An Exploration of the Efficiency of String Matching Algorithms

Vennila Santhanam,
Assistant Professor, Department of Computer Science,
Auxilium College (Autonomous), Vellore. Tamil Nadu, India.
Email - jvennilasanthanam@gmail.com

Abstract: *String searching is a process that refers to the problem of finding all the occurrences of a particular string in the given text. String matching algorithms plays a vital role in finding such appropriate match in limited duration. String matching algorithms are used in many areas such as Spell Checkers, Spam Filters, Intrusion Detection System, Search Engines, Plagiarism Detection, Bioinformatics, Digital Forensics and Information Retrieval Systems, Computational molecular biology. The World Wide Web provides settings in which efficient string matching algorithms are necessary. In this study the researchers review most of the String matching algorithms. It is a class of string algorithms that tries to find the occurrences and location of one or several strings (also called as patterns) are found within a larger string or text.*

Key Words: *String matching, Patterns, Occurrences.*

1. INTRODUCTION:

Let Σ be an alphabet set. The most basic example of string searching is where both the pattern and searched text are sequence of elements of Σ.

Kinds of String searching

String searching may involve a very long string, called the Haystack or it may be a very short string, called the Needle. The ultimate goal of string searching is to find one or more occurrences of the needle string existing within the haystack.

For the strings that represent natural language, the user might have to find all the occurrences of a 'word' despite it having any alternate spellings, prefixes or suffixes and so on.

Another more complex method of search is regular expression searching, where the user constructs a pattern of characters or another symbols. In this method the match to the pattern should fulfill the search.

Basic Classification of Search Algorithms

The various algorithms can be classified by the number of patterns that are used by it..

Single pattern algorithms

Let m be the length of the pattern, n be the length of the searchable text and $k = |\Sigma|$ be the size of the alphabet.

Algorithms using a finite set of patterns

- Aho–Corasick string matching algorithm – It is an extension of Knuth-Morris-Pratt method.
- Commentz-Walter algorithm – It is an extension of Boyer-Moore method.
- Set-BOM – It is an extension of Backward Oracle Matching.
- Rabin–Karp string search algorithm

Algorithms using an infinite number of patterns

The patterns are represented usually by a regular grammar or regular expression.

Other classifications

Other methods for classifications are also possible, that follow different approaches

Classification of the string matching algorithms based on their matching strategy are as follows:

- Match the prefix first (Knuth-Morris-Pratt, Shift-And, Aho-Corasick)
- Match the suffix first (Boyer-Moore and variants, Commentz-Walter)
- Match the best factor first (BNDM, BOM, Set-BOM)
- Other strategy (Naive, Rabin-Karp)

Naïve string search

A simple but inefficient way to see where one string occurs inside another is to check each part it could be, one by one, to see if it is present. So first the user has to check if there is a copy of the needle in the first character of the haystack; if not, the user looks to see if there is a copy of the needle that starts at the second character of the haystack; if not, then the user looks at the third character, and so forth. Normally the user has to look at one or two characters for each wrong position to see that it is a wrong position. Therefore in the average case, it takes $O(n + m)$ steps, where n is the length of the haystack and m is the length of the needle. In the worst case, searching for a string like 'aaaab'- needle in a string like 'aaaaaaaaab' - haystack, it takes a time period of $O(nm)$.

NAIVE_STRING_MATCHER (T, P)

```
Step 1: n ← length [T]
Step 2: m ← length [P]
Step 3: for s ← 0 to n - m do
Step 4: if P[1 . . m] = T[s +1 . . s + m]
Step 5: then return valid shift s
```

The naïve string-matching procedure can be depicted graphically as sliding a pattern P[1 . . m] over the text T[1 . . n] and make a note when all the characters in the pattern match the corresponding characters in the text.

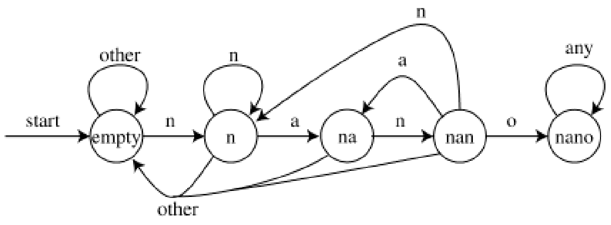

Figure 1: **Finite state automaton based search**

Here, backtracking is avoided by constructing a Deterministic Finite Automaton (DFA) which recognizes the stored strings. Here the researchers construct a DFA using the subset construction method. The usage is easy rather than construction. For example, the DFA shown in figure 1 is used to recognize the word ‘nano’. This approach can be used to search the regular expressions.

Knuth–Morris–Pratt (KMP)

KMP is a method for computing a DFA that recognizes inputs with the string to search for as a suffix. Boyer–Moore starts searching from the end of the needle, so that it can usually jump ahead a whole needle-length at each step. Baeza–Yates keeps track of whether the previous *j* characters were a prefix of the search string. Therefore it is adaptable to fuzzy string searching. The bitap algorithm is an application of Baeza–Yates' approach.

Bitap Algorithm

The **bitap algorithm** is also called as the **shift-or**, **shift-and** or **Baeza-Yates–Gonnet** algorithm. It is an approximate string matching algorithm. The algorithm gives details of whether a given text contains a substring which is "approximately equal" to a given pattern, where approximate equality is defined in terms of Levenshtein distance. If the substring and pattern are within a given distance *k* of each other, then the algorithm considers them equal. The algorithm computes a set of bitmasks containing one bit for each element of the pattern.

Input: Pattern with m characters
Output: Failure function f for P[i . . j]

BITAP ALGORITHM

```
Step 1:  m := length(pattern)
Step 2  if m == 0
Step 3: return text
   /* Initialize the bit array R. */
Step 4: R := new array[m+1] of bit, initially all 0
Step 5: R[0] = 1
Step 6: for i = 0; i < length(text); i += 1:
      /* Update the bit array. */
 Step 7:  for k = m; k >= 1; k -= 1:
 Step 8:   R[k] = R[k-1] & (text[i] == pattern[k-1])
 Step 9: if R[m]:
 Step 10: return (text+i - m) + 1
 Step 11: return nil
```

KNUTH-MORRIS-PRATT FAILURE (P)

```
Step 1: i ← 1
Step 2: j ← 0
Step 3: f(0) ← 0
Step 4: while i < m do
Step 5: if P[j] = P[i]
Step 6: f(i) ← j +1
Step 7: i ← i +1
Step 8: j ← j + 1
Step 9: else if j > 0
Step 10: j ← f(j - 1)
Step 11: else
Step 12: f(i) ← 0
Step 13: i ← i +1
```

The failure function f for P, which maps j to the length of the longest prefix of P which in turn is a suffix of P[1 . . j], encodes substrings inside the pattern itself repeatedly.

Boyer-Moore Algorithm

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in general applications. For example, in text editors and commands substitutions, the algorithm works quickly when the alphabet is moderately sized and the pattern is relatively long. The algorithm scans the characters of the pattern from right to left by beginning with the rightmost character. While testing a possible placement of pattern P against text T, if c is not contained anywhere in P, then shift the pattern P completely past T[i]. Otherwise, shift P until an occurrence of character c in P gets aligned with T[i].

Input: Text with n characters and Pattern with m characters

Output: Index of the first substring of T matching P

BOYER_MOORE_MATCHER (T, P)

```
Step 1: Compute function last
Step 2: i ← m-1
Step 3: j ← m-1
Step 4: Repeat
Step 5: If P[j] = T[i] then
Step 6: if j=0 then
Step 7: return i // we have a match
Step 8: else 11: i ← i -1
Step 9: j ← j -1 13: else
Step 10: i ← i + m - Min(j, 1 + last[T[i]])
Step 11: j ← m -1
Step 12: until i > n -1
Step 13: Return "no match"
```

The computation of the last function takes O (m+|∑|) time and actual search takes O(mn) time. Therefore the worst case running time of Boyer-Moore algorithm is O(nm + |∑|), where ∑ is the input alphabet.

Fuzzy Search

Some search methods, are designed to find a 'closeness' score between the search string and the text rather than a 'match/non-match'. Such searches are sometimes called 'fuzzy' searches.

ANALYSIS OF DIFFERENT STRING MATCHING ALGORITHMS:

Here the researchers analyze various string matching algorithm with their processing time taken to match the pattern.

For long patterns and Σ, Boyer-Moore algorithm shows much better efficiency compared to other string matching algorithms. The algorithm makes successive comparisons from right to left. When a mismatch occurs, it proposes a value by which shift is increased without skipping any valid shift.

An efficient algorithm for string matching based on repetition factors was developed by Galil and Seiferas. This algorithm has linear running time complexity and requires only O(1) storage beyond P and T. C.

The Bitap algorithm performs approximate string matching based on Levenshtein distance between strings. The algorithm requires much lesser preprocessing and can uses mostly bitwise operations, making the algorithm extremely fast.

Aho-Corasick algorithm can perform multiple pattern matching in a text in parallel achieving linear running time.

Polymorphic String Matching Combination of more than one string matching algorithm (example KMP and Boyer-Moore fusion) can be used to provide a better functional algorithm with decreased space and time complexity.

TABLE 1: Analysis of algorithms using online tools.

S. No.	Algorithm	Preprocessing time	Matching time
1	Naïve string search algorithm	0 (no preprocessing)	Θ(nm)

2	Rabin–Karp string search algorithm	Θ(m)	average Θ(n + m), worst Θ((n−m)m)
3	Knuth–Morris–Pratt algorithm	Θ(m)	Θ(n)
4	Boyer–Moore string search algorithm	Θ(m + k)	best - Ω(n/m), worst - O(mn)
5	Bitap algorithm	Θ(m + k)	O(mn)
6	Two-way string-matching algorithm	Θ(m)	O(n+m)
7	BNDM (Backward Non-Deterministic Dawg Matching)	O(m)	O(n)
8	BOM (Backward Oracle Matching)	O(m)	O(n)
9	AhoCorasick	None	O(N+M +Z)
10	CommentZ Walter	None	O(N+M+Z)+ O(MN)
11	Finite state automaton based search	O(m)	O(n)

APPLICATIONS OF STRING MATCHING ALGORITHMS:

- Text editors, search engines and digital libraries need to perform pattern matching in a text or database. Most text editors use direct implementation of Boyer-Moore algorithm to implement the find/replace command.
- String matching algorithms are widely used in DNA sequencing, finding close mutation, searching antimicrobial structures, developing local data warehouses for DNA, genes and proteins.
- Network Intrusion Detection System and computer virus detection systems incorporate use of string matching algorithms of packets against signatures. Multiple patterns from a virus database can be matched in parallel and compiled for automation that are stored for later use.
- Unix Command fgrep introduced in Version 7 UNIX uses Aho-Corasick algorithm to search a set of fixed strings.
- Approximate string matching algorithms are used in modern music search technique to retrieve musical note from musical database.
- Algorithms like DUDE used in combinatorial denoising uses string matching algorithms to come up with distribution of occurrences of its context.

CONCLUSION:

The researchers have analyzed various kinds of string matching algorithms. It is also analyzed that KMP algorithm is relatively easier to implement, because it never needs to move backwards in the input sequence and requires extra space, Rabin Karp algorithm is used to detect plagiarism and requires additional space for matching. Brute Force algorithm does not require preprocessing of the text or the pattern but the problem is to that it is very slow and also it rarely produces efficient result. AhoCorasick algorithm is useful for multi pattern matching. CommentZ-Walter algorithm takes more time to produce the result, The Boyer Moore algorithm is comparatively fast on large strings and avoids lots of unnecessary comparisons by significantly comparing pattern relative to text and the best case running complexity is sub linear.

REFERENCES:

1. Alsmadi I., Nuser M., (2012): String Matching Evaluation Methods for DNA Comparisons, *International Journal of Advanced Science and Technology*, Vol.47.
2. Amir A., Lewenstein M., and Porat E., Faster Algorithms for String Matching with K-Mismatches, *Journal of Algorithms* 502004257-275.
3. Gomaa N.H., Fahmy A.A.,(2012): Short Answer Grading using String Similarity and Corpus-Based Similarity, *International Journal of Advanced Computer Science and Applications*, Vol 3,No.11.
4. Hussain I., Kausar S., Hussain L., and Asifkhan M., (2013): Improved Approach for Exact Pattern Matching, *International Journal of Computer Science Issues*, Vol.10, Issue 3, No.1.
5. Jain P., Pandey S., (November 2012): Comparative Study on Text Pattern Matching for Heterogeneous System, *International Journal of Computer Science and Engineering Technology*, ISSN: 2229- 3345, Vol.3 No.11.

6. Knuth D.E., Morris J.H., and Pratt V.R., (1997): Fast Pattern Matching in Strings, *Journal of Computing*, Vol.6, No.2.
7. Singla N., Garg D., (January 2012): String Matching Algorithms and their Applicability in various Applications, *International Journal of Soft Computing and Engineering*, ISSN: 2231-2307, Volume I, Issue-6.
8. Vidanagamachchi S.M., Dewasurendra S.D., Ragal B.G., Niranjan M., CommentZ Walter: Any Better than AhoCorasick for Peptide Sequence Identification, *International Journal of Research in Computer Science*, eISSN:2249-8265, Vol 2, Issue 6 2012 PP 33-37.
9. Jump up^ *Hume; Sunday (1991). "Fast String Searching". Software: Practice and Experience. 21 (11): 1221–1248. doi:10.1002/spe.4380211105*
10. Graham A. Stephen (1994). *String Searching algorithms* World Scientific.
11. Alberto Apostolico, Zvi Galil (1997). *Pattern Matching Algorithms* Oxford University Press.
12. http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E 2%80%93Pratt_algorithm..
13. https://en.wikipedia.org/wiki/String_searching_algorithm
14. https://en.wikipedia.org/wiki/String_searching_algorithm
15. http://www.idc-online.com/technical_references/pdfs/information_technology/String_Matching_Algorithms.pdf
16. https://en.wikipedia.org/wiki/Bitap_algorithm