters in the input word should be equated to the letters in the dataset. This sequencing eliminates mismatched words in T	A long text string of Holy Quran was taken and the word “الرح” was matched.	The proposed algos name is WORD-MATCHING. It creates a frequency vector first, which is required to establish the sequence in which the letters in the word should be equated to the letters of the	As it stores the frequency vector, it can significantly reduce search time on large datasets.	It doesn’t provide any advantage while working on smaller datasets compared to KMP	Total execution time was evaluated. Pre-processing time for algo was 6730 ms compared to KMPs 201ms.

		comparisons and a faster search can be returned by sorting these comparisons.	in a lesser no of comparisons than ordinary comparison ordering.		tokens generated afterward. It then compares input with min frequency pair and min distance between two pairs. It iterates till letters in word are over or a conflicting match happens.			However as text size increased, new algo performed better by factor of almost 200%
3	Propose a new string matching algorithm based on frequency occurrence of letters in text	Propose FOB that analyzes the letters in P against the current frame on T from least frequency to highest, attempting to arrive at an equal frame in T with the fewest number	The suggested method ranks/orders the letters of the pattern P in order of their occurrence frequency. The letters of patterns are then compared against text T in order of their frequency of occurrence. The iteration starts with the least frequency occurrence character.	T is the Holy Quran's text, which has 77,439 words.	The FOB algorithm first ranks the letters in pattern P with respect to their frequencies. After ranking, the letter with least frequency is passed onto GET-SEARCH-LETTER-WITH-RANK which matches the letter in text T. Once a match	Helps in quickly looking for a word in a given text.	Takes time initially to construct the frequency table. Uses more memory than KMP as it has to store data on frequency of letters.	The comparison is based on how many if-statement comparisons there are in the two algorithms (FOB and KMP). FOB showed 39.6%

		of comparisons.			is found, the other literals are matched with a call to SEARCH function which acts in recursion. Epochs=3000.			improvement compared to KMP when searched for a word. In 2^{nd} exp, P wasn't present in T. FOB performed better by 51.7%. KMP and FOB algorithm was run with a set of randomly selected 1,000 words from T, each of length 3 up to 10. OBF clearly outperformed KMP by 33%
4	pattern searching in	In web application	To detect a "Pattern" P in a dataset T the KMP	India's Centraliz	Declare a 1-dimensional array	KMP method is employed	Uses a good amount of	Time complexity

	complaints reported using KMP algorithm.	issues, there are numerous types of information to be found. The aim of this paper is to give a changed adaptation of the KMP algorithm for text matching.	algorithm compares letter to letter from left to right. As soon as a nonmatch is found, it employs a pre-processed table namely the "Prefix Table" to skip character analysis during the matching process. We use the table's results of the previous character of the non matching alphabet in the pattern at place of mismatch.	ed Public Grievance Reporting And Monitoring System (CPGRAMS) provided the data.	(LPS[p]) where p is the length of the input pattern. Create variables I and j with I = 0,j = 1 and LPS[0] = 0. Evaluate Pattern[i] with Pattern[j]. If a match is found, set LPS[j]= i+1 and add one to both the I and j values. If neither of them match, look at the value of the variable I Set LPS[j] = 0 and increment 'j' by one if it is '0;' if it isn't '0,' assign i=LPS[i-1]. Rep the processes above until all of the LPS[] values are filled.	for its operational capability, and as previously said, it minimizes time complexity.	space to construct the Prefix Table.	was the evaluative criteria. n->text m->pattern length Brute force: O(n-m+1)m) Boyer Moore: O(m+(∑)), O(n) KMP: O(m), O(m+n) Rabin Karp: ϴ(m), ϴ(n+m)

5	Research on string matching Algorithm Based on KMP and BMHS2	The goal of this study is to combine multiple matching algorithms in order to increase the speed of text matching..	The KMP algorithm mainly eliminates the main string pointer's backtracking problem in the BF algorithm's matching process, and uses the partial matching results to shift the pattern string P to the right as distant as possible and then continue to compare, thereby improving the algorithm's effectiveness.	Pattern string P length is around 20 letters. The text T length is around 10M bytes. The window size to be matched is 100, and the length of the text string to be matched is set to 2 million bytes.	When there's no match found, the i pointer does not need to retrace; instead, it can utilize the "partial match" result to check if the value of I needs to be adjusted, and then "slide" the pattern to the right many positions before completing the comparison. If T[i]!=P[j], add if(i+j-km &&P[j]! =T[i+j-k]) I I +j -k ;" As a consequence, the string pointer I increases even when a nonmatch occurs, resulting in enhanced matching performance.	Avoids the BF algorithm's frequent backtracking and increases pattern matching efficiency.,	Has higher space complexity	When the pattern matches with the text T string, the enhanced algorithm's performance increases by 20% compared to the I KMP algorithm and 15% compared to the BMSH2.
6	Using KMP Algorithm in	This paper aims to show	When the pattern and string are not matching,	Enrekan g	First, we'll enter the query word to search	It results in enhanced	A far more complex	According to KMP, the

	Enrekang-Indonesian Language Translator	the implementation of the KMP algorithm in making a regional language translator Enrekang to Indonesia with focus to input in the form of characters and punctuation.	the KMP algorithm conducts the initial step by initializing the variables i and j as a focus to perform shift calculations. Furthermore, on performing pattern and string matching if there is a condition, if the two are matched, the matching result will be saved in a new variable. In the event that there is no match, a movement from left to right will be made..	regional vocabulary data set consist of a table of regional Enrekang words and punctuations	then, the algorithm will start comparing the pattern from the left direction, if the pattern matches then it’ll be checked, otherwise it'll shift one step and will repeat the same process.	algorithm performance and is more efficient than other algorithms.	research will be needed to translate complete sentences.	classification system was determined to be 100 percent capable of translating the words entered during testing. In terms of performance, the implementation takes 0.01901 milliseconds to complete.
7	To determine the next move of the player in Badminton based on the previous shot using the KMP Algorithm	The goal of this study is to develop a computational model that can assist players and coaches with predictions and recommendations for	It is divided into four stages: defining the stroke zone and type, pre-processing data, matching sequence using the Knuth-Morris-Pratt algorithm, and finally making a judgment.	The data used in this experiment is obtained from 20 world badminton matches.	The badminton court is divided into various zones (A-I). Each shot such as lob, drop, smash etc is given a number. Thus the sequence consists of alphanumeric characters. Next the user enters a zone-shot, the algo then	It is an efficient algorithm used for string matching hence used in this experiment	The marking of zones and the shot offered to create the dataset requires a lot of time and requires a lot of manual work.	There are 3 of 17 simulations that are not match with the actual movements. So, based on this experiment the accuracy

		shuttlecock placement..			finds the pattern in text. Hence the user sees all possible next shots.			of the system on the fitting step is 82.3%.
8	Analyse existing pattern matching algorithms to develop an enhanced, more efficient algorithm for string searching.	This paper aims to study two most commonly used string pattern matching algorithms and tries to propose a new and better string matching algorithm.	The KMP Pattern process has a pattern matching process 'p' and the text matching process as 't', where both are compared from left to right whereas the Boyer-Moore algorithm compares the string pattern from L-R and the characters are compared from R-L. The combination of these algorithms gives the new and improved KMPBS algorithm. The last character of the pattern string p[m] is compared to characters of text string T. If there is a match, KMP algorithm is used to match them from L-R.	'Alpha' is a character table which is used that has a default number of ASCII character s,i.e. 256 character s.	The first step of the algorithm checks for the character at the last position of the string for pattern matching and this is done using the KMP algorithm. The variable 'i' is assigned to the current character position in the text string. It keeps checking for every character in the string till a match is found(loops till match fails), and once the pattern is matched, it sets the flag value true. If match fails, then the flag value is set to false and the position of 'i' is	The KMPBS algorithm performs much better than the KMP and BM algorithms. The KMPBS algorithm is preferred because it reduces the number of iterations through each of the characters in string pattern matching for pattern match processes.	Although the time complexity is greatly reduced, the KMPBS algorithm does not entirely address the space complexity issue. With strings of long length, the KMPBS algorithm needs more space allocation, and also the right value and single values need to be addressed.	For a selected string of length l=24,a string of length l=4 is compared. The KMPBS algorithm finishes the string pattern search in 10 iterations of the character search, whereas the KMP algorithm finds the string pattern in 16 iterations. With a total text length of

			If no match is found, the characters from text of T are used to identify the pattern of string P while the jth pointer position will reset to the first character position.		changed to the unmatched character. After this BMHSI algorithm is implemented to obtain a new position. This algorithm keeps looping till the next match fails or succeeds.			327 and pattern length of 13, the KMPBS performs the process 57 times whereas the KMP algorithm takes 628 number of comparisons. With a text length of 1035, the KMPBS Algorithm performs the best with only 94 number of model series, whereas the KMP algorithm takes 1020 times. The BM Algorithm

								performs slightly better, taking 586 number of model series'. The KMPBS algorithm is approx (1/10~1/6) of the KMP algorithm and the BM algorithm (approx 1/5~ 1/3), significantly reducing the number of the matching efficiency.
9	Applications of the Knuth Morris Pratt	With the internet growing rapidly every second, string	Traffic data pockets consist of the majority of data on the internet. To read a message quickly, a high speed	Verifying the KMP algorithm is in	The KMP algorithm sets the pointer 't' and 'p' at the starting position of the text and divides the	KMP algorithm greatly reduces the time	As length of data increases, i.e size of alphabets	Test input file of 50,000 messages is provided to find the

	Algorithm in network flow	lookup is relatively more time consuming. This paper aims to find a more suitable string searching algorithm for high speed networks.	systematic tool is required to achieve this function. According to the need of users, different keywords need to be matched and generated accurately. The efficiency of the BF (Brute Force) algorithm is very low. So, the KMP algorithm is implemented that can reduce the time process for string matching and string lookup for high speed internets.	fact capable of field processin g, a performa nce test input file for 50,000 messages (ordinary LAN Traffic) is used.	partition and then compares the current pointer to the character analogous in the starting position of the string. The next() function is given and the partition is made If they match, t and p are moved respectively, else p is moved away from the starting position so the pointer t does not have to back track and restart the entire process.	complexity, O(m+n), whereas the BF algorithm has an efficiency of O(mn). The KMP algorithm’s biggest advantage to the BF algorithm is the elimination of backtracking.	increase, the KMP shows reduced performance. For large and heavy traffic data pockets, special hardware selection is required to boost the performance of the KMP algorithm.	substring “FireFox” and “Chrome” to test the total time consumption of the field matching methods. To find “Firefox”, the Brute Force algorithm consumed 0.116 seconds of CPU time,whereas the KMP took 0.074s. To find “Chrome”, BF algorithm took 0.109s and the KMP

								Algorithm took 0.056s.
10	Comparison of Search Algorithms in Javanese-Indonesian dictionary applications	Search processes typically use string match algorithms as a data search algorithm. This paper aims to find the accuracy and avg. CPU processing time of search results for three different string matching algorithms – BM Algo,KMP Algo and Horspool algorithm.	We first test the performance of the BM,KMP and the Horspool algorithm in the Indonesian-Javanese dictionary application. First, the input data string is given and then the pre-processing phase is initiated. In this phase, the text is mined and data is represented in a structured format till the data is ready to be processed. After testing out the different algorithms, the last stage is the output stage where the calculations are performed in the form of translation of vocabulary or search string data.	The three string matching algorithms are tested on the existing Javanese dictionary, on 1500 vocabularies with 400 experiments.	The BM algorithm first matches the pattern at the beginning of the text. It matches from R-Lt to match the substring pattern characters with the characters in the matched text until a match is found. The KMP algorithm matches the pattern from the beginning of the test. It matches the pattern by matching each character till a match is found. The Horspool algorithm only uses bad-character	The BM and KMP algorithm have more accuracy compared to the Horspool algorithm. The avg.time for processing of KMP is better than that of BM and Horspool Algorithms. CPU processing time of KMP algorithm is at least n^2 and n values.	The biggest drawback for the BM algorithm is the pre-processsing time and space required for string matching. This depends on the size of alphabets or patterns. For really small patterns with no overlapping strings, it is better to use	After testing the dataset, the accuracy of the Horspool algorithm is 85.3% while that of KMP and BM is 100%. Accuracy Level = (No.of successful samples/total samples)*100 The fastest speed of the KMP algorithm with an

					shifting(depends on character mismatch). It uses the rightmost character to find the shift distance that needs to be performed.		the naïve algorithm or Rabin-Karp algorithm that take O(mn) time in worst cases.	average of 25ms, Horspool 39.9ms and BM took 44.2ms. Testing on efficiency of space complexity, the BM algorithm has an overall n(11n), the KMP has an overall of n(8n) and Hosrpool has an overall of n(10n).
11	To analyze already existing string matching algorithms to	String search algorithms have many practical applications in real life like	Firstly, we match the text patterns by analysing the text inthe documents using string matching algorithms. For this, we use pre-	For experimentation of the different algorith	The Enhanced Rabin-Karp algorithm uses the hash() function to identify a e set of	The pre-existing KMP algorithm has the best efficieny for the dataset,	The string matching algorithms are only analysed as a small	After experimentation, the parameters for testing are time, number

	develop new string pattern match algorithms .	pattern searching in alphabets, binary alphabets or DNA alphabets in genome sequencing. This paper aims to analyse two algorithms to find the best performing algorithm for string pattern matching.	existing string match algorithms- BF,KMP,BM and Rabin-Karp algorithm. The performance factors are generated for these four algorithms. Then, two new string matching algorithms are proposed for string matching and both the results are compared to find the best pattern matching algorithm.	ms, the performa nce testing is done using two types of documen ts- .docx, .txt.	patterns in a particular input text. The Enhanced KMP algorithm checks for pattern p within a string t by applying that when a mismatch occurs, the suffix is matched with the prefix and the search is continued from after that suffix so as to avoid re-checking of the entire string from the beginning.	whereas, the Enhanced KMP algorithm gives better accuracy over the selected dataset parameters when compared to the pre-existing string matching algorithms.	finite length in single lines and files in the dataset. For increasing size of the strings and patterns, the KMP algorithm has decreased efficiency and accuracy rates.	of iterations and accuracy. The pre-existing KMP algorithm performs better than any other algorithms over the selected dataset parameters.

Analysis of Algorithm

For the purpose of comparing various available algorithms, we have divided this section into 3 categories:

1. Naïve string searching algorithm
2. KMP algorithm
3. Optimized KMP algorithm

Naïve string searching algorithm

The Naive KMP algorithm is essentially based on brute force comparison of every single character in the pattern at each possible position of the selected pattern in a sequence/string. It is one of the most obvious approaches to the sequence matching problem where every single character is matched to the pattern and this continues till a character match is found. If there is a mismatch, the Naive KMP algorithm proceeds to shift the index by a position to the right of the current index pointer and then again repeats the step-by-step comparison of each character.

Suppose that there is a string s = 1121131141121111211 and a pattern p = 1121.
The algorithm first calculates the length of the string and pattern. Then it assigns a pointer to the 0th index of the string s and starts traversing and comparing the first character to the pattern.

Step 1: s = **1** 121131141121111211
p = **1121** -------------------- there is no pattern match so then it jumps to the next consecutive index.

Step 2 : s = **11** 21131141121111211
p = **1121** -------------------- still there is a mismatch so it proceeds to next index.

Step 3 : s = **112** 1131141121111211
p = **1121** -------------------- pattern mismatch.

Step 4 : s = **1121**131141121111211
p = **1121** -------------------- pattern match found.

This loop continues till the algorithm reaches the end of the string s and traverses all the characters in the sequence.

Code :

```
#include <bits/stdc++.h>
using namespace std;

void search(char* p, char* s) // p is assigned to the pattern and s is assigned to the string
{
   int M = strlen(p); //length of pattern p
```

```
    int N = strlen(s); //length of string s

    /* for loop to traverse  p[] */
    for (int i = 0; i <= N - M; i++) {
        int j;

        for (j = 0; j < M; j++)
            if (s[i + j] != p[j])
                break;

        if (j == M) // if p[0...M-1] = s[i, i+1, ...i+M-1]
            cout << "String matched at index:  "
                << i << endl;
    }
}

int main()
{
    char s[] = "112113114112111211";
    char p[] = "1121";
    search(p, s);
    return 0; }
```

Output :

String matched at index: 0
String matched at index: 9
String matched at index: 13

KMP algorithm

The time complexity of KMP Algorithm is O(n+m) in the worst case where n is the length of the text and m is the length of patter to be matched. The KMP matching algorithm takes advantage of the pattern's declining feature to reduce the worst-case complexity to O(n). The essential premise of Knuth Morris Pratt's method is that anytime we identify a discrepancy (after a few iterations), thusfar we know some of the attributes in the next set’s text. We use what we know to prevent attributes from matching that we know will match in any case.

- The KMP algorithm pre-determines p[] and makes an array longest prefix suffix[] of size m that helps to skip attributes whilst comparing lps[j] with T[i].
- Then we look for lps in sub-patterns. Mostly we will concentrate on sub-strings of patterns which are either prefixes or suffixes.
- For every sub-pattern p[0..i] where i ranges from 0 to m-1, lps[i] stores the size of the maximum matching proper prefix, which is also a suffix to the sub-pattern pat[0..i].

```
def Search(p, t):
        a = length of p
        b = length of t

        prefix_table = [0]*a
        j = 0 # index for p[]

        prefixTable(p, a, prefix_table)

        k = 0 #index for t[]
        while k < b:
                if p[j] == t[k]:
                        k++
                        j++

                if j == a:
                        print ("Pattern found at " + str(k-j))
                        j = prefix_table[j-1]

                elif k < b and p[j] != t[k]:
                        if j != 0:
                                j = prefix_table[j-1]
                        else:
                                k++

def prefixTable(pat, n, table):
        l = 0

        table[0]
        i = 1


        while i < n:
                if p[i]== p[l]:
                        l ++
                        table[i] = l
                        i ++
                else:
                        if (l!= 0):
                                l = table[l-1]
                        else:
                                table[i] = 0
                                i ++
```

<u>Example:</u>

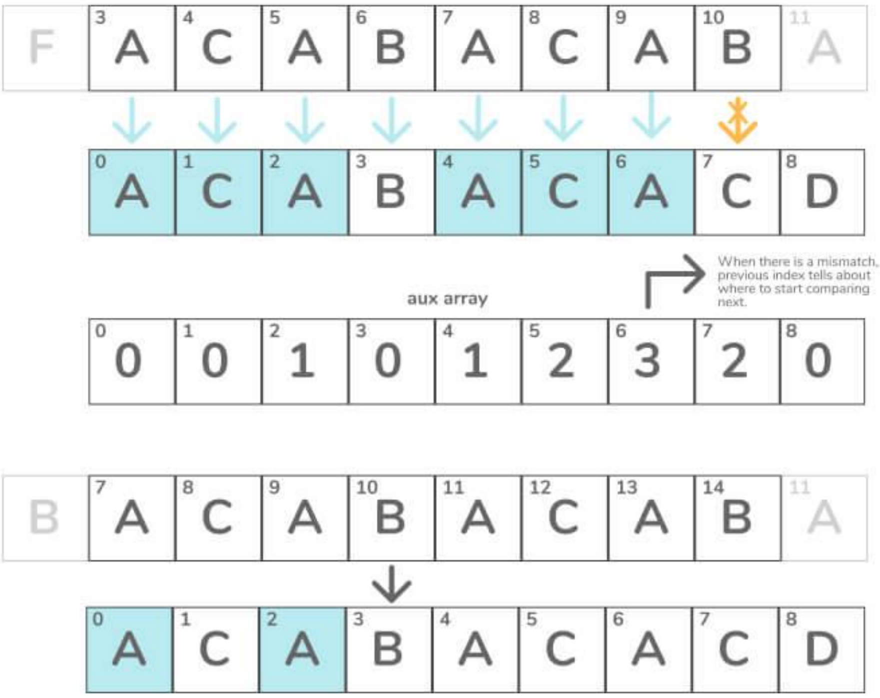
t = "SSSSUSSSUS"
p = "SSSS"
Here we are comparing first set of **t** with **p**
t = "**SSS**SUSSSUS"
p = "**SSSS**" [first position]
We found a match.
In this step, we will be comparing next set of **t** with **p**.
t = "**SSSSS**USSSUS"
p = "**SSSS**"

Now this is where KMP comes in play and optimizes over any other algorithm. In the second set, we will compare only the 4th S of the pattern with 4th character of the current set of text to decide whether the current set matches or not. And since we know, first three attributes are going to match anyway so we will skip analysing the first three attributes.

Now to decide how to skip and how many attributes are to skipped we use prefix table[];

- We start by comparing p[0] with t[0].
- We will keep comparing attributes of t[i] and pat[j] and increment i an j by 1 on successful match.
- When there is a non matched pair:
    - We know that attributes in p[0..j-1] match with the attributes in t[i-j…i-1]. (Initial value of j is 0 and we will increase it whenever there p[i]==t[j]).
    - As per the above definition, we know that prefix_table[j-1] is the number of attributes in p[0…j-1] that form both proper prefix and suffix.
    - From the above points, it is clear that there is no need to compare again the prefix_table[j-1] attributes with the attributes in t[i-j…i-1] as the analysis and matching has already been done.

DRY RUN OF THE ALGORITHM



Optimized KMP Algorithm

KMP algorithm has a time complexity of O(n + m) where n = length of the string and m = length of the pattern to be matched. Since the time complexity is already linear, it is difficult to reduce it further. However, we can try and remove some unnecessary computations which might help to bring down the overall execution time of the algorithm.

Consider input: T = “rpqpqpqrpqpqprp”, P = “pqpqprp”

	0	1	2	3	4	5	6	7	8	…
T	r	p	q	p	q	p	q	r	p	…

j	1	2	3	4	5	6	7
P [j]	p	q	p	q	p	r	p
lps	0	0	1	2	3	0	1

We compare T[i] with P[j+1]

i = 0, j = 0 → r != p (as j is already 0, increment i)
i = 1, j = 0 → p == p
i = 2, j = 1 → q == q
i = 3, j = 2 → p == p

i = 4, j = 3 → q == q
i = 5, j = 4 → p == p
i = 6, j = 5 → q != r (j moves to j = 3)
i = 6, j = 3 → q == q
i = 7, j = 4 → r !=p (j moves to j = 2)
i = 7, j = 2 → r !=p (j moves to j = 0)
i = 8, j = 0 → p == p (i moves to 8 as j is already 0). Thereafter the entire string matches.

The mismatch happens at T[7] twice. If we can somehow avoid multiple checking at the same T[i] location and make the algorithm check that T[i] is compared with P at most once, we can further reduce the running time of the KMP algorithm, thus giving us an optimized KMP algorithm.

Our goal is to effectively calculate how many shifts are required in order to avoid re-comparing the already mismatched values.

The optimized algorithm should work as follows:

When mismatch occurs at T[6], we check P[4] and T[7]. As p !=r, we check P[0] and T[8]. They match and thereafter the whole pattern matches. The values inside P [] and T[] are computed by altering the pre-processing technique.

Algorithm:

1. If pattern P has X unique characters, we compute failure table having X lps (longest prefix suffix).
2. Failure Table FT[][] counts the length of longest prefix suffix of the pattern P[1…l] + c where c is a unique key (character).
3. We have to make an approximate guess ie by how much we need to shift if the mismatched character is c.
4. Thus, if a mismatch happens i.e. T[i] != P[j+1], j is updated to FT[T[i]][j-1] and i is incremented. Thus, we skip comparing T[i] more than once as we believe that T[i] is either matched or skipped.
5. To construct failure table:

```
if (P[lps[l]] = = t):
   FT[t][l] <- lps[l] + 1;
else:
   if (lps[l] = = 0):
      FT[t][l] <- 0;
   else:
      FT[t][l] <- FT[t][lps[t] - 1];
```

For a character ‘t’ in P, we calculate longest suffix and store it in FT[t][l]

lps[l] states that P[0…lps[l]-1] is the lps of P[1..l], we only verify if P[lps[l]] = = t.

If yes, that means P[0…lps[l]] is the lps of P[1..l] + t, therefore we make FT[t][l] = = lps[l] + 1.

If false, that means P[lps[l]] doesn’t match with t but P[0…lps[l]-1] matches with lps[1…l]. Thus the next updated matching starts taking place at FT[t][lps[t] - 1].

If lps[l] = = 0, lps[l] – 1 would become -1 which is not possible. Hence we update FT[t][l] = = 0.

For example: T = “rpqpqpqrpqpqprp”, P = “pqpqprp”, failure table would look like

‘p’ [1 1 1 3 1 1 1]
‘q’ [0 0 2 0 4 0 2]
‘r’ [0 0 0 0 0 0 0]

For p,

J	0	1	2	3	4	5	6
P [j]	p	q	p	q	p	r	p
lps[j]	0	0	1	2	3	0	1
FT[p][j]	1	1	1	3	1	1	1
Explanation	As P[0] = = p	As P[0] = = p	As P[1] ! = p & FT[t][l] <- FT[t][lps[t] - 1]	As P[2] = = p, 2 + 1	As P[3] !=p & FT[t][l] <- FT[t][lps[t] - 1]	As P[0] = = p	As P[1] ! = p & FT[t][l] <- FT[t][lps[t] - 1]

Similarly, we can calculate values for q and r.

Code on :
https://colab.research.google.com/drive/1ShaQY9j_bE3848jCo94kkApFiHHVkCEO?authuser=1#scrollTo=of04wht79nAA

Results and Discussion

String matching algorithms are probably one of the most important and fundamental algorithms in computer science. It has a huge array of implementations from simple letter and sequence matching to DNA alphabet matching in genome sequencing. So it is very crucial to compare and identity the best performing string matching algorithm.

The Naive KMP algorithm has a time complexity of O(mn) as it traverses every character in the sequence. Sometimes the string does not contain the pattern, but the algorithm continues to check every character till it reaches the end of the string length, here, the match is not found in the entire string which is why it is the least performing algorithm in the string matching algorithms.

The KMP algorithm for pattern searching gives a time complexity of O(m+n) which is much better than the Naive KMP algorithm. This O(n) time complexity is achieved by completely avoiding backtracking. The algorithm analyses and skips comparisons of characters in string S that have already been checked and compared with the sequence of pattern P. The KMP algorithm preprocesses the string and then creates an array lps[] which skips already matched characters.

The Optimised KMP algorithm generates a time complexity of $O(\sum p * m + n)$.
The algorithm constructs a failure table (preprocessing) and then it takes $O(\sum p * m)$ where p is the number of unique elements in P and m is the length of the pattern.
Overall, the modified/optimized algorithm takes $O(\sum p * m + n)$ where n is the length of T.

The overall time unit looks worse compared to the original KMP algorithm. However, pre-processing needs to be done only once and when we run multiple times, as this algorithm compares T[i] only once, the overall time complexity reduces.

String Matching Algorithms	**Time Complexity Analysis**
Naive KMP Algorithm	O(mn)
KMP Algorithm	O(m+n)
Optimised KMP Algorithm	$O(\sum p * m + n)$

The new proposed Optimised KMP algorithm although looks worse than the original KMP algorithm, over multiple re-runs it ultimately performs better than the other string matching algorithms as the overall time complexity, is greatly reduced. It can be used for string and sequence matching and to automatically reduce the time it generally takes to detect misspelled words and then later correct them. Hence, this new optimized KMP algorithm minimizes the countless hours of human effort spent on checking for patterns from n number of strings.

Conclusion

Through the literature survey, we were able to find out the various fields where Knuth-Morris-Pratt (KMP) algorithm is used. We were able to get insights on how this state-of-the-art algorithm has been optimized over the years to further bring down its running time.

Through our project, we have tried to optimize it in our way by reducing the number of comparisons and thus tried reducing its execution time as well.

References:

1. V. K. P. Kalubandi and M. Varalakshmi, "Accelerated spam filtering with enhanced KMP algorithm on GPU," 2017 National Conference on Parallel Computing Technologies (PARCOMPTECH), 2017, pp. 1-7, doi: 10.1109/PARCOMPTECH.2017.8068335.
2. M. AbuSafiya, "Word Matching Algorithm based on Relative Positioning of Letters," 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), 2019, pp. 69-72, doi: 10.1109/JEEIT.2019.8717531.
3. M. AbuSafiya, "String Matching Algorithm based on Letters' Frequencies of Occurrence," 2018 8th International Conference on Computer Science and Information Technology (CSIT), 2018, pp. 186-188, doi: 10.1109/CSIT.2018.8486384.
4. Ashok, Anchana, Sumy Roslin Joseph, Anjana Vinod, and Juby Mathew. "COMPLAINT REPORTING AND PATTERN SEARCHING USING KMP ALGORITHM."
5. Y. Zhou and R. Pang, "Research of Pattern Matching Algorithm Based on KMP and BMHS2," 2019 IEEE 5th International Conference on Computer and Communications (ICCC), 2019, pp. 193-197, doi: 10.1109/ICCC47050.2019.9064076.
6. D. Anggreani, D. P. I. Putri, A. N. Handayani and H. Azis, "Knuth Morris Pratt Algorithm in Enrekang-Indonesian Language Translator," 2020 4th International Conference on Vocational Education and Training (ICOVET), 2020, pp. 144-148, doi: 10.1109/ICOVET50258.2020.9230139.
7. Riza, Lala Septem, Muhammad Irfan Firmansyah, Herbert Siregar, Dian Budiana, and Alejandro Rosales-Pérez. "Determining strategies on playing badminton using the Knuth-Morris-Pratt algorithm." *Telkomnika* 16, no. 6 (2018): 2763-2770.
8. Hou Xian-feng, Yan Yu-bao and Xia Lu, "Hybrid pattern-matching algorithm based on BM-KMP algorithm," 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE), 2010, pp. V5-310-V5-313, doi: 10.1109/ICACTE.2010.5579620.
9. Q. Meng, Z. Lei, D. He and H. Wang, "Application of KMP algorithm in customized flow analysis," 2017 3rd IEEE International Conference on Computer and Communications (ICCC), 2017, pp. 2338-2342, doi: 10.1109/CompComm.2017.8322953.
10. Yana Aditia, Gerhana, Lukman Nur, Huda Arief Fatchul, Alam Cecep Nurul, Syaripudin Undang, and Novitasari Devi. "Comparison of search algorithms in Javanese-Indonesian dictionary application." *TELEKOMIKA Telecommunication, Computing, Electronics and Control* 18, no. 5 (2020): 2517-2524.
11. Sri, M. Bhagya, Rachita Bhavsar, and Preeti Narooka. "String matching algorithms." *Int. J. Eng. Comput. Sci* 7, no. 3 (2018): 23769-23772.