STRING MATCHING

Vallepu Vinay Kumar

1002192970

Introduction

String matching algorithms play a crucial role in various applications, including text-editing programs, DNA sequence analysis, and internet search engines. These algorithms efficiently locate occurrences of specific patterns within a larger body of text, thereby enhancing the responsiveness and performance of the underlying systems. In text-editing programs, string matching algorithms are utilized to find all occurrences of a pattern within a document being edited. This pattern is typically a word or phrase supplied by the user, and efficiently locating it within the document is essential for tasks such as spell checking, grammar correction, and text search. Moreover, string matching algorithms have significant applications in bioinformatics, particularly in searching for patterns within DNA sequences. By efficiently identifying patterns in DNA sequences, these algorithms facilitate tasks such as sequence alignment, gene identification, and motif discovery.

Additionally, internet search engines rely on string matching algorithms to find web pages relevant to user queries. These algorithms efficiently search through vast amounts of web content to identify pages containing the specified keywords or phrases, thereby providing users with relevant search results. Formally, the string-matching problem is defined as follows: Given a text represented as an array $T[1:n]$T[1:n] of length nn and a pattern represented as an array $P[1:m]$P[1:m] of length $m \leq n$m≤n, both consisting of characters drawn from a finite alphabet ΣΣ, the task is to efficiently locate all occurrences of the pattern PP within the text TT. The alphabet ΣΣ represents the set of possible characters that can occur in the text and pattern. For example, ΣΣ could be defined as $\{0,1\}${0,1} for binary strings or $\{a,b,c,\ldots\}${a,b,c,...} for alphanumeric strings.

Efficient string-matching algorithms are essential for improving the performance of various systems and applications that rely on text processing, analysis, and retrieval. By effectively locating patterns within text data, these algorithms enable faster and more accurate information retrieval, analysis, and decision-making processes. In the subsequent sections, we will explore and analyze three prominent string-matching algorithms: Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin-Karp.

There are various algorithms and techniques for string matching, each with its own advantages and use cases. Some common methods include:

1. Naive string-matching algorithm
2. Rabin-Karp Algorithm
3. Knuth-Morris-Pratt (KMP) Algorithm
4. Boyer-Moore Algorithm

Rabin-Karp Algorithm

The Rabin-Karp algorithm, proposed by Rabin and Karp, is a string-matching algorithm known for its practical performance and versatility. It is capable of generalizing to other problems, including two-dimensional pattern matching. The algorithm utilizes elementary number-theoretic concepts, such as modular arithmetic, and is based on the equivalence of two numbers modulo a third number.

Pseudo Code:

```
function rabin_karp_search(text, pattern, prime):
  n = length(text)
  m = length(pattern)
  hash_pattern = hash(pattern, prime)
  hash_text = hash(text[0:m], prime)
  occurrences = []
 for i = 0 to n - m:
    if hash_pattern == hash_text and text[i:i+m] == pattern:
      occurrences.append(i)
    if i < n - m:
      hash_text = (hash_text - text[i] * pow(prime, m - 1)) * prime + text[i + m]
  return occurrences.
function hash(s, prime):
  h = 0
  for each character c in s:
    h = (h * prime + c) % large_prime
return h
```

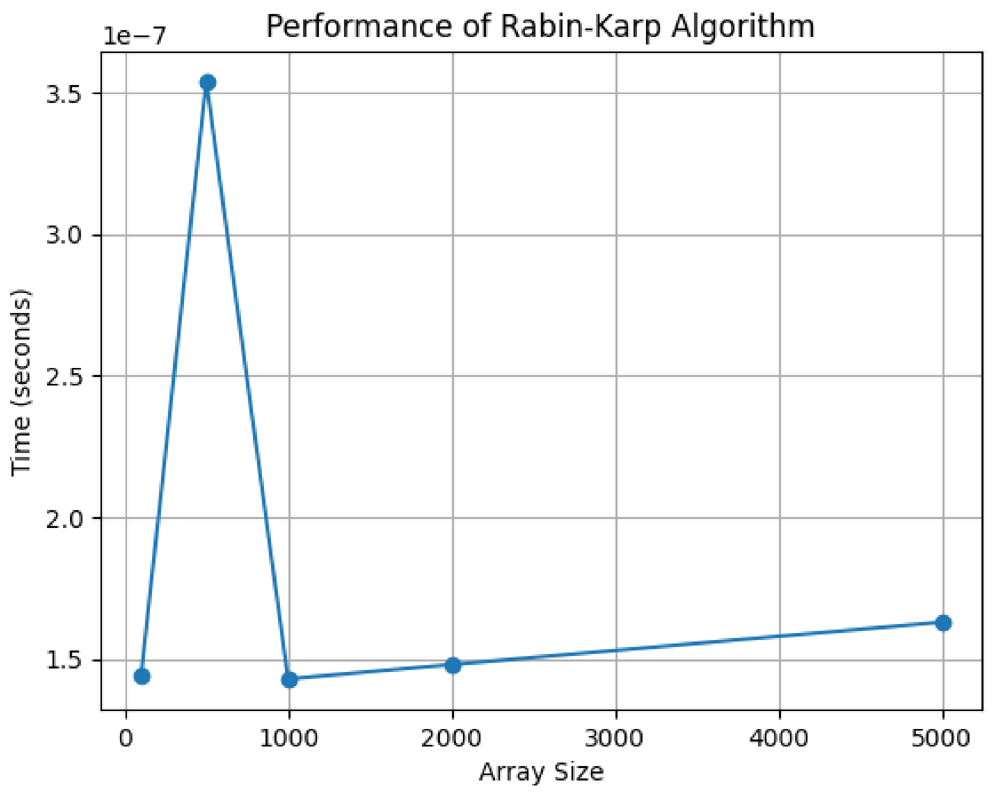
The algorithm aims to find all valid shifts s s where the decimal value of the substring matches the decimal value of the pattern p p. It computes the decimal value p p of the pattern using Horner's rule in time $O(m)$ O(m). Similarly, it computes the decimal value t 0 t 0 of the initial substring of length in the text in time $O(m)$ O(m). By comparing p p with each of the $t\ s$ t s values, the algorithm determines all valid shifts s s in time $O(n)$ O(n).

Time Complexity:

Preprocessing Time: $O(m)$ O(m), where m m is the length of the pattern.

Search Time: $O((n - m + 1) \times m)$ O((n−m+1)×m), where n n is the length of the text and m m is the length of the pattern.

Space Complexity: O(1)



GitHub Link:
https://github.com/vinaykumarvallepu/CSE_5311_Paper/blob/main/rabin_karp_search.ipynb

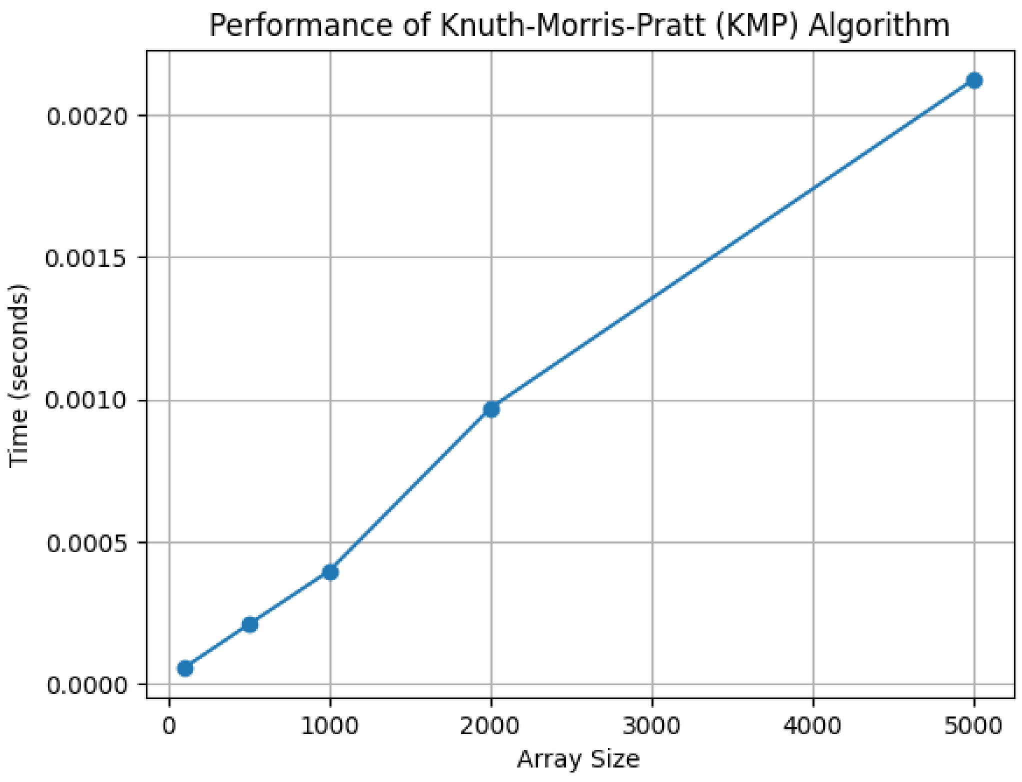
Knuth-Morris-Pratt (KMP) Algorithm:
The Knuth-Morris-Pratt (KMP) algorithm is a string-matching algorithm known for its efficiency in locating occurrences of a pattern within a larger text. It achieves this efficiency by preprocessing the pattern to efficiently skip characters in the text based on previously matched characters.

Pseudo Code:

```
function kmp_search(text, pattern):
  n = length(text)
  m = length(pattern)
  prefix_table = compute_prefix_table(pattern)
  occurrences = []

  j = 0
  for i = 0 to n - 1:
    while j > 0 and text[i] ≠ pattern[j]:
      j = prefix_table[j - 1]
    if text[i] == pattern[j]:
      j++
    if j == m:
      occurrences.append(i - m + 1)
      j = prefix_table[j - 1]
  return occurrences

function compute_prefix_table(pattern):
  m = length(pattern)
  prefix_table = array of size m
  prefix_table[0] = 0
  k = 0

 for i = 1 to m - 1:
    while k > 0 and pattern[i] ≠ pattern[k]:
      k = prefix_table[k - 1]
    if pattern[i] == pattern[k]:
      k++
    prefix_table[i] = k
return prefix_table
```

Algorithm Features:
Prefix Table Preprocessing: KMP preprocesses the pattern to construct a prefix table, also known as the failure function, which provides information about the longest proper prefix that is also a suffix for each prefix of the pattern. Efficient Search: During the search phase, KMP uses the prefix table to avoid unnecessary character comparisons, leading to faster search times. Linear Time Complexity: KMP has a time complexity of $O(n + m)$ O(n+m), where n n is the length of the text and m m is the length of the pattern. Use Cases Information retrieval systems Bioinformatics (DNA sequence analysis) Natural language processing (text processing tasks)

Complexity:
Time Complexity: $O(n + m)$ O(n+m)
Space Complexity: $O(m)$ O(m)



GitHub Link:
https://github.com/vinaykumarvallepu/CSE_5311_Paper/blob/main/kmp_search.ipynb

Boyer-Moore Algorithm

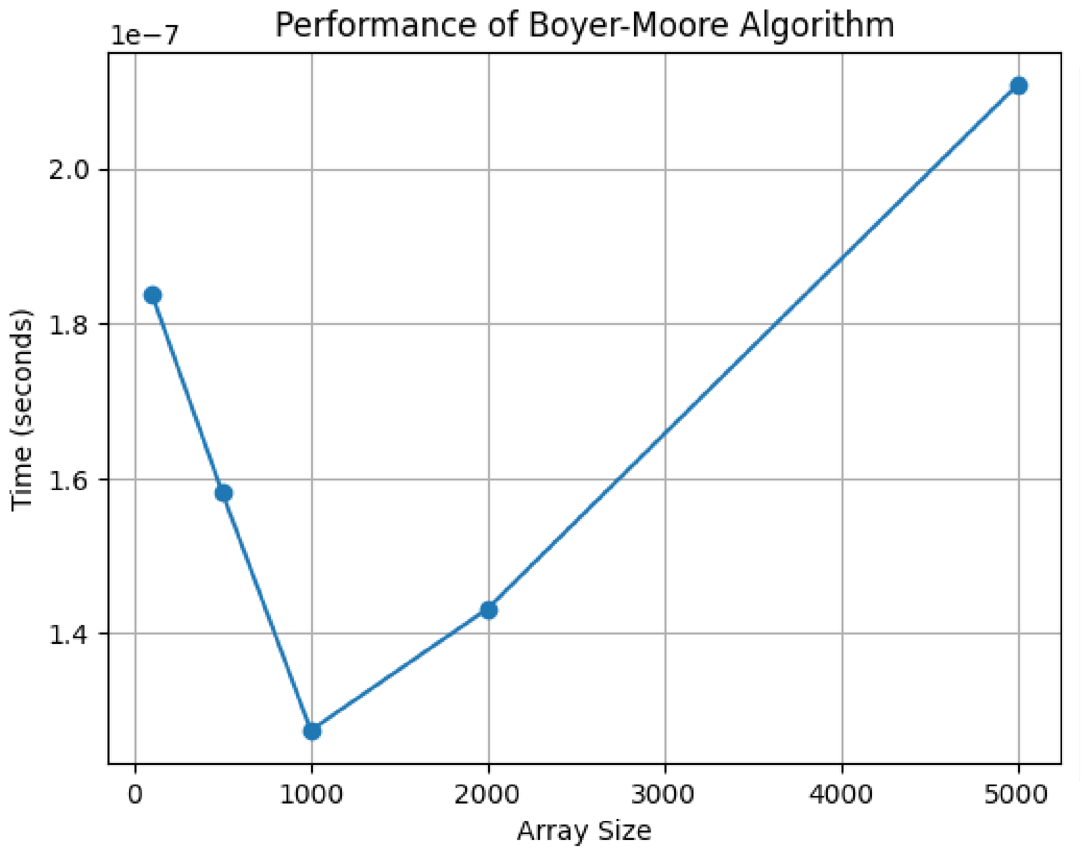
The Boyer-Moore algorithm is a string-matching algorithm known for its practical efficiency, particularly for texts with frequent mismatches. It preprocesses the pattern and uses two shift rules—the bad character rule and the good suffix rule—to skip ahead in the text when a mismatch occurs.

Pseudo Code:

```
function kmp_search(text, pattern):
  n = length(text)
  m = length(pattern)
  prefix_table = compute_prefix_table(pattern)
  occurrences = []

  j = 0
  for i = 0 to n - 1:
    while j > 0 and text[i] ≠ pattern[j]:
      j = prefix_table[j - 1]
    if text[i] == pattern[j]:
      j++
    if j == m:
      occurrences.append(i - m + 1)
      j = prefix_table[j - 1]

  return occurrences

function compute_prefix_table(pattern):
  m = length(pattern)
  prefix_table = array of size m
  prefix_table[0] = 0
  k = 0

  for i = 1 to m - 1:
    while k > 0 and pattern[i] ≠ pattern[k]:
      k = prefix_table[k - 1]
    if pattern[i] == pattern[k]:
      k++
    prefix_table[i] = k

  return prefix_table
```

Algorithm Features:
Preprocessing: Boyer-Moore preprocesses the pattern to compute the bad character rule and the good suffix rule, which determine the shifts to be made in case of mismatches.
Practical Efficiency: It is efficient in practice, especially for texts with frequent mismatches.
Worst-case Complexity: Boyer-Moore has a worst-case time complexity of $O(nm)$O(nm), where nn is the length of the text and mm is the length of the pattern.

Complexity
Time Complexity: $O(nm)$ O(nm) (worst-case), $O(n/m)$ O(n/m) (best-case)
Space Complexity: $O(1)$ O(1)



GitHub Link:
https://github.com/vinaykumarvallepu/CSE_5311_Paper/blob/main/boyer_moore_search.ipynb