Review on String-Matching Algorithm

Zhaoyang Zhang *

University of Electronic Science and Technology of China

ABSTRACT: String-matching algorithm is one of the most researched algorithms in computer science which has become an important factor in many technologies. This field aims at utilizing the least time and resources to find desired sequence of character in complex data content. The most classical and famous string-search algorithms are Knuth-Morris-Pratt (KMP) algorithm and Boyer-Moore (DM) algorithm. These two algorithms provide efficient heuristic jump rules by prefix or suffix. Bitap algorithm was the first to introduce bit-parallelism into string-matching field. Backward Non-Deterministic DAWG Matching (BNDM) algorithm is a modern practical algorithm that is an outstanding combination of theoretical research and practical application. Those meaningful algorithms play a guiding role in future research in string-search algorithm to improve the average performance of the algorithm and reduce resource consumption.

1. INTRODUCTION

String-matching problem is one of the oldest and most widely studied problems in computer science. The problem is to find and match single or multiple patterns in the target string or text, having practical applications in a range of fields, including DNA bio-information matching, language translation, data compression, information retrieval, etc. More ordinarily, string matching is an inevitable step to identifying instruction in compilers and systems. Advanced matching methods with small time complexity become an important factor of computer science.

Although string-matching algorithm has been developed for half a century, very practical algorithms have only appeared for decades. There is a gap between theoretical research and application in the field. Experts specialized in algorithm research only concern algorithms that look wonderful in theory with good time complexity. While developers pursue the algorithm possibly fastest in practical. For a nonexpert in a searching algorithm, it is easy to get drowned in voluminous books of diverse matching algorithm, resulting in choice on an ostensibly simple but with the similar or even worse effect of a simple algorithm. A practical algorithm is supposed to have better performance in application and is easy to compete for implementation code in acceptable time. It is easy to find commonly used algorithms are classical algorithms and variants, which usually provide simple and effective approaches.

String-matching is usually divided into exact pattern matches and fuzzy matches. Fuzzy match algorithm is used to find whether an approximately equal substring occurs in the text string. 'Approximately equal' is defined by Levinstein distance [1]. In this paper we only review on exact pattern search. We review classical string-matching algorithms: KMP algorithm, BM string-search algorithm, Bitap algorithm, and BNDM algorithm, discussing characteristics of these four algorithms. This paper is aimed to get enlightenment and direction guidance in string-matching research.

2. REWIEW ON 4 ALGORITHMS

2.1. Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt (KMP) Algorithm was introduced by James Hiram Morris, Vaughan Pratt and Donald Knuth jointly in 1977's SIAM Journal on Computing [2]. KMP is the improvement of Naïve string searching algorithm. The core of KMP algorithm is to acquire information from matching failure to avoid pointer fallback. KMP algorithm is the first one to provide matching methods based on prefixes, indicating a way to optimize string search algorithm. In the follow-up, many developers deepened their research based on KMP and produced many optimizations and hybrid methods [2, 3].

Before search, KMP preprocesses generate a table recording the maximum length of a substring which is both a prefix and a suffix. This table determines to skip length beforethe next alignment, avoiding pointer fallback. Here we give the pseudocode as followed:

```
Algorithm kmp_string_search:
        Input:
    T[0:n-1]
    P[0:m-1]
        Output:
     R[]
```

*Corresponding author. Email: 2019190504036@std.uestc.edu.cn

```
        New array F[0:m-1]
/*preprocess*/
        F[0]←-1 /*assistive*/
        t←1
        p←0
        While (t< m-1) do
                if (P[t]==P[p]) then
                        F[t]←p
                        t←t+1
                else if (p>0) then
                        p←F[p]
/*the second longest prefix must be the longest prefix of the first prefix*/
                else
                        F[t]←0
                        t←t+1
/*search*/
        c←0
        while (i<n) do
                if (T[i]==P[j]) then
                        i←i+1
                        j←j+1
                        if (j==m) then
                                R[c]←i-m
                                c←c+1
                    j←F[m-1]
                    else
                        j←F[j-1]
                    if (j<0) then
                        i←i+1
                        j←0
        Return R
```

In this pseudocode, the auxiliary table is established in preprocessing. In this stage, there are three branches controlling the increment of t and p. In the first branch t and p increase 1 synchronously. In the second branch t is replaced with smaller F[t], thereby increasing $p - t$. In the third branch p increase 1 and t remains. Therefore, either p or the low boundary $p - t$ increases. The iteration must end up after $2m - 2$ loops. Thus, the time complexity is $O(m)$. Searching stage is simpler. Each step aligns pattern or moves text pointer one step. This fact implies that the loop executes at most $2n$ times. Thus, the algorithm execution search time complexity is $O(n)$.

2.2. Boyer-Moore Algorithm

Boyer-Moore (BM) string-matching algorithm was developed by Robert Stephen Boyer and J Strother Moore in 1977 [4]. This algorithm requires to preprocess on pattern. BM algorithm would skip some of characters instead of traversal. Generally, the longer the search keyword, the faster the algorithm speed. In general, it is 3-5 times faster than KMP algorithm. The high efficiency comes to the algorithm use this information from the fact that for every failed matching attempt, to exclude unmatched locations. This algorithm is often used in the search matching function in text editors. For example, the well-known GNU grep command uses BM, which is an important reason why GNU grep is faster than BSD grep.

BM algorithm is exceeding classical and successful, resulting it has been studied by later generations and produced many variants and optimization such as Apostolico–Giancarlo algorithm [5] and Horspool algorithm [6]. Meanwhile, BM algorithm performs splendidly in exact pattern matching. It has been the standard practical string-search literature benchmark [7].

Differing from Brute-Force and KMP algorithm, BM algorithm compares from end to beginning in the window. If the last compared text character mismatches and does not occur in a pattern, we directly move the window to skip the character. Otherwise, we skip maximum distance of two heuristic search rules [4]: The bad character rule and the good suffix rule.

2.2.1. The bad character rule

The bad character rule follows a simple principle. When a mismatch is found, assuming the character pair are $T[i] = c,\ P[j] \neq c \quad (j \leq m,\ j \leq i \leq n)$. The closest character 'c' in left is P[k], where $k = \max\{\alpha | P[\alpha] = c, \alpha < j\}$ if such 'c' doesn't occur. Then we start comparison from $T[m - 1 + j + k]$ and $P[m - 1]$

```
T:.....cxxx
K: .c...dxxx
K:    .c...dxxx
```

To implement the bad characters rule, we establish a 2D table to save the last occurrence of a character at left of a certain position.

```
    New array Bcr[c][j]
For (c∈Σ)
        j←0
        while (j<m-1) do
                Value ←-1       /*represents no match*/
                for (i←j-1;i>-1;i--) do
                        if (P[i]==c) then
                                value←i
                                Break   /*exit for loop*/
                Bcr[c][j]←j–value
                j←j+1
```

The realization is very plain and naïve by searching every character on left of P[i] and record the last occurrence of c. The time complexity of processing the table is $O(m\sigma)$, where σ is the size of alphabet.

2.2.2 The good suffix rule

The good suffix rule is more complex which has 3 cases. If the algorithm matches a good suffix and there is part of the substring in the pattern, we align them. In case that a substring S which is a good suffix of P and T has been matched when a mismatch occurs at P[i], we search a substring $S' = S$ in $P[0: i - 1]$.

Case 1 : If existed, we move pattern to align the rightmost S' with S.

```
T:.....csfz
K: .sfz...dsfz
K:      .sfz...sfz
```

Case 2: If not, we find the longest suffix of S which is a prefix of P and align them.

```
T:.....csfz
K:  fz...dsfz
K:      fz.....sfz
```

Case 3: Neither of above two cases, right shift the whole pattern m characters.

```
T:...cxxsfz
K:...dxxsfz
K:      ...dxxsfz
```

The core issue is which case the suffix should be applied. In order to implement the suffix rule efficiently, we define an auxiliary array suffix [], where $\text{suffix}[i] = s$, $s = \max\{s \mid P[i-s+1:i] == P[m-s+1:m]\}$. The construction of this array is simply realized as followed:

```
New array suffix[0:m-1]
Suffix[m-1]=m;
for(i←m-2;i>-1;i--) do
    q←i
    while ((q>-1)&&(P[q]==P[m–1–i+q])) do
        q←q–1
    suffix[i]=i-q
```

With the suffix[] array, we can establish Grs[] array in a very clever method to search find substring by good suffix.

```
New array Gsr[0:m-1]
    for (i←0;i<m;i++) do
        Gsr[i]←m
    for (i←m-1;i>-1;i--) do
        if (suffix[i]==i+1) then
            for (j←0;j<m-1–i;j++)
                if (Grs[j]==m)then
                    Grs[j]←m–1–i
    for (i←0;i<m;i++) do
        Grs[m–1–suffix[i]]←m–1-i
```

When a character meets several of the above three cases at the same time, we choose the smallest Gsr[i]. if there is a substring satisfies case 1 and a prefix satisfies case 2, we choose case 1 with which has a smaller step length.

Here we can give a complete pseudocode of BM algorithm:

```
Algorithm BM
    Input:
        T[0:n-1]
        P[0:m-1]
    Output:
        R[]
    i←m-1
    c←0
    /*Bcr[c][j] and Gsr[j] calculate in advance*/
    while (i < n) do
        for (j←m-1;P[j]==T[i–m+1+j];j--) do
        if (j==0) then
            R[c]←i-m+1
            c←c+1
            i←Bcr[P[0]][0]
        else
            i←max(Bcr[T[i–m+1+j]][j],Gsr[j])
        Return R[]
```

But in practical implement, most websites only create a one-dimensional bad character array to save the rightmost occurrence of character c. However, such a one-dimensional array does not affect the actual output result. A very loose explanation is that if we always align the rightmost c with mismatched character in text, the good suffix rule always ensures the algorithm works properly. The following is a rigorous prove.

Firstly, if the mismatched character $T[i] = c$ is found only on the right side of P[j] at P[k] ($k > j$). Good suffix rule will be applied in case 2 or case 3. Here we can deduce that $Gsr[j] \geq k+1 > j+1 \geq Bcr[c][j]$. Then, if the mismatched character $T[i] = c$ is on both side of P[j]. If the good suffix rule applies case 2 or case 3,we still have $Gsr[j] \geq j+1 \geq Bcr[c][j]$. . Otherwise, a rightmost substring satisfying $P[k-m+j-2:k] = P[j+1:m-1]$ ($k < j$), deducing that $Gsr[j] = (m-1)-k \geq j-k = Bcr[c][j]$. At this point, we can always ensure that the good suffix rule can move a greater distance.

So, pseudocode of the bad character rule is optimized as followed:

```
New array B[c]
    for (c∈Σ) do
        Bcr[c]←-1
    for (j←0;j<m;j++) do
        B[p[j]]←j
```

Then we adjust BM algorithm minorly by replacing i←max(Bcr[T[i–m+1+j]][j],Gsr[j]) with i←max(j-B[T[i-m+1+j]],Gsr[j].

2.3. Bitap Algorithm

Bitap algorithm is an approximate string-matching algorithm, also known as shift-or, shift-and and Baeza-Yates-Gonnet algorithm. The exact string bitap search algorithm was first introduced by BálintDömölki in 1964 and extended by R.K. Shyamasundar [8] in 1977. Later in 1989 Ricardo Baeza-Yates extended it to process with wildcard and mismatch [9]. In Baeza-Yates' paper [9], bit-parallelism method was implemented in string searching for the first time which convinced the classical Shift-Or algorithm. Then in 1991, Udi Manber and Sun Wu's paper introduced Shift-And algorithm that was improved on basis of Shift-Or, giving a new extension to fuzzy match of regular expression [10]. It was improved again in 1999 by Baeza-Yates and Gonzalo Navarro [11].

Bitap algorithm performs string matching based on prefix and is much simpler than KMP. It maintains a dynamic array in form of bitmask by bit-parallelism to saves information that are the prefix of the pattern string and a substring of the text are identical. Assuming such an array is $D[0:m-1], \forall D[i] \in \{0,1\}$. We assert $D[i] = 1$ iff $P[0:i] = T[j-i:i]$. it is worth noticing that array Dis dynamic and every D[i] will be updated based on new read character. When $D[m-1] = 1$, we affirm that the pattern is found in text.

The algorithm uses an auxiliary preprocessed two-dimension table B. Table B records occurrence of each character alphabet in m bit bitmasks.

Here we give the update rule first:

$$D \leftarrow (D << 1 \mid 0^{m-1}1) \,\&\, B[S[i+1]]$$

This update rule is the soul of Bitap algorithm where it cleverly extends and find all matched prefix P and suffixs of S using bit parallel computing.

Here we give an example of updating D. Assuming T is cbcbcba and P is cbcba.

Table 1 Auxiliary Table B of Bitap Example

B	B bitmask
a	10000
b	01010
c	00101
*	00000

where * represent characters that don't occur in pattern. Now we can calculate D bitmask step by step.

Table 2 Implementation of Bitap

Steps	Read	D bitmask	B bitmask	Matched prefix of P and suffix of T
0	Initial	00000		
1	c	00001	00101	c
2	b	00010	01010	cb
3	c	00101	00101	cbc c
4	b	01010	01010	cbcb cb
5	c	00101	00101	bc c
6	b	01010	01010	cbcb cb
7	a	10000	10000	cbcba

Based on the algorithm logic described above, pseudocode can be given as follow:

```
Algorithm bitap_search:
    input:
        T[0:n-1]
        P[0:m-1]
    output:
        R[]
    /*generate bitmask table B*/
    for (c∈Σ) do
        B[c]←0
        for (i←0;i<m;i++)do

    B[P[m−1−i]]←B[P[m−1−i]]+2^i
    /* regenerate non-zeros bitmask for characters in
P*/
    D←1       /*D = 0^(m-1)1 */
    i←0
    while (i<n) do
    D←((D<<1)|1)&B[T[i]]
        if ((D&2^(m - 1))≠0)then
        R.append({i-m+1})  /*add i-m+1 to set
R*/
        i←i+1
    return R
```

In preprocess stage, the algorithm initializes all bitmask to 0^m. Then for $c \in P$, regenerating B[c] by adding 2^i converting i bit to 1 in binary. This part runs in $O(m+\sigma)$ where σ is the size of alphabet. In search stage, it is a no-jump-out $n \times m$ iteration. Obviously, its operation time complexity is always $O(nm)$, despite the structure of text and pattern.

Bit-parallelism algorithm in Bitap is used to simulate KMP algorithm and accelerate the implementation of the classical algorithm. In practical implementation, we need additional bitmask for every character. Therefore, Bit parallel algorithm is more suitable for the case of pattern string with fewer characters or input over a smaller alphabet.

2.4. Backward Non-Deterministic DAWG Matching Algorithm

The relationship between BNDM algorithm and BDM algorithm is very similar to Bitap algorithm and KMP algorithm, where Bitap depends on prefix search, but BNDM depends on suffix and substring search. BDM is a search method based on substring, and its difficulty lies in how to search substring. "Flexible Pattern Matching in Strings" [12] introduced Suffix Automata (SAM) to realize distinguish substring. Here we do not dig into SAM but utilize SAM as a DFA tool to instruct how to do state transition. Compared with the original BDM, BNDM is simpler, uses less memory, has better reference locality, and is easy to be extended to more complex pattern strings.

BNDM algorithm also maintains dynamic array to record all the positions of substrings that have been found in pattern. In formal description, D is a m bit array, where $\forall D[i] \subset \{0,1\}, i \subset [0, m-1]$, and initially set to be 1. $D[i] = 1$ iff $P[i:i+k] = T[j-k:j]$ $(i+k \leq$

m) .when the algorithm reads T[j − k] from right to left in window. last is the step length after failure to match. last is initially asserted m. j is the number of unmatched characters in window, initially asserted m . last is asserted j if $D[0] = 1$, which means a suffix of T is the prefix of P. $last = 0$ is exactly the judgment of text matched pattern. To implement the algorithm, we need set up an auxiliary table B that is similar to Bitap algorithm.

Here we give the pseudocode of BNDM algorithm:

```
Algorithm BNDM
    Input:
        T[0:n-1]
        P[0:m-1]
    Output:
        R[]
    for (c∈Σ) do
        B[c]←0
    for (i←0;i<m;i++)do
        B[P[m−1−i]]←B[P[m−1−i]]+2^i
    j ← 0
    while (j+m≤n) do
        i←m
        shift←m
        D←2^m−1  /* D ← 1m*/
            while D ≠ 0 do
        /* T[j+i:j+m] is a substring of pattern*/
            i←i–1
            D←D&B[T[j + i]]
            if (D&2^(m-1)≠0 )then
            /* T[j+i:j+m] is a pattern prefix*/
                if (i==0)then

    R.append({j})
                                /*add j to set R*/
                else shift←i
                D←D<<1
        j←j+shift
    Return R
```

We can easily estimate the preprocess time complexity is $O(\sigma + m)$ like Bitap. In search stage, the worst case is only a minor increment shift happens, for example $T = a^n, P = a^{m-1}b$. the inner loop will be run $m - 1$ time before exit to deduce shift to 1. The outer loop will be operated from m to n with increment of 1. This means the overall the worst-case time complexity is $O(mn)$. At the best case for example $T = ab^{n-1}, P = a^m$ the search window skips at length m. The inner loop will only run once with $O(1)$ time complexity and the outer looper will be executed n/m times. Therefore, the best-case time complexity is $O(n/m)$.

3. DISSCUSION

These four algorithms roughly cover the mainstream string-searching strategies, including searching by prefix, by suffix and by best factors [12]. KMP and BM are the two most classical string-search algorithms. They provide heuristic jump rules and enlightenment on effective ways for the later string search algorithms. Any string-matching algorithm must check at least $n - m + 1$ characters in text in worst case. This strictly shows that no algorithm has better computational complexity than KMP or BM algorithm. It indicates that the subsequent improvement and optimization of KMP and BM are to improve the average performance and reduce the spatial complexity as possible. Bitap algorithm is different from the two well-known string search algorithms in its natural mapping to simple bitwise operation. The most remarkable point of Bitap algorithm is that it introduces bit-parallelism into string search. Although its time complexity is not excellent in theory, it provides an approach to accelerate computation by bitmask. Each bit merely represents occurrence and almost all operations are bit operations. BNDM algorithm resembles the hybrid of Bitap algorithm and BDM algorithm. It takes full advantage of bit parallelism calculation from Bitap and efficient heuristic jump rules from BDM algorithm, having high operation speed in most cases. In other words, BNDM is an algorithm combining theoretical research and practical application and is easy to implement.

String matching has been one of the hotspots in the field of IDS research. With the rapid development of network, there is a performance bottleneck in the network application based on character matching technology. To improve the overall performance of the system is the main work of researchers and designers. Character matching is one of the main technologies to implement network intrusion detection. Therefore, efficient algorithms will be one of the means to improve the overall performance of the system. The main direction of the next research is to find ways to improve the average performance of the algorithm and reduce resource consumption: the main way can reduce the number of actual matches by reducing the m or n values in the actual network applications; To reduce the time cost of matching, we try to migrate some matching algorithms to hardware implementation or parallel architecture implementation.

4. CONCLUSION

We reviewed the string-search algorithm and briefly introduced Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm, Bitap algorithm, and Backward Non-Deterministic DAWG Matching algorithm. We sketched the background development and operation principle of each algorithm, explaining the detailed running process and efficiency of each algorithm. KMP and BM algorithm are the most classical and well-known algorithm which are pioneers-initiated research on efficient string-search algorithm. Bitap algorithm was the first to introduce bit-parallelism into string-match field, providing simple and efficient calculation to improve performance and reduce RAM occupation. BNDM algorithm is a combination of theoretical research and practical application, optimizing BDM algorithm to improve efficiency and implement convenience. The development of these four algorithms demonstrates the direction of the next research which is to

improve the average performance of the algorithm and reduce resource consumption. This passage is limited by reviewing only four exact pattern-matching algorithms, excluding other famous and efficient algorithms like Rabin-Karp algorithm, in lack of guidelines in fuzzy match algorithm.

REFERENCES

1. Levenshtein, V. I."Binary Codes Capable of Correcting Deletions, Insertions and Reversals", Soviet Physics Doklady, vol. 10, p. 707, 1966.
2. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, “Fast pattern matching in strings,” SIAM Journal on Computing, vol. 6, no. 2, pp. 323–350, 1977.
3. Hou Xian-feng, Yan Yu-bao and Xia Lu, "Hybrid pattern-matching algorithm based on BM-KMP algorithm," 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), 2010, pp. V5-310-V5-313, doi: 10.1109/ICACTE.2010.5579620.
4. Michael Maher, "How tp Extend Partial Deduction to Derive the KMP String-Matching Algorithm from a Naive Specification," in Logic Programming: Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, MIT Press, 1996, pp.539-539.
5. R. Boyer and J. Moore, "A fast string searching algorithm", Communications of the ACM, vol. 20, no. 10, pp. 762-772, 1977. Available: 10.1145/359842.359859.
6. A. Apostolico and R. Giancarlo, "The Boyer–Moore–Galil String Searching Strategies Revisited", SIAM Journal on Computing, vol. 15, no. 1, pp. 98-105, 1986. Available: 10.1137/0215007.
7. R. Horspool, "Practical fast searching in strings", Software: Practice and Experience, vol. 10, no. 6, pp. 501-506, 1980. Available: 10.1002/spe.4380100608.
8. A. Hume and D. Sunday, "Fast string searching", Software: Practice and Experience, vol. 21, no. 11, pp. 1221-1248, 1991. Available: 10.1002/spe.4380211105.
9. R. Shyamasundar, "Precedence parsing using domolki's algorithm", International Journal of Computer Mathematics, vol. 6, no. 2, pp. 105-114, 1977. Available: 10.1080/00207167708803130.
10. R. Baeza-Yates. "Efficient Text Searching." PhD Thesis, University of Waterloo, Canada, May 1989.
11. S. Wu and U. Manber, "Fast text searching", Communications of the ACM, vol. 35, no. 10, pp. 83-91, 1992. Available: 10.1145/135239.135244.
12. R. Baeza-Yates and G. Navarro, "Faster Approximate String Matching", Algorithmica, vol. 23, no. 2, pp. 127-158, 1999. Available: 10.1007/pl00009253.
13. G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge: Cambridge University Press, 2002.