e-ISSN:2582-7219

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH IN SCIENCE, ENGINEERING AND TECHNOLOGY

Volume 7, Issue 10, October 2024

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 7.521

© 2024 IJMRSET | Volume 7, Issue 10, October 2024| DOI: 10.15680/IJMRSET.2024.0710054

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

String Pattern Matching Algorithms and Comparison

Priya S, Nikitha G, Chaitra A R

Assistant Professor, Department of Computer Science & Applications, The Oxford College of Science, Bangalore, India

MSc Student, Department of Computer Science & Applications, The Oxford College of Science, Bangalore, India

MSc Student, Department of Computer Science & Applications, The Oxford College of Science, Bangalore, India

ABSTRACT: String pattern matching is a fundamental problem in computer science, with applications ranging from text processing and data retrieval to bioinformatics and cybersecurity. This paper explores various string pattern matching algorithms, focusing on their methodologies, efficiency, and use cases. Through a comparative analysis, we can get the between these algorithms in terms of preprocessing time, space complexity, and performance in different contexts. The choice of an optimal algorithm depends on factors such as pattern length, alphabet size, and the nature of the text, making the understanding of these algorithms crucial for efficient string processing in diverse applications.

KEY ALGORITHMS: Knuth-Morris-Pratt (KMP), Boyer-Moore, Horspool algorithm.

I. INTRODUCTION

String pattern matching is a fundamental concept in computer science and programming, used to find occurrences of a substring (pattern) within a larger string (text).

Pattern matching is the process of checking a perceived sequence of string for the presence of the constituents of some pattern. The patterns generally have the form sequences of pattern matching include outputting the locations of a within a string sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other string sequence (i.e., search and replace).

These are the algorithms used for pattern searching

1. Horspool's String Search Algorithm
2. Knuth–Morris–Pratt algorithm
3. Boyer–Moore string search algorithm

1. Basic Concepts

- **String**: A sequence of characters. For example, "hello world" is a string.
- **Pattern**: A substring or a specific sequence of characters that you want to find within the main string.
- **Regular Expressions**: Regular expressions (regex) are a powerful tool for pattern matching that allow you to define complex search patterns using special syntax. They are widely used in programming languages, text editors, and command-line tools.

II. ALGORITHMS

1. Horspool Algorithm:

Horspool's algorithm is an optimization of the Boyer-Moore string search algorithm. It improves model performance by using a simplified version of the Boyer-Moore method. The main idea is to reduce the number of comparisons required by cutting parts of the text that do not match the pattern.

Preprocessing: Horspool algorithm uses only Bad Character Table

Bad Character table: The Horspool algorithm depends on the bad character table of the Boyer-Moore algorithm. This algorithm creates a table that maps each character in the pattern to its last occurrence index. If a mismatch occurs, the table tells how far to shift the pattern based on the mismatched character.

© 2024 IJMRSET | Volume 7, Issue 10, October 2024| DOI: 10.15680/IJMRSET.2024.0710054

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

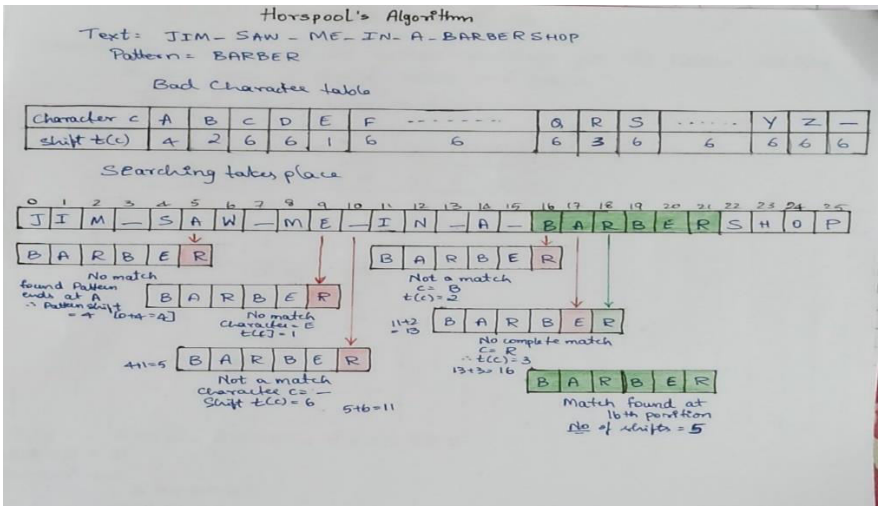
Algorithm:

```
FUNCTION Horspool(text, pattern):
   m = length(pattern)
   n = length(text)

   if m = 0 then
      return 0
   for each character c in alphabet:
      shift[c] = m
   end for

      for i from 0 to m - 2:
      shift[pattern[i]] = m - 1 - i
   end for

   s = 0
   while s <= n - m:
      j = m - 1
            while j >= 0 and pattern[j] = text[s + j]:
         j = j - 1
      if j < 0 then
         return s
            s = s + shift[text[s + m - 1]]
   end while
      return -1
```

Time Complexity: **Space Complexity**: O(k)

Best: O(n/m)
Average: O(n)
Worst: O(n·m)
  n = length(text)
  m = length(pattern)

Example:



© 2024 IJMRSET | Volume 7, Issue 10, October 2024| DOI: 10.15680/IJMRSET.2024.0710054

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

2. Knuth–Morris–Pratt algorithm: The Knuth-Morris-Pratt (KMP) algorithm for string pattern matching. It efficiently searches for occurrences of a "pattern" string within a "text" string by preprocessing the pattern to allow the search to skip over portions of the text that have already been matched. The task is to print all indexes of occurrences of pattern string in the text string.

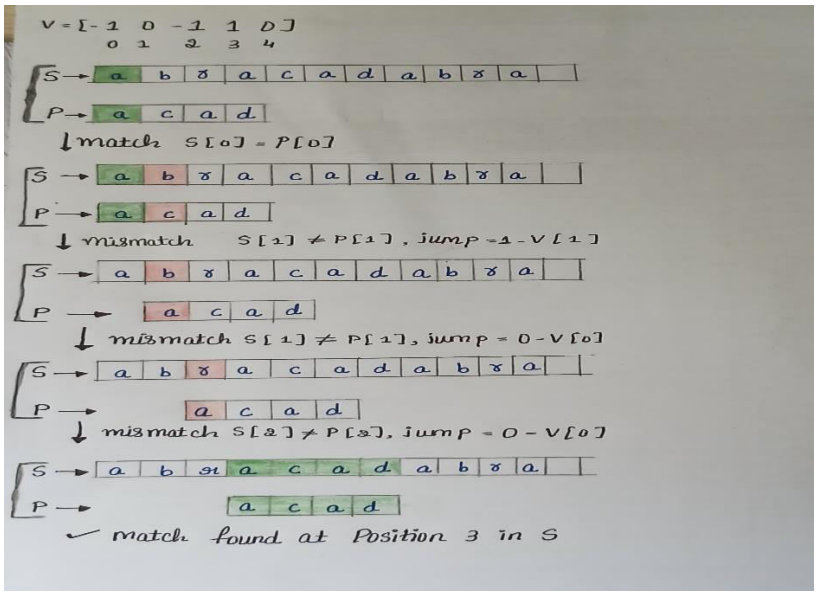
Algorithm:

```
define variables:
int j ← 0                (the position of the current character in S)
int k ← 0                (the position of the current character in W)
an array of integers, T  (the table, computed elsewhere)
let nP ← 0
while j < length(S) do
if W[k] = S[j] then
let j ← j + 1
let k ← k + 1
if k = length(W) then
let P[nP] ← j – k
 nP ← nP + 1
let k ← T[k]
else
let k ← T[k]
if k < 0 then
let j ← j + 1
let k ← k + 1
```

Time Complexity
Preprocessing Time : O(m)
Running Time : O(m+n)
n = length(text)
m = length(pattern)

Space Complexity :- O(m).

Example :-



© 2024 IJMRSET | Volume 7, Issue 10, October 2024| DOI: 10.15680/IJMRSET.2024.0710054

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

3. Boyer–Moore string search algorithm:

It is a particularly efficient string searching algorithm. The algorithm preprocesses the target string (key) that is being searched for, but not the string being searched in. Generally the algorithm gets faster as the key being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it is searching. The Boyer Moore algorithm does preprocessing and 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. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

preprocessing : The algorithm preprocesses the pattern to create two tables:

- **Bad Character Table**: This tells how far to shift the pattern when a mis match occurs based on the last occurance of the mismatched character in the pattern.
- **Good Suffix Table**: This indicated how to shift the pattern when a mismatch happens, based on the matched suffix of the pattern.

Algorithm:

```
FUNCTION BoyerMoore(text, pattern):
  n = length(text)
  m = length(pattern)
  if m = 0 then
    return 0  // match found at index 0
  for each character c in alphabet:
    badcharshift[c] = -1  // default shift is -1 (not found)
  end for
    for i from 0 to m - 1:
    badcharshift[pattern[i]] = i
  end for
  // start searching
  s = 0
  while s <= n - m:
    j = m - 1
    while j >= 0 and pattern[j] = text[s + j]:
      j = j - 1
    if j < 0 then
      return s  // match found at index s
        s = s + max(1, j - badcharshift[text[s + j]])
  return -1  // no match found
```

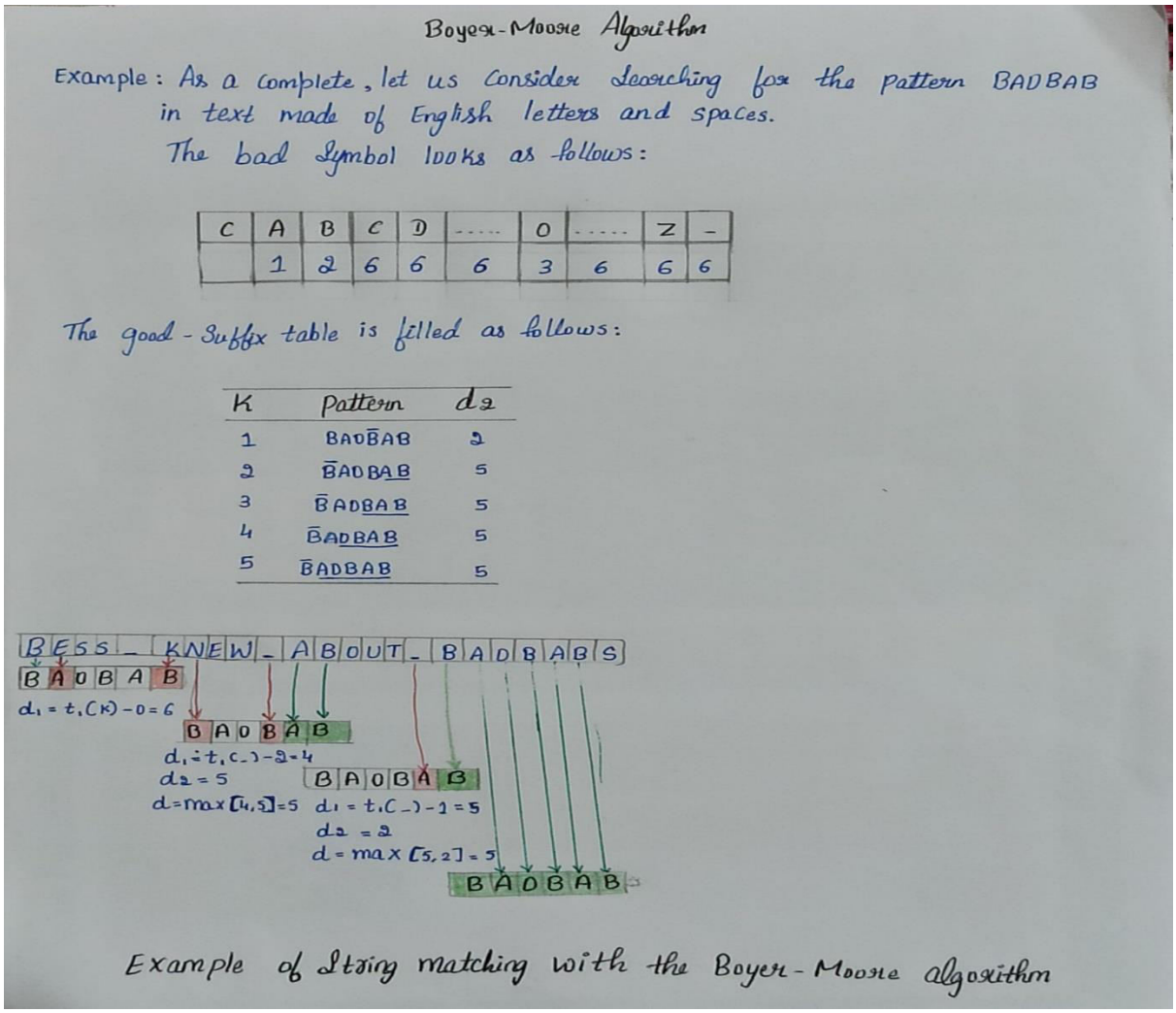
© 2024 IJMRSET | Volume 7, Issue 10, October 2024| DOI: 10.15680/IJMRSET.2024.0710054

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Example:
Let us consider searching for the pattern "BAOBAB" in a text "BESS_KNEW_ABOUT_BAOBABS" made of English letters and spaces(_)



III. COMPARISION FOR THE BEST ALGORITHM

Comparing these four algorithms:

Algorithm	Space Complexity	Time Complexity	
		Best Case	Worst Case
Horspool Algorithm	O (k)	O(n/m)	O(n*m)
Knuth–Morris–Pratt algorithm	O (m)	O(m)	O(n+m)
Boyer–Moore string search algorithm	O (m)	O(n/m)	O(n*m)

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 7.521| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

1. Horspool's String Search Algorithm
This algorithm uses a bad character heuristic, which means it shifts the pattern along the text based on the mismatched character.

- **Time Complexity:** O(n*m) in the worst case, but O(n/m) on average.
- **Efficiency:** More efficient than the naive approach, especially for large texts.
- **Pros**: Better average-case performance than naive, uses a bad-character heuristic.
- **Cons**: Not as fast as some other advanced algorithms.

2. Knuth–Morris–Pratt algorithm
This algorithm uses a lookup table to store the longest prefix that is also a suffix for each substring of the pattern.

- **Time Complexity:** O(n + m), making it more efficient than the previous two algorithms.
- **Efficiency:** Very efficient, especially for large texts and patterns.
- **Pros:** Efficient for all cases due to preprocessing of the pattern, no backtracking.
- **Cons:** More complex to implement due to the preprocessing step.

3. Boyer–Moore string search algorithm
This algorithm uses a combination of bad character and good suffix heuristics to shift the pattern along the text.

- **Time Complexity:** O(n/m) on average, making it one of the most efficient string search algorithms.
- **Efficiency:** Very efficient, especially for large texts and patterns.
- **Pros**: Very efficient for large alphabets and long patterns due to its use of two heuristics (bad character and good suffix).
- **Cons**: Requires more preprocessing time and space.

Which is the best?
The best string searching algorithm depends on the specific context and requirements of your application, such as the size of the text and pattern, frequency of searches, and the types of characters involved.

Based on the time complexities and efficiencies, we can consider **Boyer–Moore string search algorithm** as the best option. It has an average time complexity of O(n/m), making it very efficient for large texts and patterns. However, it's worth noting that the Knuth–Morris–Pratt algorithm is also very efficient and may be a better choice in certain situations.

REFERENCES

1. introduction-to-the-design-and-analysis-of-algorithms-3rd-ed.-levitin-2011-10-09
2. References papers - String matching algorithms and their comparison (June 2020)
3. "An Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. This book covers basic algorithms including the naive pattern matching approach.
4. L. Otero-Cerdeira, F. J. Rodriguez-Martinez, and A. GÃ¸smez-Rodriguez,"Ontology matching: A literature review," Expert Systems with Applications, vol. 42, no. 2, pp. 949-971, 2015.
5. S. Faro and T. Lecroq, "The Exact Online String Matching Problem:A Review of the Most Recent Results," (in English), Acm Computing.Surveys, vol. 45, no. 2, Feb 2013.
6. G. F. Ahmed and N. Khare, " Hardware based String Matching Algorithms: A Survey," International Journal of Computer Applications, vol.88,no. 11, pp. 16-19, 2014.
7. Gonçalves, J., & Barbosa, L. (2020). A Comprehensive Survey on String Matching Algorithms and Their Applications.
8. Zhang, Y., Wang, Y., & Yu, Y. (2020). An Improved Knuth-Morris-Pratt Algorithm for String Matching.
9. Müller, H., & Böhm, K. (2020). Approximate String Matching Algorithms: A Review and Performance Evaluation.
10. Alam, M. S., & Sultana, A. (2021). A Comparative Analysis of String Matching Algorithms: KMP, Boyer-Moore, and Rabin-Karp.
11. Zhao, W., & Zhang, Y. (2021). Fast String Matching with Multiple Patterns Using a Suffix Tree.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com