International Journal of Computer Science and Engineering Communications
Volume.6, Issue.1 (2018): Page.1780-1787
www.ijcsec.com

An Elementary Algorithm for Pattern Matching

Nidhi Bansal
Assistant Professor,
Department of Computer Science,
Hindustan University, Chennai, India
goelnidhi8@gmail.com

Abstract—*A string matching algorithm aims to find one or several occurrences of a string within another. String matching is a classical problem in computer science. Our approach presents an elementary and efficient algorithm. First, we find some index values of pattern of length m from text T, the algorithm returns the position of the first character of the desired substring in the text. In second phase it matches whether the substring at this index value matches the actual pattern P. The algorithm works in linear time, if the number of occurrences of the pattern in a string is very less.*

Keywords: *Pattern matching, Time complexity, String length*

Citation: Nidhi Bansal.(2018). An Elementary Algorithm for Pattern Matching. *International Journal of Computer Science and Engineering Communications,* 6(1), 1780-1787. Article ID 6117801787.

Copyright © 2018 Nidhi Bansal., This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

I. INTRODUCTION

Strings are a collection of patterns of different type. Many patterns are found within a larger string or text. There are algorithms called as String searching algorithms or string matching algorithms used to find one, several or all occurrences of a defined string (pattern) in a larger string (typically a text). Let Σ be an alphabet which is usually a finite set of the pattern and the text to be searched as vectors. The Σ may be a usual human alphabet (for example, the letters A through Z in the Latin alphabet). Other applications may use binary alphabet (Σ = {0,1}) or *DNA alphabet* (Σ = A,C,G,T}) in bioinformatics [1].We assume that the text is an array T[1..n] of length n and that the pattern is an array of length P[1..m] of length m and that m<=n. The character arrays T and P are often called strings of characters. We say that pattern P occurs with shift s in text T (or equivalently that the pattern P occurs beginning at position s+1 in text T) if 0<=s<=n-m and T[s+1….s+m] = P[1..m]. If P occurs with shift s in T then we calls a valid shift otherwise we call s an invalid shift. The string matching algorithm is the problem of

finding all valid shift with which a pattern P occurs in given text T [2].There are various types and classifications of the string matching. There are two types of string matching Exact string matching and Approximate string matching. The search to be done on exact occurrence of the pattern comes underneath the category of exact string matching. Approximate string matching allows inaccurate searching acceptance founded on specific applications. Based on the number of patterns, string matching has two classifications: Single Pattern string matching and Multiple Pattern string matching. In Single Pattern string matching a single pattern is to be searched in the text whereas in multiple pattern string matching multiple patterns are searched in the text. Based on the order of searching string matching have four classifications i.e. left to right matching, right to left matching, specific order matching and no order matching [3]. The string matching problem has a lot of different applications in multiple areas [4]. First, an adapted and efficient algorithm of this problem can aid to enhance the responsiveness of text-editing software. Other applications in information technology include web search engines, spam filters, natural language processing, computational biology (search of particular pattern in DNA sequence), and feature detection in digital images.

A string search algorithm takes a text T of length n and a pattern P of length m as the input. The text is then scanned using a window that has length equal to the size of the pattern. The leftmost ends of the pattern and window are aligned. The brute force method works by comparing each character of the pattern with that of the text and in case of a mismatch the pattern is shifted by one position to the right. Other existing algorithms generally work in two phases: the pre-processing phase and the matching phase. The pre-processing phase is used to determine the number of positions by which the pattern needs to be shifted in case of a mismatch in the matching phase. The main goal of string matching algorithms is to increase efficiency by reducing the number of comparisons and increase the length of shifts in case of a mismatch. In this paper, we present we present an elementary and efficient algorithm. Our algorithm works in two phases: the pre-processing phase and the matching phase. The pre-processing phase will determine the index value according to which pattern is shifted. The pattern might appear zero time, once, a few times or many times in the text depending to its length. We can observe and analyse the effectiveness of algorithm by measuring its processing time and comparing to other algorithms. The rest of the paper is organized as follows. Section 2 discusses about some previously existing algorithms. Section 3 presents the proposed algorithm. Section 4 presents the experimental results of the proposed algorithm and finally the paper is concluded in Section 5.

II. LITERATURE REVIEW

A. Brute Force Algorithm

The very basic and conventional string matching strategy is Brute Force Algorithm. It is also known as naïve algorithm. It achieves the character comparisons among the input text and pattern from left to right manner. If the mismatch occurs or a complete match then it shifts one step to the right [4]. In this algorithm there is no pre-processing stage and it needs the constant extra space. The main advantage of this algorithm it is very easy to implement but it is very slow compared to other algorithms [5]. The time complexity of this algorithm is O (mn) and the expected number of character comparison is 2^n.

B. Boyer Moore Horspool Algorithm

The Boyer Moore Horspool algorithm or Horspool algorithm is used to find the substring in the input text or large document collections. In 1980, this algorithm was published by Nigel

Horspool. In the Boyer-Moore- Horspool algorithm, it compares the text character T[i] with the last character p[j] of the pattern. If they match, then it compares the previous characters of the text with corresponding characters in the pattern consecutively right to left, until to detect either a frequency of the pattern or a mismatch on a text character [6]. The algorithm uses the bad character heuristic and the good suffix heuristic to determine the pattern shift in case of mismatch of a pattern character. The Boyer Moore algorithm works with the property that the speed of algorithm is directly proportional to the length of the pattern. However the algorithm suffers from the phenomenon that it tends to work inefficiently on small alphabets. The skip distance tends to stop growing with the pattern length because substrings re-occur frequently [7]. Also, the pre-processing for the good suffix heuristic is difficult to understand and implement. Furthermore, it suffers from the need for very large tables or state machines and thus requires extra space [7]. It also requires extra time for processing the pattern.

C. Rabin- Karp Algorithm

Rabin-Karp Algorithm is the simplest string searching algorithm. This algorithm was developed by Michael O. Rabin and Richard M. Karp in 1987. This algorithm uses the hash function to discover the potential pattern in the input text. For the length of text n and pattern p of mutual length m, its average and best case running time is O (n+m) in space O (p), and also the worst-case time is O (nm) in space O (m)[8]. It is used to discover the hash value of the certain pattern substring and then it discovers the hash value of all possible m length substring of the input text. If the hash value of the pattern and text substring match than it returns the value otherwise next substring value is matched to calculate the string of length m.

D. Knuth-Morris-Pratt Algorithm

The Knuth–Morris–Pratt were developed a linear time string searching algorithm by analysis of the brute force algorithm or naive algorithm. The algorithm was developed in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris and they published it jointly in 1977.The Knuth-Morris-Pratt algorithm moderates the total number of comparisons of the pattern against the input string. A matching time of O(n) is accomplished by evading associations with essentials of 'S' that have earlier been involved in the comparison with some of the specific element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' certainly not occurs[9]. The KMP algorithm makes use of the information gained by previous character comparisons unlike the naïve algorithm. Hence it never needs to move backwards in the text, this makes the algorithm useful for processing large files [10]. However the performance of the KMP algorithm degrades for longer patterns as the possibility of character mismatch increases. As observed, all of the algorithms discussed rely on pre-processing the pattern and using the pre-computed values for searching a string. The Boyer Moore and Knuth–Morris–Pratt algorithm are more effective for searching. However, Boyer Moore algorithm fails to achieve the desired performance for shorter patterns. On the other hand although the KMP algorithm works well for shorter patterns, it is far more complex than the proposed algorithm. The proposed algorithm is easy to understand and implement.

III. PROPOSED WORK

In this algorithm we work in two phases. First phase is the pre-processing phase and second phase is matching phase .In pre-processing phase, we find some index values of substrings of length m from text T and the second phase matches these substrings with actual pattern P. We tried to show the working of two phases by following two algorithms.

A. Phase 1 Pre-Processing Phase

In the first phase that is pre-processing phase we will find the some index values from each substring of length m from text T which matches with first and last character of pattern P and insert these index values into a the queue Q. The approximate running time of pre-processing phase is O (n-m).

```
1. n = length S;
2. m = length P;
3. x <- P[0], y <- P[m-1];
4. For (i=0 to n-m)
5. If (S[i]==x && S[m+i-1]==y)
6. Insert i into the queue Q
7. Return Q
```

B. Phase 2 Matching Phase

Matching phase totally depends on the size of the queue. In this phase, we find the substrings of length m on the basis of index value and compare the pattern with the substrings of size m. If a match occurs, then it is called as successful hit otherwise it will be spurious hit . The approximate running time of matching algorithm O (qm) where q is the size of the queue and m is the length of pattern. Following algorithm shows the matching phase.

```
1. While(Q!= empty)
2. {
3. I= dequeue()
4. K =I;
5. count==0;
6. For(j=0 to m-1)
7. {
8. If(P[j]==S[K])
9. {
10. K=K+1;
11. count = count+1;
12. } // end of if condition
13. else
14. Goto 1
15. }
16. If(count==m)
17. {
18. Print("pattern occur at location ", i);
19. }
20. } // end of while loop
21. Exit
```

IV. Experimental Results

The Figure 1 shows the example problem. Then in figure 2, 3 and 4 we have shown the steps during the search of the Pattern P = a a b of length 3 in a text of length 14. We have given a text T = a a b a a c a a d a a b a d and the queue Q.

A. Phase 1 Pre-Processing Phase

For our example n=14, m=3. For each i the algorithm matches the character at ith index to the first character of pattern. If there is a match then it matches the last character of the pattern with corresponding position in string. If last character also matches then ith index is stored in queue Q. In figure 2, For i=0, value at ith index matches the first character of the pattern. So we will check whether last character also matches the corresponding value. Here last character also matches with corresponding position in string. So we will store ith value i.e. 0 as the first element in queue Q. For i=1 to 11, we do same procedure checking the value at ith index for given example. It is shown in Figure 3 and 4. After pre-processing phase our queue contains 2 elements, it means there may be matching pattern at these two indexes. We will do this in next phase.

B. Phase 2 Matching Phase

We dequeue index values from the queue until queue is empty and match pattern at that index value. We will match the value at corresponding index of string with the first character of pattern till the length of the pattern. If the pattern matches with the pattern at index value it becomes a hit. As seen for our example in Figure 5 and 6, there are two hits and both hits are successful, no spurious hit. So for our given example, we find the matching pattern at two places i.e. there are two occurrences of pattern in text.

String T=

0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	a	b	a	a	c	a	a	d	a	a	b	a	d

Pattern P=

0	1	2
a	a	b

Fig 1 String Length T=14, Pattern Length P=3

For i=0

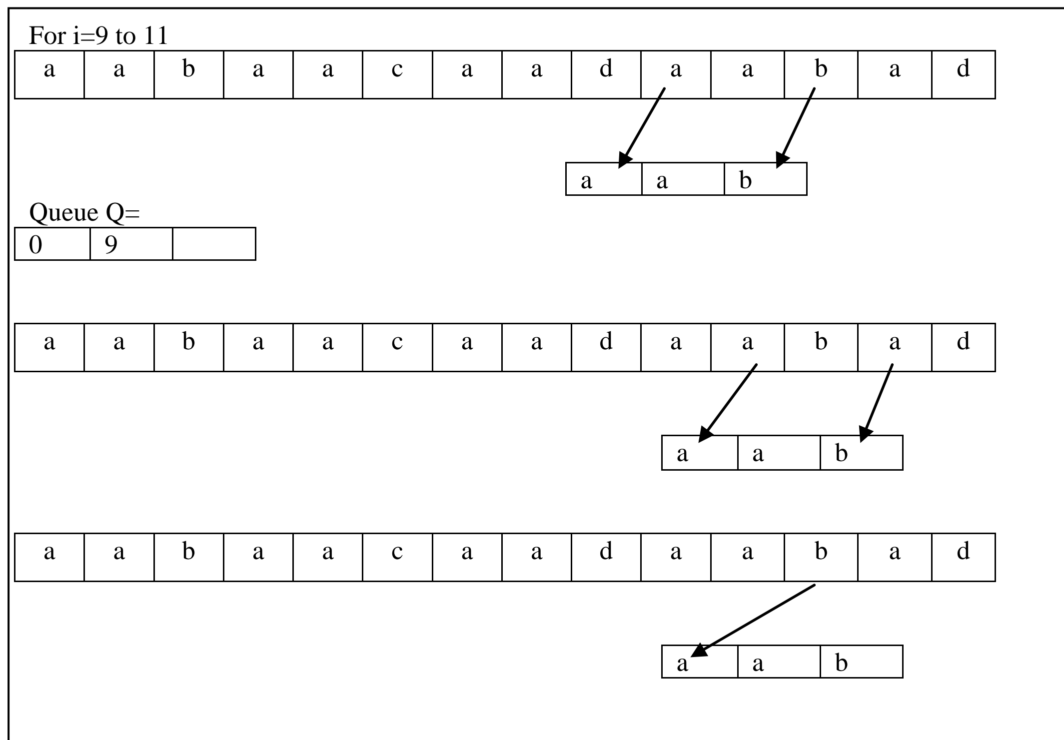
0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	a	b	a	a	c	a	a	d	a	a	b	a	d

a	a	b

Queue Q=

0		

Fig 2 Pre-Processing for i=0, Match the value at i=0 of T with i=0 of P



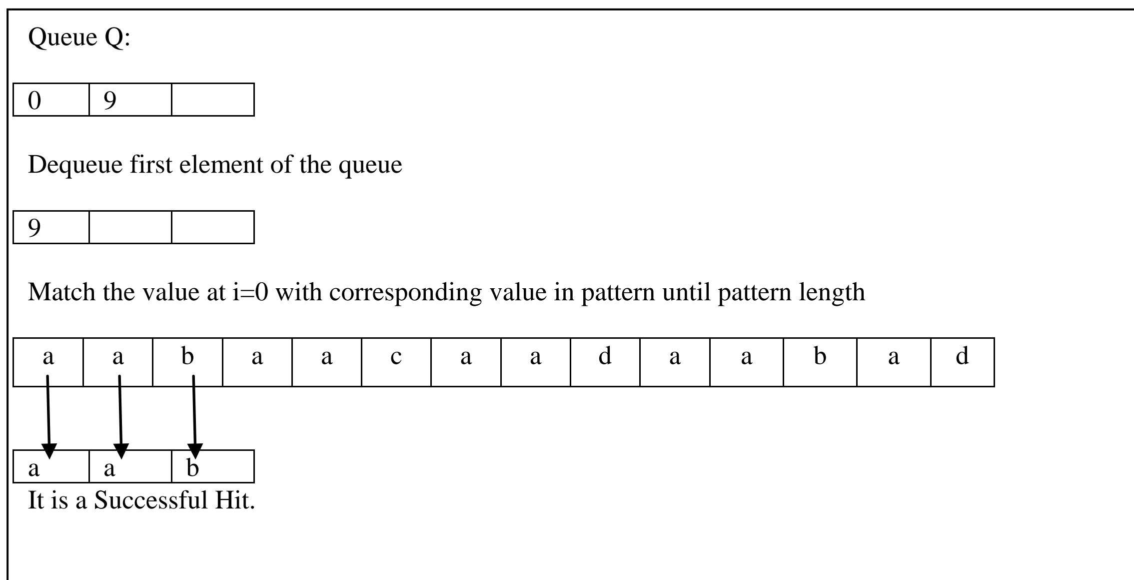


Fig 3: Matching Phase for first element of the queue.

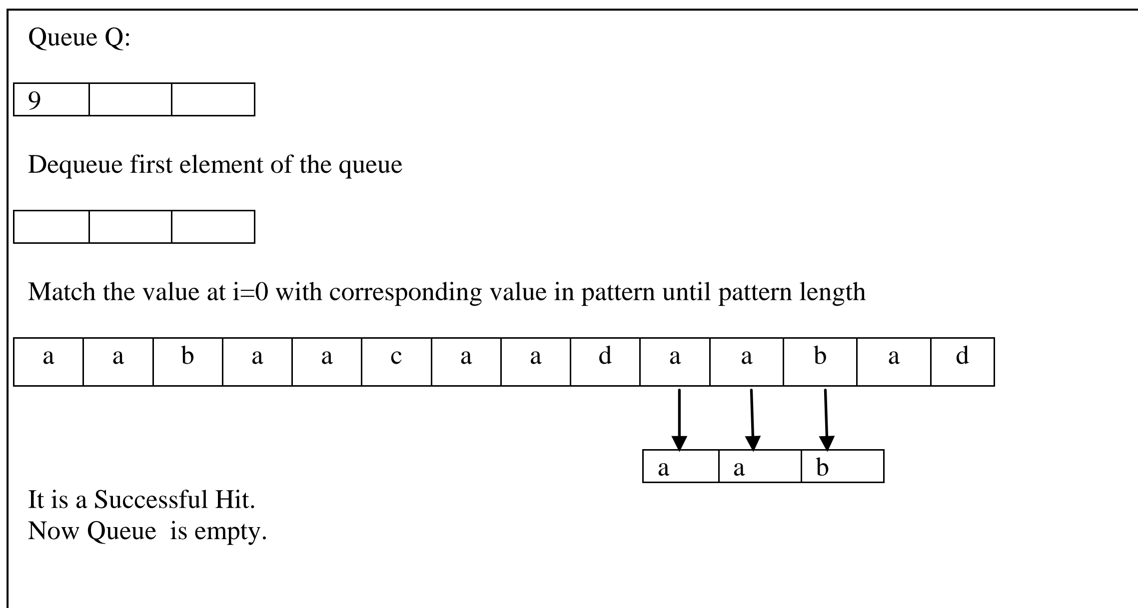


Fig 4: Matching Phase for next element of the queue.

Algorithms	Time Complexity For Pre-Processing	Time Complexity For Matching
Brute force algorithm	No pre-processing	O((n-m+1)*m)
Boyer Moore	O(m + |Σ|)	O(n)
Rabin karp	O(m)	O(n+m)
Knuth-morris-pratt	O(m)	O(n+m)
Proposed algorithm	O(n-m)	O(q*m)

Table 1: Comparison of Time Complexities Of Pattern Matching Algorithms

C. Comparison of Time Complexities of Pattern Matching Algorithms

We compared time complexity for both the phases for our proposed algorithm with some existing algorithms as shown in Table 1.For our example time taken for pre-processing phase is O(n-m) = O(11). But for matching phase time taken is only O(q*m) = O(6) which is better than time taken by Rabin Karp and KMP i.e. O(17). So proposed algorithm provides best results for matching phase as compared to some existing algorithms.

V. Conclusions

We evaluated the performance of some existing algorithms with our proposed algorithm. Our proposed algorithm is a linear time pattern matching algorithm. It could be better to other algorithms in some constraints. The running time of this algorithm depends on size of the queue i.e. the number of index value in the queue which shows that the number of expected patterns in text T. If there is only one element in the queue then maximum time taken by matching phase would be O (m), where m is the size of pattern. So if the no of occurrences of pattern in string is less this algorithm gives best result compared to Rabin-Karp and KMP algorithm.

References

[1] Wikipedia The free Encyclopedia en.wikipedia.org/wiki/String_searching_algorithm

[2] Thomas H Corman, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein “Introduction to Algorithms- String matching”, EEE Edition, 2nd Edition, Page no 906-907.

[3] Alberto Apostolico and ZviGalil, “ Pattern Matching Algorithms” Published in Oxford University Press, USA, 1st edition, May 29, 1997.

[4] Rahul M, Diwate B, Satish J, Alaspurkar, “ A. Study of Different Algorithms for Pattern Matching”, International Journal of Advanced Research in Computer Science and Software Engineering. 2013; 3(3):1–8.

[5] Al-Mazroi A, Rashid NA, “A Fast Hybrid Algorithm for the Exact String Matching Problem” American Journal of Engineering and Applied Sciences. 2011; 4(1):102–07.

[6] Boyer RS, Moore JS, “A fast string searching algorithm”, Communication of the ACM. 1977; 20(10):762–72.

[7] http://www.cs.utexas.edu/~moore/best-ideas/string-searching/index.html

[8] Shivaji SK, Prabhudeva S, “Plagiarism Detection by using Karp-Rabin and String Matching Algorithm Together”, International Journal of Computer Applications. 2015; 116(23):1–5.

[9] Gope AP, Behera RN, “A Novel Pattern Matching Algorithm in Genome Sequence Analysis”, (IJCSIT) International Journal of Computer Science and Information Technologies.2014; 5(4):5450–57.

[10] http://cs.indstate.edu/~kmandumula/presentation.pdf