International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

An Elegant and Efficient String Matching Algorithm

D. Abhyankar[1], P. Saxena[2]

[1]Lecturer, [2]Reader, School of Computer Science & IT, Devi Ahilya University , Indore (M.P.)

***Abstract*—String matching is a fascinating problem to solve elegantly. We propose an algorithm which solves the string matching in an elegant and efficient way. Our experimental study suggests that algorithm performs efficiently on English texts. Moreover, proposed algorithm is easy to maintain and easy to implement. Both from software engineering point of view and algorithm design point of view algorithm is a salient choice to capture place in software of real world.**

***Keywords*—Algorithm, Pattern Matching, Running Time, String Matching, Text Processing.**

I. Introduction

String matching problem fascinates a lot of computer scientists, and finds applications in word processor and other utility software. String matching algorithms try to find a place where pattern string is found in a large text string. We assume n is the length of the pattern, m ($m \geq n$) is the length of the text and Σ is the size of the alphabet. Here, Σ may be an English alphabet (for instance, the letters A through Z in the English alphabet). Other applications may involve DNA alphabet ($\Sigma = \{A, C, G, T\}$) in bioinformatics. There exist more than twenty five string matching algorithms in the literature [1]; therefore it is not feasible to discuss all of them exhaustively. Table I presents the running time of the key algorithms. Among these algorithms *Boyer–Moore algorithm* is used as a standard benchmark of the practical string matching algorithms. We propose an elegant and efficient algorithm to solve string matching problem and to compete with the algorithms in the literature.

Our string matching algorithm may be used in database and text processing applications. Text editors need a mechanism to search strings in the document. Pattern matching programming languages may use our algorithm to perform string matching. It may assist programs that filter and modify text. Spell checkers search an input text for words in the dictionary and reject any strings that do not match. Our algorithm may find applications in spell checkers.

TABLE I

Algorithm	Running time of Preprocessing	Running Time of Matching
Naive Algorithm	0	O(nm)
Rabin–Karp string search algorithm	O(m)	average O(n + m), worst O((n−m)m)
Knuth–Morris–Pratt algorithm	O(m)	O(n)
Boyer–Moore algorithm	O(m + k)	best O(n/m), worst O(n)
Sunday Algorithm	O(m + k)	best O(n/m), worst O(nm)

II. Idea Behind Proposed Algorithm

This algorithm requires a bit of pre-processing step. For every symbol in the alphabet Pre-processing function computes the leftmost occurrence of symbol in the pattern string. If symbol does not occur in the pattern it stores location n where n-1 is the last valid location in the pattern. Proposed algorithm is based on the concept of comparison window. Size of comparison window is the size of the pattern. Every symbol in the comparison window has a location in the window denoted as j. Every symbol in the comparison window has its leftmost occurrence in the pattern denoted as k. In comparison window we carry out a right to left search. We look for the symbol whose $j < k$. Such a symbol cannot be a part of pattern occurrence. When we find such symbol in the window, we move the window beyond this symbol. If there is no such symbol in the window, we start a match in the window. If match succeeds, algorithm returns starting location of the pattern; otherwise we move window by one position in the right direction and repeat the process. If all windows exhaust and algorithm does not find a successful match, it returns -1.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

III. PROPOSED ALGORITHM

Here we present C++ instance of the proposed algorithm.

```
// Stores leftmost occurrence of character in pattern
inline void Preprocess(char* P, int n, char* Occ) {
        int j = 0;
        while(j < Size) {
                Occ[j] = n;
                j++;
        }
        j = n-1;
        while(j >=0) {
                Occ[P[j]] = j;
                j--;
        }
}

inline int Match(char* T, char* P, int n) {
        int j = 0;
        while(j<n) {
                if(T[j]!=P[j])
                return -1;
                j++;
        }
        return 1;
}
int Find(char* T, int m, char* P, int n) {
char Occ[Size];
Preprocess(P,n,Occ);
int start = 0;
char* x;
while(start <= (m-n)){  // start indicates start of comparison window
        int j =  n - 1;
        x = &T[start];
        int k;
        while(j >= 0) {
                k = Occ[x[j]]
                if(j < k)
                        break;
                j--;
        }

        if(j==-1){
                if(Match(x,P,n)==1){
                        delete[] Occ;
                        return start;
                }
                  j=0;
        }
        start = start+j+1;
}

delete[] Occ;
return -1;

  }
```

The following example illustrates the working of the algorithm.

Example:

Text String: w x x w z w u w x z **(Iteration 1)**

Pattern String: w u w x z

Position No. 0 1 2 3 4

w x x w z w u w x z **(Iteration 2)**

w u w x z

0 1 2 3 4

w x x w z w u w x z **(Iteration 3)**

w u w x z

0 1 2 3 4

TABLE III

Iteration no.	j	Symbol Involved	k	j < k
First iteration of Outer while loop	4	z	4	False
	3	w	0	False
	2	*x*	*3*	***True***
Second iteration of outer while loop	4	w	0	False
	3	u	1	False
	2	w	0	False
	1	*z*	*4*	***True***
Third iteration of outer while loop	4	z	4	False
	3	x	3	False
	2	w	0	False
	1	u	1	False
	0	w	0	False

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

The table II effectively captures the execution sequence of the algorithm. The pattern has size 5; therefore comparison window has size 5. In the first iteration of outer while loop we find symbol at location j=2 whose leftmost occurrence in the pattern is at location k = 3. Because j < k, symbol x which is at location 2 of text cannot be a part of pattern occurrence. Comparison window will now begin at third location of the text. Now in second iteration we find symbol z which is at location j = 1 of the window. The leftmost occurrence of symbol z is at position k= 4 in the pattern. As j < k, z cannot be a part of pattern occurrence. Now window moves to fifth location of the text. Now in third iteration we do not get any symbol for which window location is lesser than the leftmost location in the pattern. This fulfils the matching criterion; therefore we call function Match which returns success.

IV. Analysis

On average we will quickly find symbols which cannot be part of pattern occurrence, and comparison window will slide rapidly towards end of the array. When we cannot move the window there is high likelihood of finding the pattern. Therefore in such cases a match is likely to succeed. This suggests a highly efficient average case for algorithm.

When the text character which is compared with the rightmost pattern character does not appear in the pattern at all, then the comparison window can be shifted by n positions behind this text character. The best case for the algorithm is gained if at each attempt the rightmost symbol of the window does not appear in the pattern string. In best case the algorithm requires only O (n/m) comparisons.

Proposed algorithm is easy to understand, easy to maintain and easy to implement. Because theoretical running time of the algorithm is competitive, and its software complexity is low, it can be implemented in utility software.

V. Comparative Study

We compared proposed algorithm with leading existing algorithms using large data sets of English text. Result of the experiments suggests that proposed algorithm performs better on English text. Result of the comparative study has been summarized in table III.

S.No.	Algorithms	Average Execution time (in ms)
1	Naive String Matching	13.236
2	Rabin–Karp string search algorithm	11.349
3	Sunday Algorithm	11.215
4	Knuth–Morris–Pratt algorithm	15.834
5	Boyer–Moore algorithm	12.267
6	Proposed Algorithm	10.382

VI. Conclusion

Proposed algorithm elegantly solves string matching problem. Analysis of proposed algorithm suggests competitiveness in average case running time. Also, profiling experiments suggest that proposed algorithm performs effectively on English texts. The algorithm has significantly shown efficiency for the application of string matching. Proposed algorithm runs efficiently, and since it is easy to implement, it is a competitive choice to become a part of real world software like word processor.

REFERENCES

[1] Maxime Crochemore, Christophe Hancart and Thierry Lecroq, "Algorithms on Strings", Cambridge University Press, 2007.

[2] Melichar, Borivoj, Jan Holub and J. Polcar, "Text Searching Algorithms", Forward String Matching, Vol. 1, 2005.

[3] Sedgewick and Wayne, "Algorithms", 4th edition, Addison-Wesley Professional 2011.

[4] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford, "Introduction to Algorithms", 3rd edition, MIT Press and McGraw-Hill, 2009.