Multiple Pattern String Matching Methodologies: A Comparative Analysis

Zeeshan Ahmed Khan1, R.K Pateriya2

1Research Scholar
2Assistant Professor

Department of Computer Science & Engineering
Maulana Azad National Institute of Technology, Bhopal, 462051, INDIA

1zeeshan.khan12@gmail.com
2rk_pateriya@yahoo.co.in

Abstract—String matching algorithms in software applications like virus scanners (anti-virus) or intrusion detection systems is stressed for improving data security over the internet. String-matching techniques are used for sequence analysis, gene finding, evolutionary biology studies and analysis of protein expression. Other fields such as Music Technology, Computational Linguistics, Artificial Intelligence, Artificial Vision, have been using string matching algorithm as their integral part of theoretical and practical tools. There are various problems in string matching appeared as a result of such continuous, exhaustive use, which in turn were promptly solved by the computer scientists. The more practical solutions to the real world problems can be solved by the multiple pattern string matching algorithms. String Matching Algorithms like Aho-Corasick, Commentz-Walter, Bit parallel, Rabin-Karp, Wu-Manber etc. are to be focused in this paper. Aho-Corasick algorithm is based on finite state machines (automata). Commentz Walter algorithm is based on the idea of Knutt-Morris-Pratt and finite state machines. Bit parallel algorithm like shift-or makes use of wide machine words (CPU registers) to parallelize the work. Rabin-Karp uses hashing to find any one of a set of pattern strings in a text. Wu-Manber looking text in blocks instead of one by one character combining idea of Aho-Corasick and Boyer-Moore. Each algorithm has certain advantages and disadvantages. This paper presents the comparative analysis of various multiple pattern string matching algorithms. A comparison of Aho-Corasick, Commentz-Walter, Bit-Parallel(Shift-OR), Rabin-Karp, Wu-Manber etc. type of string matching algorithms is presented on different parameters.

Index Terms— String matching, Aho-Corasick, Commentz Walter, Bit parallel, Rabin-Karp, Wu-Manber, FSM.

I. INTRODUCTION

String matching is a technique to find out pattern from given text. Let \( \Sigma \) be an alphabet. Elements of \( \Sigma \) are called symbols or characters. For example, if \( \Sigma = \{a,b\} \), then abab is a string over \( \Sigma \). String ab, ba, aba, bab are set of patterns over given string. The patterns strings are denoted by \( P[1,...,m] \). The text string is denoted by \( T[1,...,n] \). If \( P \) occurs with shift \( s \) in \( T \), then we call \( s \) a valid shift; otherwise, we call \( s \) an invalid shift. The string matching problem is the problem of finding all valid shifts with which a given pattern \( P \) occurs in a given text \( T \). Figure 1 shows this definition[1].

These problems find applications in information retrieval systems like search engines, bioinformatics, computer security, and DNA sequence analysis.

“When I got to the bottom of the stairs, I saw that the stairs were high and steep. I took a deep breath and began to climb the stairs. When I reached the top of the stairs I looked around me.”

Input Text String and Pattern

“When I got to the bottom of the stairs, I saw that the stairs were high and steep. I took a deep breath and began to climb the stairs. When I reached the top of the stairs I looked around me.”

Output

Two examples of multiple-pattern matching problems:

English:

Text: CPM_annual_conference_announce
Set of Symbols: \{ announce, annual, annually \}

DNA:

Text: AGATACGATATATAC
Set of Symbols: \{ ATATATA, TATAT, ACGATAT \}

String matching algorithms can be categorized basically in two types exact and approximate string matching algorithms. Exact string matching algorithms are further divided into single and multiple pattern string matching. Each category can be applied to various areas of application. Multiple pattern matching algorithms have more practical and realistic applications like.
DNA sequencing[10][11], intrusion detection/prevention system (IDS/IPS)[13][14][15], data mining, search engines[12], detecting plagiarism[8][9], Music Technology[7] etc. Multiple pattern string matching algorithms are mainly discussed in this paper.

String matching algorithms are also used in Intrusion Detection Systems (IDSs) that have become widely recognized as powerful tools for identifying, deterring and deflecting malicious attacks over the network. Essential to almost every intrusion detection system is the ability to search through packets and identify content that matches known attacks. Space and time efficient string matching algorithms are therefore important for identifying these packets at line rate[4].

The paper is organized as follows. Section II survey on the most significant algorithms for multiple string matching algorithms like Aho-Corasick, Commentz Walter, Bit-Parallel(Shift-OR), Rabin-Karp, Wu Manber. Section III comparative study of various algorithms described. Finally Section IV concludes comparative survey by putting strong points in favour of some algorithm in particular case of usage.

## II. VARIOUS MULTI-PATTERN STRING MATCHING ALGORITHMS

### A. Aho-Corasick Multi-Pattern Algorithm

The Aho-Corasick algorithm[1] (AC Algorithm) was proposed in 1975 and remains, to this day, one of the most effective pattern matching algorithms when matching patterns sets. Initially, the AC algorithm combines all the patterns in a set into a syntax tree which is then converted into a non-deterministic automaton (NFA) and, finally, into a deterministic automaton (DFA) as shown in Figure 2. The resulting FSM is then used to process the text one character at a time, performing one state transition for every text character. Whenever the FSM reaches designated “final” states that correspond to the identification of a pattern match as in Figure 3.

The pseudocode for the matching phase of the algorithm is given below:

```plaintext
1: procedure AC(y, n, q0)
   /*Input:
    \( y \) -- array of bytes representing the text input
    \( n \) -- integer representing the text length
    \( q0 \) -- initial state
   */
2:   state \( \rightarrow q0 \)
3:   for i = 1 to n do  //Matching
4:      whileleg(state, y[i]) = faildo
5:         //while g(state, y[i]) is undefined
6:            state \( \rightarrow f\) (state)
7:    end while
8:   state \( \rightarrow g\) (state, y[i])
9:   if ostate) then
10:      output //This an accepting state, i.e. state \( \in A \)
11:     end if
12: end procedure
```

The AC algorithm has the significant advantage that every text character is examined only once, i.e. the lookup cost is \( O(N) \) where \( N \) the length of the text, regardless of the number of patterns or their length. A major disadvantage of the AC algorithm is the high memory cost required to store the transition

---

**Table 1**

<table>
<thead>
<tr>
<th>State(i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure(i)</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**Table 2**

<table>
<thead>
<tr>
<th>(i)</th>
<th>Output(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>{ row }</td>
</tr>
<tr>
<td>6</td>
<td>{ arrows }</td>
</tr>
<tr>
<td>9</td>
<td>{ row }</td>
</tr>
<tr>
<td>12</td>
<td>{ sun }</td>
</tr>
<tr>
<td>17</td>
<td>{ under }</td>
</tr>
</tbody>
</table>

---

www.ijsrp.org
rules of the underlying deterministic finite automaton.

**B. Commentz-Walter Multi-Pattern Algorithm**

The popular GNU fgrep utility uses the Commentz-Walter algorithm for multiple string search[2]. Commentz-Walter algorithm combines the Boyer-Moore technique with the Aho-Corasick algorithm. In preprocessing stage, differing from Aho-Corasick algorithm, Commentz-Walter algorithm constructs a converse state machine from the patterns to be matched. Each pattern to be matched adds states to the machine, starting from right side and going to the first character of the pattern, and combining the same node.

In searching stage, Commentz-Walter algorithm uses the idea of Boyer-Moore algorithm. The length of matching window is the minimum pattern length. In matching window, Commentz-Walter scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses a precomputed shift table (CWSHIFT[\(i\)]) to shift the window to the right.

For pattern set \{search, ear, arch, chart\}, Figure 4 shows the Commentz-Walter state machine and the goto function. Table 3 shows the output function and Table 4 shows the shift distance.

If the text string is "stremeadnssearchof", Figure 5 shows the searching process. For seeing clearly, we draw the key characters in red and green, and with the same intention in Figure 4 and Figure 5.

**Fig.4. Commentz-Walter state machine and goto function**

<table>
<thead>
<tr>
<th>nodes</th>
<th>4</th>
<th>6</th>
<th>9</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>{arch}</td>
<td>{search}</td>
<td>{ear}</td>
<td>{chart}</td>
</tr>
</tbody>
</table>

**TABLE 3. COMMENTZ-WALTER OUTPUT FUNCTION**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>e</th>
<th>h</th>
<th>r</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>others</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**TABLE 4. CWSHIFT TABLE. CWSHIFT[0a]=1, CWSHIFT[0e]=2, ..., CWSHIFT[9r]=2, ETC.**

In Figure 5, the length of matching window is 3. On step 1, when input is ‘r’, the state is 7, when input is ‘t’, the state is 0, CWSHIFT[\(7t\)]=3, the text shift 3 characters. On step 2, when input is ‘a’, the state is 0, CWSHIFT[\(0a\)]=1, the text shift 1 character. On step 7, when input is ‘r’, the state is 7, when input’s ‘a’, test state 8, when input is ‘r’, the state is 9, the output is ‘ear’, when input is ‘s’, the state is 0, CWSHIFT[\(9s\)]=2, the text shift 2 characters.

A multiple string matching algorithm that compares from the end of the pattern, like Boyer-Moore, using a finite state machine, like Aho-Corasick. In computer science, the Commentz-Walter algorithm is a string searching algorithm invented by Beate Commentz-Walter. Like the Aho–Corasick string matching algorithm, it can search for multiple patterns at once. It combines ideas from Aho–Corasick with the fast matching of the Boyer–Moore string search algorithm. For a text of length \(n\) and maximum pattern length of \(L\), its worst-case running time is \(O(nL)\), though the average case is often much better.

The pseudocode for the matching phase of the algorithm is given below:

```
1: procedure CW(y, n, m, p, root)
    /* Input: 
    . y ← array of \(n\) bytes representing the text input
    . n ← integer representing the text length
    . m ← array of keyword lengths
    . p ← number of keywords
    . root ← root node of the trie */
2: v ← root //The current node
3: i ← \(\min\{m[0], m[1], ..., m[p – 1]\}\)
   // \(i\) points to the current position in \(y\)
4: j ← 0 //\(j\) indicates depth of the current node \(v\)
5: while \(i < n\) do //Matching
6: while \(v\) has child \(v_0\) labeled \(y[i – j]\) do
7: \(v ← v_0\)
8: \(j ← j + 1\)
9: if out(\(v\)) = \(v\) then
10: output \(i – j\)
11: end if
```

**Fig.5. Commentz-Walter searching process.**
C. Wu-Manber (WM) Multi-Pattern Algorithm

Wu-Manber algorithm is a high-performance multi-pattern matching algorithm based on Boyer-Moore algorithm. It only uses the bad-character shift, and considers the characters from the text in blocks of size B instead of one by one, expanding the effect of bad-character shift. Wu-Manber algorithm also uses the hashing table to index the patterns in the actual matching phrase, thus saving a lot of time. The best performance of Wu-Manber algorithm is O(B n/m). The running time of the Wu-Manber algorithm does not increase in proportion to the size of the pattern set. The performance of the Wu-Manber is dependent on the minimum length of the pattern. In practice, Wu-Manber algorithm is isoneof the best performance in average case [4].

In preprocessing stage, Wu-Manber algorithm builds three tables, a SHIFT table, a HASH table, and a PREFIX table. The SHIFT Table is implemented, but not exactly the same, to the regular shift table in a Boyer-Moore type algorithm. It is used to determine how many characters in the text can be shifted (skipped) when the text is scanned. The HASH and PREFIX tables are used when the shift value is 0. They are used to determine which patterns are a candidate for the matching and to verify the match. For pattern set \{search, hear, arch, chart\}, Table 6 shows the SHIFT table 5 and Table 6 shows the HASH table for B=2.

<table>
<thead>
<tr>
<th>BC</th>
<th>ar</th>
<th>ch</th>
<th>ea</th>
<th>ha</th>
<th>he</th>
<th>rc</th>
<th>se</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**TABLE 5**

<table>
<thead>
<tr>
<th>a</th>
<th>r</th>
<th>O</th>
<th>s</th>
<th>e</th>
<th>a</th>
<th>r</th>
<th>c</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>e</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>h</td>
<td>a</td>
<td>r</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 6**

If the text string is “strcmatecadnsearchof”, Figure 6 shows the searching process.

Wu-Manber combines the advantages of both Aho-Corasick and can index the document once as well as skip over words in a jump table. Wu-Manber algorithm performs well under scenarios with small or medium number of patterns, but it does not perform well under scenarios with large number of patterns, because the likelihood of shifting the text sliding window is decreased quickly with the increase of the number of patterns.

D. Rabin-Karp Multi-Pattern Algorithm

In computer science, the Rabin–Karp algorithm is a string searching algorithm created by Michael O. Rabin and Richard M. Karp in 1987 that uses hashing to find any one of a set of pattern strings in a text. [5] The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

Consider an M-character sequence as an M-digit number in base b, where b is the number of letters in the alphabet. The text subsequence \{t[i .. i+M-1]\} is mapped to the number.

\[ x(i) = t[i]b^{M-1} + t[i+1]b^{M-2} + ... + t[i+M-1] \]

Furthermore, given \(x(i)\) we can compute \(x(i+1)\) for the next subsequence \{t[i+1 .. i+M]\} in constant time, as follows:

\[ x(i+1) = t[i+1]b^{M-1} + t[i+2]b^{M-2} + ... + t[i+M] \]

\[ x(i+1) = x(i).b \quad \text{// Shift left one digit} \]

\[ - t[i].b^{M} \quad \text{// Subtract leftmost digit} \]

\[ + t[i+M] \quad \text{// Add new rightmost digit} \]

In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.

Let’s say that our alphabet consists of 10 letters. Our alphabet = a, b, c, d, e, f, g, h, i, j Let’s say that “a” corresponds to 1, “b” corresponds to 2 and so on. The hash value for string “cax” would be:-

www.ijsrp.org
3*100 + 1*10 + 8*1 = 318

If M is large, then the resulting value (~bM) will be enormous. For this reason, we hash the value by taking it mod a prime number q. The mod function is particularly useful in this case due to several of its inherent properties:

- [(x mod q) + (y mod q)] mod q = (x+y) mod q
- (x mod q) mod q = x mod q

For these reasons:

h(i) = ((t[i].b^{M-1} mod q) + (t[i+1].b^{M-2} mod q) + ... + (t[i+M-1] mod q)) mod q

h(i+1) = (h(i).b mod q)/Shift left one digit
-\{t[i]\}. b^M mod q/\Subtract leftmost digit
+\{t[i+M] mod q\}/Add new rightmost digit
mod q

For text of length n and p patterns of combined length m, its average and best case running time is O(n+m) in space O(m). In contrast, the Aho-Corasick string matching algorithm has asymptotic worst-time complexity O(n+m) in space O(p), but its worst-case time is O(nm). In contrast, the Aho-Corasick string matching algorithm has asymptotic worst-time complexity O(n+m) in space O(m).

1: Procedure RabinKarpSet(string s[1..n], set of string subs, m):
2: set hsubs := emptySet
3: for each sub in subs
4: insert hash(sub[1..m]) into hsubs
5: hs := hash(s[1..m])
6: for i from 1 to n
7: if hs ∈ hsubs and s[i..i+m-1] ∈ subs
8: return i
9: hs := hash(s[i+1..i+m])
10: return not found

Rabin-Karp is good for plagiarism[8][9], because it can deal with multiple pattern matching. It is not faster than brute force matching in theory, but in practice its complexity is O(n+m). With a good hashing function it can be quite effective and it’s easy to implement.

E. Bit-Parallel(Shift OR) Multi-Pattern Algorithm

Bit-parallelism is a technique introduced by Baeza-Yates and Gonnet in which takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic (suffix) automata.[6] in Figure 7 and Table 7 and 8.

Text= “hhello”
Pattern = { “hello”, “world”}

The Bit Vectors are set in the following manner.

The pseudocode for the matching phase of the algorithm is given below:

1. procedure Shift-Or(P,m,T,n)
2. for all i ∈Σ do
3. B[i] = 1^w;
4. for i = 0 to m -1 do
5. B[P[i]] = B[P[i]] & 1^{w-i-1}1;
6. D = 1^w; mm = 0^{w-mm}; i = 0;
7. while i < n do
8. D = (D & mm) ≠ mm then
9. if ((D&mm)) ≠ mm then
10. Pattern detected beginning at T[i];

![Fig 7. The Automaton recognizing the set of patterns is shown](image)
The Rabin-Karp algorithm achieves an average case running time of \(O(m+n)\) by using hashing. The worst case running time is still \(O(mn)\) however. The Rabin-Karp algorithm exploits the fact that if two strings are equal, their hash values are also equal. Rabin-Karp is simple and can be easily extended to two-dimensional pattern matching.

The bit-parallel algorithms are more efficient that other string matching algorithms for small and long patterns respectively. Their running time decreases as the pattern length increases and they produce similar running times in all cases with the exception of the binary alphabet.

### COMPARISON OF VARIOUS MULTIPLE PATTERN STRING MATCHING ALGORITHM

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Aho-Corasick</th>
<th>Rabin-Karp</th>
<th>Bit-Parallel (Shift OR)</th>
<th>Commentz Walter</th>
<th>Wu-Manber</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Time Complexity</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Sub-linear</td>
</tr>
<tr>
<td>2 Search Type</td>
<td>Prefix</td>
<td>Prefix</td>
<td>Prefix</td>
<td>Prefix</td>
<td>Suffix</td>
</tr>
<tr>
<td>3 Key Ideas</td>
<td>Finite automaton that tracks the partial prefix match.</td>
<td>Compare the text and the patterns from their hash functions</td>
<td>Bit-parallelism and (q)-gram for prefix matching.</td>
<td>it uses is to shift by as much as determined by the longest proper prefix of the pattern</td>
<td>Determine the shift distance from a block of characters in the suffix of the search window.</td>
</tr>
<tr>
<td>4 Approach</td>
<td>Automaton on-based</td>
<td>Hashing-based</td>
<td>Bit-parallelism based</td>
<td>Automaton + Heuristic based</td>
<td>Heuristics based</td>
</tr>
</tbody>
</table>

### REFERENCES


www.ijsrp.org


