CS 584 – Algorithm Design and Analysis Project

Comparison of String Searching Algorithms

Harie Vashini Dhamodharasamy Kalpana

PSU ID: 978127144

Computer Science

Winter 2020

Table of Contents

1. ABSTRACT

In Real-time, world problems need fast algorithm with minimum error. Every text-editing application like notepad, MS Word, MS Excel, etc., requires a mechanism to find a particular pattern/string. The objective of string or pattern searching is to search for a particular pattern in a large body of text. For instance, when one needs to find some text or a word in a text editor, it is tough to find it manually. On the other hand, using a shortcut key Ctrl+F, which generates a pop-up window to enter text that needs to be searched is easy to find the occurrences of that particular text in the entire document by highlighting all its occurrences. Along with this application, string searching algorithms can also be used in searching for particular patterns in DNA sequences, Internet search engines also use them to find Web pages relevant to queries. The motivation behind this paper is to discuss various string searching algorithms (also called as string-matching algorithms) and its comparisons. I have implemented Naïve method, Knuth-Morris-Pratt Algorithm, Boyer Moore Algorithm, Rabin Karp Algorithm. For this project, I have used a plaintext file from project Gutenberg to perform the string searching algorithms and to search for patterns in it. Comparative analysis on Execution time of a same pattern and different lengths of patterns is done on these string searching algorithms.

2. INTRODUCTION

A string searching algorithm aligns the pattern with the beginning of the text and keeps shifting the pattern forward until a match or the end of the text is reached. It is a special case of pattern matching where a pattern is described by a finite set of symbols or alphabet (Σ). Σ may be a usual human language alphabet, for example, the letters from A to Z or other applications that use a binary alphabet ($\Sigma = \{0,1\}$) or a DNA alphabet ($\Sigma = \{A, C, G, T\}$) in case of bioinformatics. String-searching algorithm involves finding one or all the occurrences of a pattern P [1…m] of length m in a larger text T [1…n] of length n where m, n>0 and m ≤ n. The problem is to find an integer s, called **valid shift** where $0 \le s < n-m$ and T [s+1......s+m] = P [1......m].

For example, consider a DNA sequence "AGCAATGTTCAGCAATAAGCAAT", if we are to find pattern "CAAT" in it then, we get indices 2,12,19 as output in search of that pattern. Here the length of the text n = 23 and length of the pattern m = 4, our first occurrence is seen at position 2 i.e. at shift s=2 which is valid since $0 \le 2 < 19$.

String-searching algorithms can be divided into Exact string-searching and Approximate string-searching respectively. Exact string-searching algorithm does not allow any tolerance, all occurrences of a given pattern P from a given text T are found. In this string searching, the characters present in a pattern window and a text window are compared. The length of both windows must be of equal length during the comparison phase. Shifting of characters in case of a mismatch is necessary to develop efficient algorithms. Approximate string-searching algorithm finds a substring that is close to a given pattern string. This algorithm is contrary to exact string-searching algorithm that expects a full match. Here we find all the occurrences of the pattern in text whose edit distance to the pattern is at most k. Edit distance (Hamming distance or Levenstein distance) between two strings is equal to minimum number of character insertion, deletions, substitutions need to make them equal. These algorithms are introduced to address spell errors present in patterns or texts, low quality of texts, and difficulty in searching foreign names.

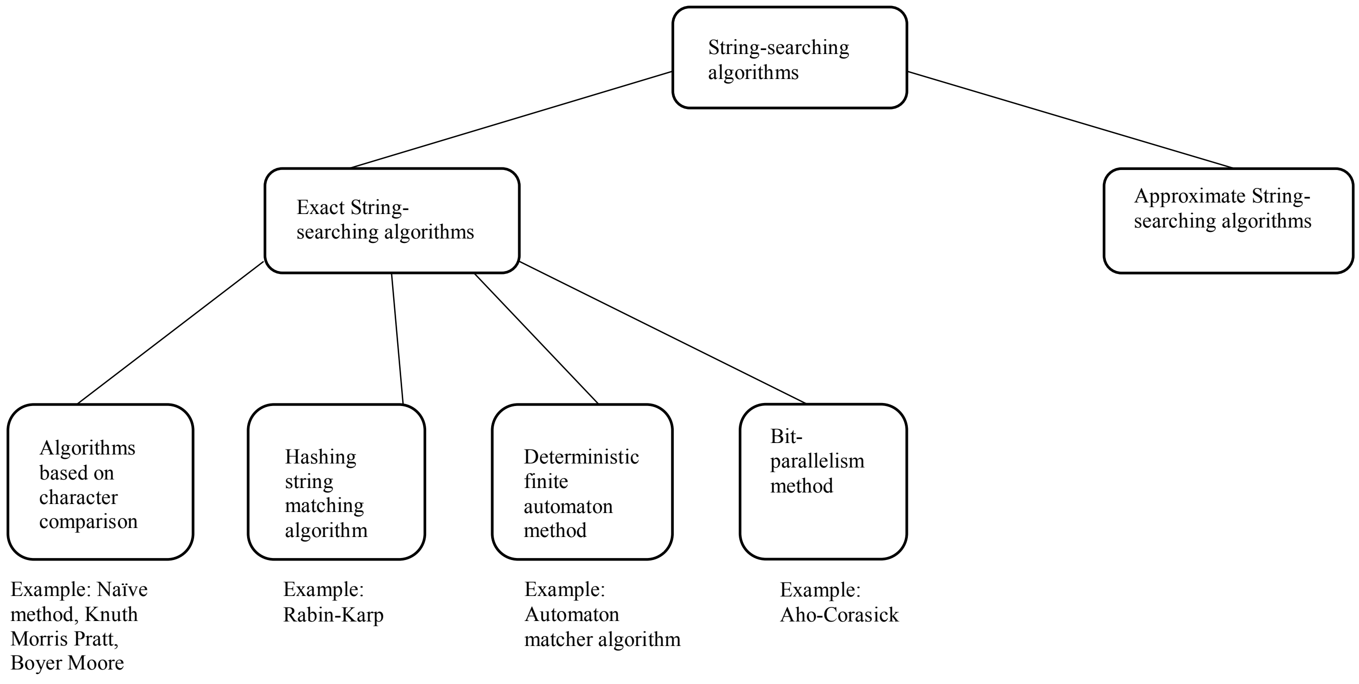
In this paper, we will be focusing on Exact string-searching algorithms. Four exact string-searching algorithms are discussed in detail and we compare the efficiency of each algorithms to determine the best. **Section 3** will discuss on Exact string-searching algorithms having subsections: **3.1** Algorithms based on character comparison which gives a brief view on **3.1.1** Brute force approach, **3.1.2** Knuth Morris Pratt and **3.1.3** Boyer Moore; **3.2** Hashing String matching algorithm which describes **3.2.1** Rabin-Karp algorithm. **Section 4** consists of description of the experiment, results and comparisons among the algorithms. **Section 5** lists various applications of string-searching algorithms and ending with **section 6** that gives conclusion.

3. EXACT STRING-SEARCHING ALGORITHMS

The exact string-searching algorithms deal with finding all not part occurrences of pattern P in text T. This means all alphabets in the pattern must be matched to corresponding match subsequence perfectly. Exact string-searching algorithms can be classified based on different character comparison methods.
Four categories:

- Algorithms Based on Character Comparison
- Hashing String Matching Algorithm
- Deterministic Finite Automaton Method
- Bit-Parallelism Method

We focus on two categories, Algorithms based on character comparison and Hashing string matching algorithm.


String-searching algorithms
Exact String-searching algorithms
Approximate String-searching algorithms
Algorithms based on character comparison
Hashing string matching algorithm
Deterministic finite automaton method
Bit-parallelism method
Example: Naïve method, Knuth Morris Pratt, Boyer Moore
Example: Rabin-Karp
Example: Automaton matcher algorithm
Example: Aho-Corasick

3.1 ALGORITHMS BASED ON CHARACTER COMPARISON

Character-based approach is known as a classical approach that compares characters to solve string searching problems. Character-based approaches have two key stages: searching and shift phases. We discuss three algorithms here, namely Brute force approach, Knuth Morris Pratt and Boyer Moore.

3.1.1 BRUTE FORCE APPROACH / NAÏVE METHOD

This approach scans the text from left to right and checks the characters of the pattern, character by character against the substring of the text string beneath it. This algorithm is the simplest string-searching algorithm and does not require pre-processing phase also no need of extra space. Since it does not require pre-processing, Naïve method's running time is equal to its matching time.

Pseudocode

```
NAIVE-METHOD (T, P)
    n = T.length
    m = P.length
    for s = 0 to n - m
        if P[1..m] = = T[s+1..s+m]
            print "Pattern found at index", s
```

T refers to String Text of length n and P refers to the pattern of length m, to be searched.

Time complexity

Best case: O(n) occurs when the first character of the pattern is not found in the text.
Worst case: O(m*(n-m+1)) occurs when the last character is different or when the characters in both text and pattern are the same. Where, sometimes worst case turns to $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$.

Space complexity

It is O(1) since no extra space is used.

3.1.2 KNUTH-MORRIS-PRATT ALGORITHM

Naïve approach does not work best for all cases mainly when we see any matching characters followed by mismatching characters. This algorithm searches for occurrences of a pattern P within a main text X from left to right by employing the observation that when a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons, thus benefiting from previously matched characters. This algorithm provides the advantage that the pointer in the text is never decremented.

Number of alphabets to be skipped in order to avoid redundant comparisons can be found by preprocessing. We pre-process pattern and prepare an integer array lps [] that gives us count of alphabets to be skipped.
For each sub pattern P[0..i], lps[i] stores length of the maximum matching proper prefix

lps[i] = the longest proper prefix of P[0..i] , which is also a suffix of P[0..i]

and start matching characters from T[i] and P[j] (j index of P, from where the comparison starts in order to omit redundancy).

Pseudocode

```
COMPUTE_PREFIX(P, m, lps)
  length := 0
  lps[0] := 0
  for all character index 'i' of P, do
    if P[i] = P[length], then
      increase length by 1
      lps[i] := length
    else
      if length ≠ 0 then
        length := lps[length - 1]
        decrease i by 1
      else
        lps[i] := 0

KMP_SEARCH(T, P)
  n := size of T
  m := size of P
  call COMPUTE_PREFIX(P, m, lps)
  while i < n, do
    if T[i] = P[j], then
      increase i and j by 1
    if j = m, then
      print the location (i-j) as there is the P
      j := lps[j-1]
    else if i < n AND P[j] ≠ T[i] then
      if j ≠ 0 then
        j := lps[j - 1]
      else
        increase i by 1
```

Time complexity

The cost of each iteration of the repeat while loop in KMP_SEARCH function is O(1). Therefore O(n). The preprocessing phase in this case the COMPUTE_PREFIX function takes Θ(n). Thus, total time complexity of this algorithm is O(n).

Space complexity

It is O(m) since we construct lps array.

3.1.3 BOYER MOORE ALGORITHM

The BM approach is the fastest pattern searching algorithm for a single keyword in both theory and practice. In the KMP algorithm the pattern is scanned from left to right, but the BM algorithm compares characters in the pattern from right to left. BM algorithm involves two stages, pre-processing and searching. This algorithm can shift as many m characters as possible in case of a mismatch; that is, it computes the amount by which the pattern is moved to the right before a new matching attempt is undertaken. The shift can be computed using two heuristics called Bad Character Heuristics and Good Suffix Heuristics.

Bad Character Heuristics

A bad character is a character in the text that does not occur in the pattern at all. When a mismatch happens at this character we shift the pattern until the mismatch becomes a match or the pattern moves past the mismatch character.

Good Suffix Heuristics

A good suffix is a suffix that has matched successfully. A preprocessing table is created as suffix table. In this procedure, the substring or pattern is searched from the last character of the pattern. When a substring of main string matches with a substring of the pattern, it moves to find other occurrences of the matched substring. It can also move to find a prefix of the pattern which is a suffix of main string. Otherwise, it moves the whole length of the pattern.

BM algorithm processes the pattern and creates different arrays for both heuristics and uses the best of the two heuristics at every step.

Pseudocode

Bad character heuristic

```
badCharacterHeuristic(P, badCharacterArray)
  m := pattern length
  for all entries of badCharacterArray, do
    set all entries to -1
  for all characters of the P, do
    set last position of each character in badCharacterArray.

BM_SEARCH(P, T)
  m: = length of P
  n:= length of T
  call badCharacterHeuristic(P, badCharacterArray)
  shift := 0
  while shift <= (n - m), do
    j := m -1
    while j >= 0 and P[j] = T[shift + j], do
      decrease j by 1
    if j < 0, then
      print the shift as, there is a match
      if shift + m < n, then
        shift:= shift + m – badCharacterArray[T[shift + m]]
      else
        increment shift by 1
    else
      shift := shift + max(1, j-badCharacterArray[T[shift+j]])
```

Good suffix heuristic

```
full_Suffix_Match(shiftArray, borderArray,P)
  m := pattern length
  j := m
  j := m+1
  borderArray[i] := j
  while i > 0, do
    while j <= m AND P[i-1] ≠ P[j-1], do
      if shiftArray[j] = 0, then
        shiftArray[j] := j-i;
      j := borderArray[j];
    decrease i and j by 1
    borderArray[i] := j

 partial_Suffix_Match(shiftArray, borderArray, P)
  m := pattern length
  j := borderArray[0]
  for index of all characters ‘i’ of P, do
    if shiftArray[i] = 0, then
      shiftArray[i] := j
    if i = j then
      j := borderArray[j]

BM_SEARCH(T, P)
  m := pattern length
  n := text size
  for all entries of shiftArray, do
    set all entries to 0
  call full_Suffix_Match(shiftArray, borderArray, P)
  call partial_Suffix_Match(shiftArray, borderArray, P)
  shift := 0
  while shift <= (n - m), do
```

```
j := m -1
while j >= 0 and P[j] = T[shift + j], do
  decrease j by 1
if j < 0, then
  print the shift as, there is a match
  shift := shift + shiftArray[0]
else
  shift := shift + shiftArray[j+1]
```

Time complexity

Best case time complexity is O(n/m) occurs when the character is not present and shift is by m. Worst case time complexity is O(nm) occurs when all the characters of text and pattern are same. It takes O(n+m) for preprocessing.

Space complexity

Θ(m) in order to store shift during preprocessing of the pattern.

3.2 HASHING STRING MATCHING ALGORITHMS

Hashing provides a simple method to avoid a quadratic number of character comparisons in most practical situations. Hashing-based approaches find hash values for the character to match rather than characters. The hashing-based approach saves a large amount of computation as it compares integer values instead of characters. We will discuss Rabin-Karp algorithm one such hashing string searching algorithm.

3.2.1 RABIN-KARP ALGORITHM

The RK string searching algorithm exploits a hash function to speed up the search. For a given pattern p with characters m, the RK string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. Therefore, the algorithm involves two key steps, pre-processing and searching phases. The pre-processing phase generally converts a string into a decimal number. That is, a string of c characters is converted into a string of d decimal numbers (Radix based). Hash for all pattern characters is computed up to m−1, where m is a pattern window that comprises m characters. The pattern p is divided by pre-defined prime number q. Modulus operation is then used to calculate the remainder of pattern p with q. For each shift that ranges from shift (s=0 to n−m), the remainders of pattern and text are compared for matching. If the hash values are not equal, the algorithm will estimate the hash value for next M-character sequence. If the hash values are equal the algorithm will do the brute force comparison with the pattern and M-character sequence. The key to RK performance is the efficient computation of hash values of the contiguous substrings of the text.

Pseudocode

```
RABINKARP_SEARCH (T, P, d, q)
 n = T.length
 m = P.length
 h = $d^{m-1}$ mod q
 p = 0
 $t_0$ = 0
 for i = 1 to m                    // preprocessing
         p = (dp + P[i]) mod q
         $t_0$ = ($dt_0$ + T[i]) mod q
 for s = 0 to n – m                // matching
         if p == $t_s$
                 if P[1..m] = = T[s+1..s+m]
                         print "Pattern occurs with shift", s
         if s < n - m
                 $t_{s+1}$ = (d ($t_s$ – T[s+1]h) + T[s+m+1]) mod q
```

Time complexity

The algorithm takes Θ(m) preprocessing time. The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of T[] match with hash value of P[].

Space complexity

O(1), preprocessing involves hashing of pattern which does not require extra space.

4. EXPERIMENTAL ANALYSIS

All the above algorithms were implemented in python. I used plaintext of 'little women' book from Gutenberg project to implement these algorithms. We have used different patterns, to analyze these algorithms.

Analysis of different algorithms on basis of a pattern which is not found in the text
Pattern= "This eBook is from gutenberg project"

	Brute Force	Knuth Morris Pratt	Boyer Moore	Rabin Karp
Time Taken (in seconds)	0.54091620 s	0.33241892 s	0.04350209 s	0.62255788 s

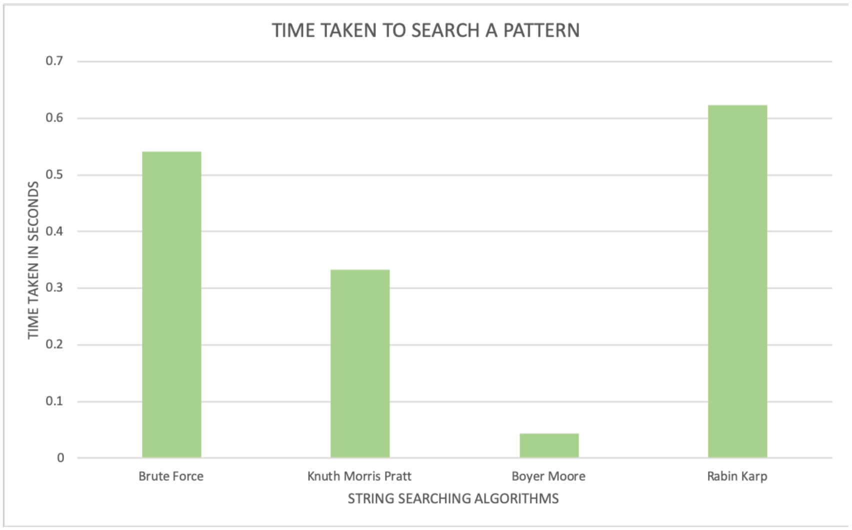


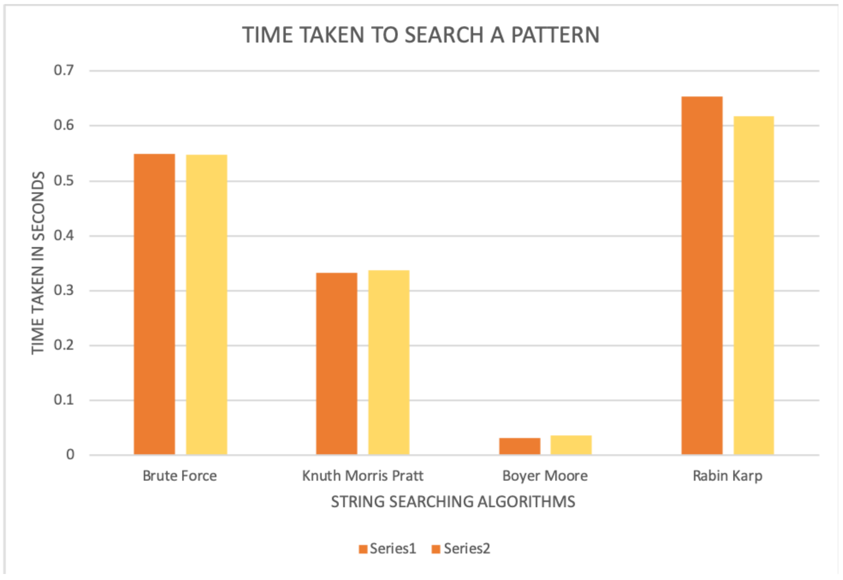
Figure 1. Graph of different algorithms with a pattern outside text

Here the searching is done till the end of the text to find the pattern, since the pattern is not present in the text. Boyer Moore algorithm completes the search faster than other three algorithms.

Analysis of different algorithms on patterns found at start of the text and end of the text
Pattern= “The Project Gutenberg EBook of Little Women, by Louisa May ”
This pattern is found at index 1

Pattern= “subscribe to our email newsletter to hear about new eBooks.”
This pattern is found at index 1032355

	Brute Force	Knuth Morris Pratt	Boyer Moore	Rabin Karp
Time Taken to find the pattern at index 1 (in seconds)	0.54908204 s	0.33291506 s	0.03060889 s	0.65343308 s
Time Taken to find the pattern at index 1032355 (in seconds)	0.54817509 s	0.33637595 s	0.03659701 s	0.61782383 s



Series1: Pattern at index 1 Series 2: Pattern at index 1032355

Figure 2. Graph of different algorithms on patterns at first and last part of text

Searching two patterns both of same size (m=59) at the start of the text and end of the text. We see KMP and Boyer Moore algorithms take more time to find the pattern at index 1032355 than it takes to find the other pattern at index 1. Whereas, Brute Force and Rabin Karp algorithms take less time to find the pattern at index 1032355 than it takes to find the other pattern at index 1. On the whole, Boyer Moore performs the search faster.

Analysis of different algorithms on different lengths of pattern
Short string Pattern= "Yes, Jo"
Pattern of size - 7

Medium string pattern= "Was it all self-pity, loneliness, or low spirits? Or was it the waking up of a sentiment which had bided its time as patiently as its inspirer? Who shall say?"
Pattern of size - 160

Large string Pattern= "It was certainly proposing under difficulties, for even if he had desired to do so, Mr. Bhaer could not go down upon his knees, on account of the mud. Neither could he offer Jo his hand, except figuratively, for both were full. Much less could he indulge in tender remonstrations in the open street, though he was near it. So the only way in which he could express his rapture was to look at her, with an expression which glorified his face to such a degree that there actually seemed to be little rainbows in the drops that sparkled on his beard."
Pattern of size - 550

	Brute Force	Knuth Morris Pratt	Boyer Moore	Rabin Karp
Time Taken to find the short string pattern (in seconds)	0.546636819 s	0.337356091 s	0.142224073 s	0.632728098 s
Time Taken to find the medium string pattern (in seconds)	0.544543028 s	0.332935095 s	0.025352001 s	0.630917311 s
Time Taken to find the Large string pattern (in seconds)	0.628976107 s	0.334483147 s	0.020893097 s	0.637598991 s

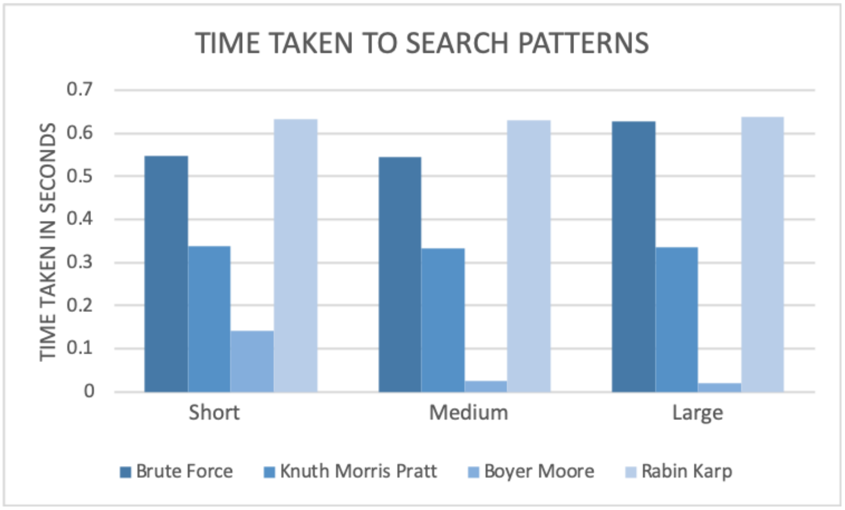


Figure 3. Graph of different algorithms on different lengths of pattern

KMP algorithm finds short length pattern slower than other two length patterns. Brute Force and Rabin Karp algorithms find large length pattern slower and Boyer Moore takes time to find the medium text pattern. Time taken to find various length patterns also depend on the position of those patterns in the text.

Analysis of different algorithms on different lengths of text
Pattern= "CHAPTER"

This pattern is searched in different length texts. We have saved the source file into four size ranges, into full text (size – 1032414), 1/4th text (size - 248321), half text (size – 497170) and 3/4th text (size – 767606). We search the pattern through all these four text files.

Full text shows 47 results, Three-fourth text shows 36 results, half text shows 24 results and one-fourth text shows 12 results.

	Brute Force	Knuth Morris Pratt	Boyer Moore	Rabin Karp
One-fourth text	0.13773298 s	0.08480596 s	0.02750707 s	0.16416717 s
Half text	0.27021789 s	0.16669512 s	0.05885219 s	0.32198286 s
Three-fourth text	0.43485618 s	0.25694895 s	0.09097004 s	0.47952008 s
Full text	0.56038308 s	0.34684300 s	0.11912608 s	0.63914514 s

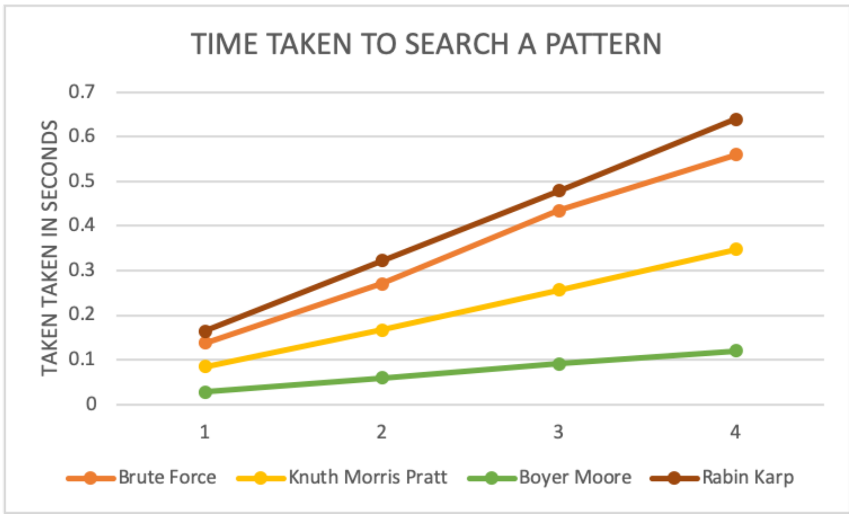


Figure 4. Graph of different algorithms on different lengths of text

Time taken to search the pattern in a text with larger size (full text) obviously takes more time.

In all cases we see Boyer Moore algorithm is the fastest algorithm and finds the pattern in fewer seconds compared to other algorithms. Rabin Karp has a worst-case complexity of O(nm) and so takes more time to search for any pattern in the text.

5. APPLICATIONS

Applications in Information Technology includes Multimedia applications, Network applications, search engines, Bioinformatics.

Multimedia Applications
String searching plays a vital role in information extraction, topic tracking, question answering etc. In case of Text mining task, used to extract previously unknown information from large quantities of text. With the use of string searching algorithms, one can detect the similarities between texts and is thus used in plagiarism detection technique.

Network Applications
The threats of hacking and intruder attacks constantly exist, security has been made a primary concern in networks at present.

Network Intrusion detection system is a device or software application that monitors a network or system for malicious activity. It inspects all inbound and outbound network activity and identifies suspicious patterns. string searching algorithms can be investigated to verify the packets. So, we can say signature-based intrusion detection and anomaly detection systems can be introduced using string searching algorithms.

Search Engines
Indexing and retrieval methods may fail to retrieve actual information based on user interest through recognition of text, sound, image, or video. String searching plays a vital role in solving such issues because it does not require recognition of each character in the text or content in image or video. Instead, string searching considers the entire input as one pattern to find a match.

Bioinformatics and Forensic
String searching and DNA analyzer are used to in collaboration to find pattern occurrences in perspective to the issues involving genetic sequences and finding DNA patterns. In forensic science string searching algorithms are used to authenticate data with the original data, such as blood and DNA samples for large data.

6. CONCLUSION

In this paper, various types of string-searching algorithms were examined. We can conclude Boyer Moore as the best algorithm. When the pattern is long, it has an ability to skip more characters and has a complexity of O(n/m). KMP and Brute force method obtain similar results but we can notice KMP is always a bit better than brute force. KMP can be the best choice when the length of the pattern is small and when Boyer Moore is not in choice. Rabin Karp produces good results in general but due to collision in hashing, it ends up with bad time complexity. So, Rabin karp's complexity depends on the generated hash values. Its time complexity is always more than KMP, Boyer Moore but with brute force it differs. We can see time complexity of Rabin Karp and brute force are almost similar while searching for a large size pattern in the text. Rabin-Karp is a decent option for some particular cases, for example when the pattern and the text are very small.

REFERENCES

https://core.ac.uk/download/pdf/26834073.pdf
https://arxiv.org/pdf/1401.7416.pdf
https://ieeexplore.ieee.org/abstract/document/8703383/references#references
https://github.com
https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/
https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf
https://www.iosrjournals.org/iosr-jce/papers/Conf.17025-2017/Volume-1/7.%2032-35.pdf?id=7557
https://www.tutorialspoint.com/Boyer-Moore-Algorithm
https://www.javatpoint.com/daa-knuth-morris-pratt-algorithm
Introduction to Algorithms Book by CLRS, Chapter 32