Algorithms for String matching

Marc GOU

July 30, 2014

Abstract

A string matching algorithm aims to find one or several occurrences of a string within another. The algorithm returns the position of the first character of the desired substring in the text. There are many different solutions for this problem, this article presents the four best-known string matching algorithms: Naive, Knuth-Morris-Pratt, Boyer-Moore and Rabin-Karp. The results show that Boyce-Moore is the most effective algorithm to solve the string matching problem in usual cases, and Rabin-Karp is a good alternative for some specific cases, for example when the pattern and the alphabet are very small.

1 Introduction

The purpose of this paper is to study different algorithms for the *String matching problem*. These algorithms are used for trying to find one, several or all occurrences of a defined string (*pattern*) in a larger string string (typically a text). 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 a text-editing software. Other applications in information technology includes web search engines, spam filters, natural language processing, computational biology (search of particular pattern in DNA sequence), feature detection in digital images...

There are different solutions that allow to solve the string matching problem. First, we have the *naive algorithm*, the simplest one, which tries to match the pattern to each string of the same length in the text. From the 1970s, several others algorithms, more sophisticated and more effective, have been invented. In 1975, Knuth, Pratt and Morris invented the first algorithm that preprocesses the pattern to obtain a better performance, it is the *Knuth-Morris-Pratt Algorithm*. In 1977, another algorithm that preprocesses the pattern was invented: *Boyer-Moore Algorithm* [2], its main characteristic is the fact that it attempts to establish the correspondence of the substring with the pattern in the reverse direction. In 1987, Rabin and Karp propose an algorithm that is based on a completely different approach: *Rabin-Karp Algorithm* [5], which computes a hash function for the pattern and then look for a match by using the same hash function for each possible substring of the same length in the text.

In this paper, we present the four algorithms mentioned above. The final goal is to be able to answer the question which of these algorithms is the best. To achieve this, we implement, test, and compare the efficiency of each algorithms. The comparison will be executed in different situations: small and large alphabet, 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 algorithms by measuring their execution times in these different conditions.

In Section 2 the problem is introduced and explained in detail. We will also present the different notations used in this paper. In Section 3 we devote a section to each solution to the problem, which explains in detail the operation of the algorithm. We start with the simplest solution, the naive algorithm. Then we show three more sophisticated and more efficient solution.

After that, we show and compare the obtained results of each algorithm in different considered cases in the Section 4, we observe that the Boyer-Moore Algorithm is the best solution, but not in all cases. Finally, in Section 5, we summarize the important elements of this paper with the conclusions.

2 Background

We introduce the following notations:

- the *alphabet*, Σ, is a finite set of symbols,
- the string from where to search, the *text*, is T, and its length $\mid T \mid = n$, and
- the string that is searched, the *pattern*, is P, and its length $\mid P \mid = m$.

It is clear that the text must have at least the same length as the pattern, $n \geq m$, otherwise the problem would not make sense. In brief, the string matching problem consists of finding all *valid shifts* with which a given pattern P occurs in a given text T [4]. A position $s \in \{0, \ldots, n-m\}$ in T is a valid shift if P occurs with *shift* s in T, or mathematically, if we have

$$pattern[i] = text[s+i], \forall i \in \{0, \ldots, m-1\}$$

In all other cases, s is a *invalid shift.* If s is a valid shift, then we have $P[0...m-1] = T[s...s+m-1]$.

Example: If sale is a preconfigured keyword in a spam filter, when it scans this following subject line, the position $s = 3$ is a correct shift. See Figure 1.

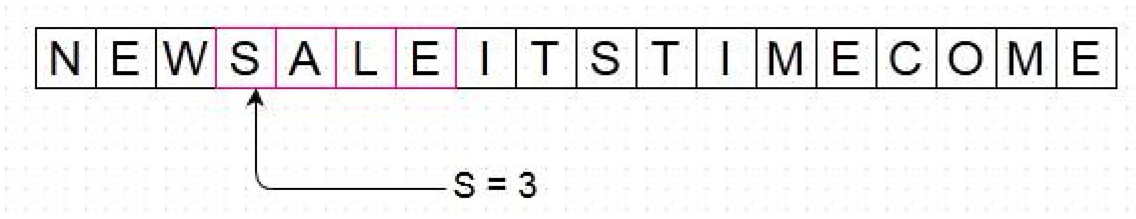

Figure 1: Example of a string matching problem application

In following section, we would meet the notation a^n, it simply represents $\underbrace{aa...a}_{n}$.

3 Algorithms

3.1 Naive Algorithm

The idea of the naive solution is just to make a comparison character by character of the text $T[s...s+m-1]$ for all $s \in \{0, \ldots, n-m+1\}$ and the pattern $P[0...m-1]$. It returns all the valid shifts found. Figure 2 shows how the algorithm work in a practical example. In Figure 3 is an implementation written in pseudo-code of the naive algorithm.

The problem of this approach is the effectiveness. In fact, the time complexity of the Naive algorithm in its worst case is $O(M \times N)$. For example if the pattern to search is a^m and the text is a^n, then we need M operation of comparison by shift. For all the text, we need $(N-M+1) \times M$ operation, generally M is very small compared to N, it is why we can simply considered the complexity as $O(M \times N)$.

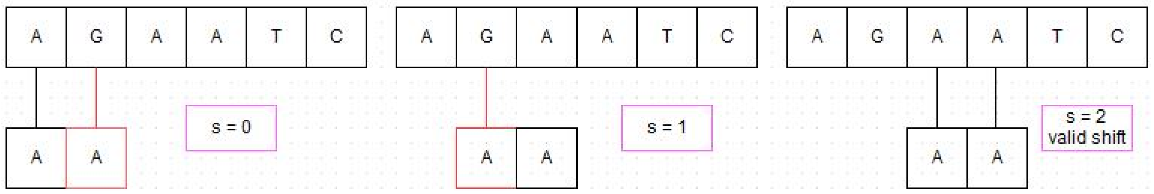

Figure 2: Example of operation of the naive string matcher in a DNA string

```
// Return an array which contains all valid shifts in text (str)
NaiveMethod(text, pattern)
{
    N = length(text);
    M = length(patter,);
    limit = N-M;
    j = 0, k = 0;
    arrayOfValidShift[];

    for(i = 0; i <= limit; i++)
    {
        j = 0;
        k = i;
        for(j = 0; j <= M AND str[k] == pat[j]; j++)
            k++;

        if(j >= M)
            Add i to arrayOfValidShift;
    }

    return arrayOfValidShift;
}
```

Figure 3: Implementation of the naive solution

3.2 Knuth-Morris-Pratt Algorithm (KMP)

The KMP algorithm is a linear time algorithm, more accurately $O(N+M)$. The main characteristic of KMP is each time when a match between the pattern and a shift in the text fails, the algorithm will use the information given by a specific table, obtained by a preprocessing of the pattern, to avoid re-examine the characters that have been previously checked, thus limiting the number of comparison required. So KMP algorithm is composed by two parts, a searching part which consists to find the valid shifts in the text, where the time complexity is $O(N)$, obtained by comparison of the pattern and the shifts of the text, and a preprocessing part which consists to preprocesse the pattern.

The goal of the preprocessing of pattern consist to obtain a table that gives the next position in the pattern to be processed after a mismatch. For a pattern $P[0...m-1]$, the table of result of the preprocessing will give for each character j contained in the pattern a value which is defined as the substring that is in the same time the longest prefix of the pattern and the suffix of the substring of pattern $P'[0...j]$ [5]. The complexity of the preprocessing part is $O(M)$, applying

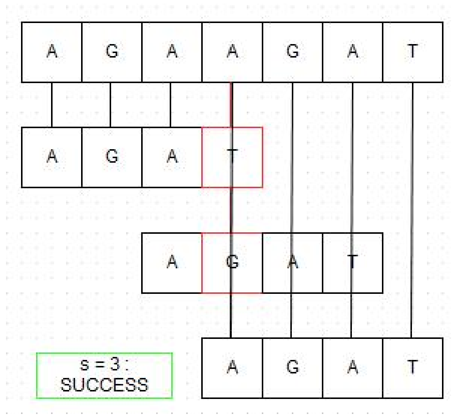

Figure 4: Example of operation of the KMP string matcher in a DNA string

the same searching algorithm to the pattern itself.

In Figure 4 there is an example where we need three attempts to find a valid shift, whereas with the naive solution, we need four attempts, we could not skip the shift at the position one.

3.3 Boyer-Moore Algorithm (BM)

The basic idea behind this solution is that the match is performed from right to left. This characteristic allows the algorithm to skip more characters than the other algorithms, for example if the first character matched of the text is not contained in the pattern $P[0...m-1]$, we can skip m characters immediately. As the KMP algorithm, this algorithm preprocesses the pattern to obtain a table which contains information to skip characters for each character of the pattern. But BM algorithm use also another table based on the alphabet. It contains as many entries as there are characters in the alphabet [2].

In the example below, we can easily persuade the advantage of BM algorithm over KMP and the naive one, we only need four attempts to find the valid shift. In this case, the time complexity of the BM algorithm is sublinear: $O(N/M)$.

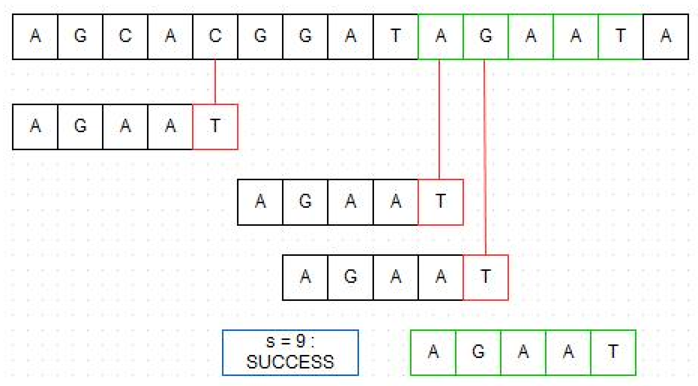

Figure 5: Example of operation of the BM string matcher in a DNA string

In the worst case, the complexity of the algorithm is $O(N \times M)$, it happens for example when the size of the alphabet is one, or more generally when the pattern and the text are strings composed by sequences of one same character.

3.4 Rabin-Karp Algorithm (RK)

The Rabin-Karp algorithm uses a totally different approach to solve the string matching problem. This method is based on hashing techniques. We compute a hash function $h(x)$ for the pattern $P[0...m-1]$ and then look for a match by using the same hash function for each substring of length $m-1$ of the text [5].

The Rabin-Karp also use preprocessing technique before the search operation. Its preprocessing operation is the hashing of the pattern, which is $O(M)$ complexity. So, the running time of the algorithm is $O(M \times (N - M + 1))$, but in general, we will see, that the algorithms will run with a complexity $O(N)$.

Let's introduce following notations:

- $h(p)$: the hashed value of the pattern
- $h(t_s)$: the hashed value of the substring $[s, ..., s + M - 1]$

In Figure 6 there is an implementation of this algorithm written in pseudo-code. We can see that the computation of $h(t_{s+1})$ (`REHASH(text[j], text[j+M], hText, d)` in the code) can be based on the value of $h(t_s)$ (`hText` in the code) by using `REHASH`, which is an $O(1)$ complexity operation [3].

Example: if we have P ="cd" and T = "abcd". Based on the implementation, we can easily obtained $h(p) = 99 \cdot 2 + 100 = 298$, where 99 and 100 are respectively the integer value of c and d in ASCII representation.

We compute $h(t_0) = 292$ in the same way, we can see that $h(p) \neq h(t_0)$, so we will use the `REHASH` function to compute $h(t_1) = 295$. This value does not match with $h(p)$ too, so we compute $h(t_2) = 298$, it matches with $h(p)$, but we still need to check character by character to avoid collisions.

4 Comparisons and analysis of the effectiveness

4.1 Working environment

The hardware used for different tests of these algorithms is a laptop with a Intel i5 dual-core processor and with 6GB of RAM memory. This laptop runs under Linux Mint 14. All the algorithms are implemented in C++, the compilator used is g++, the different strings using in different tests are randomly generated by a script written in Php.

In this paper, we will consider two different alphabets, a small and a large:

1. $|\Sigma| = 4$: $\Sigma = \{A, C, G, T\}$
2. $|\Sigma| = 62$: $\Sigma = \{0, \dots, 9, a, \dots, z, A, \dots, Z\}$

The first alphabet corresponds to the set of DNA nucleobases, the research of a defined DNA sequence is one of the direct application of string matching [1]. The second alphabet have some applications in security area (spam filter...) for example [5].

The size of the text is fixed to ten millions, five different random texts are used for each test. Seeing that the texts are randomly generated, the number of matches depends on the size of the alphabet and the size of the pattern. The test finds all matches in the text and return a vector which contains all the valid shifts.

The results of these tests are expressed in *seconds*.

```
#define REHASH(a, b, h, d) ((((h) - (a)*d) << 1) + (b))

// Return an array which contains all valid shifts in text (str)
RabinKarpMethod(text, pattern)
{
    N = length(text);
    M = length(pattern);

    d = 1, hText = 0, hPattern = 0, j = 0;

    // PREPROCESSING
    for(i = 1; i < M; ++i)
        d = (d<<1);

    for(i = 0; i < M; ++i)
    {
        hPattern = ((hPattern<<1) + pattern[i]);
        hText = ((hText<<1) + text[i]);
    }

    arrayOfValidShits[];

    // SEARCHING
    while(j <= N-M)
    {
        if(hPattern == hText) // Potential match
        {
            for(i = 0; i < M AND text[j+i] == pattern[i]; i++); // Check

            if(i >= M) // Match / Valid shift -> add in the array
                Add j to arrayOfValidShift;
        }

// Computation of the new value of hText based on its old value
        hText = REHASH(text[j], text[j+M], hText, d);

        ++j;
    }

    return arrayOfValidShift;
}
```

Figure 6: Implementation of Rabin-Karp solution

4.2 Time measurements

Table 1 shows the results obtained with a big alphabet (alphanumeric) and Table 2 shows the results obtained with a small alphabet (DNA).

Table 1: Table of results in milliseconds - Alphanumeric Alphabet

$\|pattern\|$	matches	Naive	KMP	BM	RK
3	40	225	221	242	194
10	0	224	221	82	194
50	0	224	221	25	194

Table 2: Table of results in milliseconds - DNA Alphabet

$\|pattern\|$	matches	Naive	KMP	BM	RK
3	156455	331	308	340	204
10	8	329	306	204	187
50	0	328	305	148	186

4.3 Analysis

By looking at these results, we can conclude that the best algorithm in majority of cases is Boyer-Moore. More the pattern is long, more its advantage become significant, the reason to this can be easily explained by the fact that it could skip more characters in this case, its complexity is sublinear: $O(N/M)$. But we also notice that when the pattern is very small, the advantage of BM algorithm disappears completely.

Knuth-Morris-Pratt and the naive algorithm obtain similar results, but the result of KMP is always a bit better. KMP could be a good choice if the length of pattern is very short.

Rabin-Karp obtains very good results in these tests, its results are a lot better than KMP and the naive solution and it is definitely the best choice in the situations where Boyce-Moore is not adapted.

In general, we also notice that the results with a bigger alphabet is always better than a small one. The reason is, because with a small alphabet we have more similarity between the pattern and substrings of the text even there are not matches, but these similarities require more comparisons.

5 Conclusion

We have presented the most famous string matching algorithm. To answer the question: Which algorithm is the best? We implemented the four algorithms in a practical programming language (C++), and after made different tests and comparisons with these implementations.

The answer is that Boyer-Moore is the best algorithm. Not in all cases, as we can observe in Section 4, but in the practical cases, generally the length of the pattern and the size of alphabet are not so small that the advantageous characteristic of BM disappears, that is why we can consider BM algorithm as the best one. Rabin-Karp is also an effective algorithm, it is even better than BM when the pattern and the alphabet are very small. The naive algorithm is the worst solution, with the slowest execution time.

References

[1] Ricardo A. Baeza-Yates. "algorithms for string searching: A survey". *ACM SIGIR Forum, 23*, pages 34–58, 1989.

[2] Robert S. Boyer and J. Strother Moore. "a fast string searching algorithm". *Communications of the ACM, Volume 20, Number 10*, pages 762–772, october 1977.

[3] Christian Charras and Thierry Lecroq. "exact string matching animation in java". *Report LIR 97.10, Université de Rouen*, 1997. `http://www-igm.univ-mlv.fr/~lecroq/string/index.html`.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms, Third edition.* The MIT Press, 2009.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth edition.* Addison-Wesley, 2011.