A String Matching Algorithm Based on Efficient Hash Function

Zhao Fuyao
School of Information Science and Engineering
Central South University
Hunan, China
Email: zhao.fuyao@yahoo.com

Liu Qingwei
School of Information Science and Engineering
Central South University
Hunan, China
Email: liuqingwei2019@gmail.com

***Abstract*—Based on the Karp-Rabin algorithm, a fast string matching algorithm is presented in this paper. It is also a hash-based approach, comparing the hash value of strings called fingerprint rather than the letters directly. The characteristic of the algorithm is that the hash function exploits bitwise operations and also considers about the size of the alphabet and the length of the pattern. We prove that the probability of a hash collision is minute and that the average running time is linear. Through a series of experiments, the remarkable performance of our algorithm is demonstrated.**

***Keywords*—string matching; hash; fingerprint; bitwise operations**

I. Introduction

The exact string matching problem can be formalized as follows. Given a pattern P of length m and a text T of length n $(n \geq m)$, in which all characters are taken from a fixed alphabet Σ, the problem is to find all occurrences of P in T.

Many algorithms, among which the most famous two are the Knuth-Morris-Pratt [1] and the Boyer-Moore algorithm [2], have been proposed to solve the problem. A comparison about the performance of different string matching algorithms is shown in Fig. 1 given in [3]. The map illustrates the following points: Shift-Or algorithm [4] (prefix based) runs faster for short patterns (up to length 8) on small alphabets (about length 4). Horspool algorithm [5] (suffix based) is more competitive in larger alphabets. The better choice for longer patterns on small alphabets is the BNDM (Backward Nondeterministic Dawg Matching) algorithm [6] or BOM (Backward Oracle Matching) algorithm [7] (factor based).

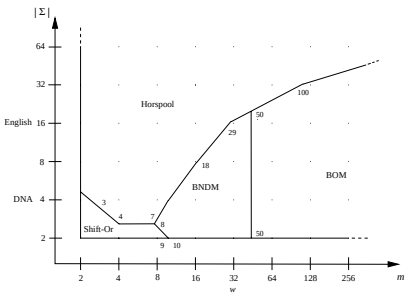

Fig. 1. Map of experimental efficiency for different string matching algorithms, given in [3].

Another approach is the Karp-Rabin algorithm [8], [9], which uses the idea of fingerprinting that hashes the long strings to much shorter symbols called fingerprints. This technique is widely used in string related problems and performs well in practice. However, the original Karp-Rabin algorithm is not efficient enough to compete with the above algorithms.

Our idea is to accelerate the computation of fingerprints by bitwise operations on the basis of Karp-Rabin algorithm. Meanwhile, the pattern length and the alphabet size are also made use of to modulate the hash parameters. We prove that the probability of a hash collision in our algorithm is very low, and that the complexity of running time on average is linear. Besides, the experimental results show that the new algorithm could speed up the string matching process considerably. Not only can it run faster than other prefix-based approaches, but it also has dominance over the situation where alphabet and pattern are both small. Nevertheless, as the size of the alphabet and the length of the pattern grow, other sublinear approaches such as Horspool algorithm [5] and BOM algorithm [7] would perform better.

The following definitions are used in this paper.

Σ is a finite ordered alphabet, and $|\Sigma|$ is its size. We denote w the machine word length. The fingerprint of pattern P is denoted by p, and that of the text block $T[s+1..s+m]$ is denoted by t_s. The notation $(b_\ell ... b_1)_2$ means a bit sequence of length l in a computer word, and uses exponentiation on bit to show its repetition (e.g. $(1110000)_2$ is usually denoted as $1^3 0^4$). Bitwise operations are denoted as follows:"|" (the bitwise or), "&" (the bitwise and),"<<" (the bitwise shift-left) and ">>" (the bitwise shift-right). The $probability$ of an event A is denoted $Pr(A)$.

The rest of the paper is organized as follows. Section II reviews the concept of fingerprinting and the classical Karp-Rabin algorithm. Next, we devise a fast string matching algorithm in section III. In section IV, a theoretical analysis of the algorithm is given. And in section V, we provide our experimental results. Finally, section VI gives our conclusions.

II. Fingerprinting and Matching

In Karp-Rabin algorithm [8], [9], strings are hashed to fingerprints using the rolling hash function. During the process, the text blocks in a window moving through the text will be

handled consecutively. Based on the old fingerprint, the new one can be rapidly got in constant time only by removing the old value from the window and adding in new value.

As a result, Karp-Rabin algorithm simply needs to compare the fingerprints of strings rather than the letters directly. Only when the two fingerprints are equal, does the algorithm implement a checking to validate. If the original strings do not agree, then it produces a $false\ match$, otherwise, reports an occurrence. The pseudo-code is shown below. Typically, q is taken to be a large prime and d is $|\Sigma|$.

Karp-Rabin(T, n, P, m, d, q)

```
1: ▷ Preprocessing
2: p ← 0
3: t_0 ← 0
4: h ← d^{m-1} mod q
5: for i ← 1 to m do
6:     p ← (d · p + P[i]) mod q
7:     t_0 ← (d · t_0 + T[i]) mod q
8: end for
9: ▷ Searching
10: for s ← 0 to n − m do
11:     if p = t_s then
12:         if P[1..m] = T[s + 1..s + m] then
13:             report an occurrence
14:         end if
15:     end if
16:     if s < n − m then
17:         t_{s+1} ← (d(t_s − T[s + 1] · h) + T[s + m + 1]) mod q
18:     end if
19: end for
```

In Karp-Rabin algorithm, the probability of a false match under a Bernoulli model with equiprobability of letters is $\leq \frac{1}{q}$ [10]. And the expected running time is $O(m+n)$ [8].

However, in spite of the minuscule probability of a false match, Karp-Rabin algorithm is still inferior to other string matching algorithms in practice. Closer scrutiny shows that it is the complex arithmetic operations rather than checking for false matches that circumscribes the algorithm's performance.

III. New Algorithm to String Matching

Aimed at improving the Karp-Rabin algorithm, we propose a faster hash function which avoids the complex arithmetic operations. More precisely, we take $\beta\ least\ significant\ bits$ (LSB) from each character, which can be done efficiently via bitwise operations. Then, by using rolling hash technique, a binary sequence can be got and used as the fingerprint of the string. Assuming the alphabet contains 128 ASCII characters and $\beta = 5$, several LSBs are shown in Table I.

TABLE I
THE LSBS OF SOME CHARACTERS. IT IS POSSIBLE THAT TWO DIFFERENT CHARACTERS HAVE THE SAME LSB.

Character	ASCII Value	Binary	LSB
L	76	1001100	01100
a	97	1100001	00001
c	99	1100011	00011
e	101	1100101	00101
l	108	1101100	01100

Based on the hash technique above, we give two algorithms which are slightly different in the strategy of computing β and the hash process.

Algo1(T, n, P, m, $|\Sigma|$)

```
1: ▷ Preprocessing
2: return an error if m > w
3: β ← ⌊w/m⌋
4: p ← 0
5: t_0 ← 0
6: for i ← 1 to m do
7:     p ← (p << β) | (P[i] & 0^{w−β}1^β)
8:     t_0 ← (t_0 << β) | (T[i] & 0^{w−β}1^β)
9: end for
10: ▷ Searching
11: for s ← 0 to n − m do
12:     if p = t_s then
13:         if |Σ| ≤ 2^β OR P[1..m] = T[s + 1..s + m] then
14:             report an occurrence
15:         end if
16:     end if
17:     if s < n − m then
18:         t_{s+1} ← ((t_s << β) & 0^{w−βm}1^{βm}) | (T[s + m + 1] & 0^{w−β}1^β)
19:     end if
20: end for
```

Algo2(T, n, P, m)

```
1: ▷ Preprocessing
2: m' ← min(w, 2^{⌊log_2 m⌋})
3: β ← w/m'
4: p ← 0
5: t_0 ← 0
6: for i ← 1 to m' do
7:     p ← (p << β) | (P[i] & 0^{w−β}1^β)
8:     t_0 ← (t_0 << β) | (T[i] & 0^{w−β}1^β)
9: end for
10: ▷ Searching
11: for s ← 0 to n − m do
12:     if p = t_s then
13:         if P[1..m] = T[s + 1..s + m] then
14:             report an occurrence
15:         end if
16:     end if
17:     if s < n − m then
18:         t_{s+1} ← ((t_s << β) | (T[s + m' + 1] & 0^{w−β}1^β)
19:     end if
20: end for
```

The two algorithms share the same structure. They first calculate β according to the size of Σ and the length of the pattern P (lines $1 \sim 3$), then compute the fingerprints of the pattern and the first block of text (lines $4 \sim 9$). At last they maintain a loop that iterates s from 0 to $n-m$ (lines $10 \sim 20$). In each stage, the algorithms compute t_{s+1} through t_s and compare the fingerprint of the pattern with that of the text block. If $p = t_s$, then they check if $P[1..m] = T[s+1..i-1]$. Consequently, the two algorithms report valid occurrences of P in T and only them.

Yet, Algo1 emphasizes that each character in the pattern should contribute to the fingerprint with equal information.

Moreover, if $|\Sigma| \leq 2^\beta$, it is unnecessary to check $P[1..m] = T[s+1..s+m]$ when $p = t_s$, since the information of the strings is all packed in one computer word. However, it can not work in the circumstances where $m > w$.

Unlike Algo1, Algo2 fits the fingerprint of the pattern just within one computer word. To achieve this, only its prefix of length m' is hashed, restricted to $m' \leq m$ and $m' = 2^r$ ($r \in \mathbf{Z}$), which avoids the use of the bit mask $0^{w-\beta m}1^{\beta m}$ in Algo1 because every "<<" operation in Algo2 can implicitly do this.

To see the suggested algorithm is essentially a variant of the Karp-Rabin algorithm, consider the line 17 of the pseudo-code of Karp-Rabin algorithm, by properly choosing d and q, all the arithmetic operations can be replaced by bitwise operations:

$$t_{s+1} \leftarrow (((t_s \ \& \ H) << D) \mid T[s+m+1]) \ \& \ Q.$$

And equivalently

$$t_{s+1} \leftarrow ((t_s << D) \ \& \ Q) \mid T[s+m+1].$$

Then it makes no difference with line 18 of Algo1 except that Algo1 does $T[s+m+1] \ \& \ 0^{w-\beta}1^\beta$ first, which means our algorithms can be seen as a special implementation of Karp-Rabin algorithm by adaptively adjusting the parameters and optimally pre-handling the input strings.

IV. Theoretical Analysis

In this section we prove that the two proposed algorithms have a linear expected running time, and derive their collision probability for a better understanding of their difference in performance. Before the proof, two assumptions are made:

- All the strings are randomly built under a Bernoulli model in which each letter in Σ occurs with the same probability.
- $m \leq 2^w$, which is the case of most applications.

Lemma 1: When Algo1 is executed, the probability of $t_s = p, s \in \{0, 1, \cdots, n-m-1\}$ is

$$Pr(t_s = p) = \frac{1}{\min(|\Sigma|^m,\ 2^{\beta m})}. \tag{1}$$

Proof: The fingerprints t_s and p contain βm bits totally, with each character contributing a binary sequence of β bits whose number of combinations is $\min(|\Sigma|,\ 2^\beta)$. Because of the equal probability of letters, we can clearly get the answer by the product rule. ∎

Lemma 2: When Algo2 is executed, the probability of $t_s = p, s \in \{0, 1, \cdots, n-m-1\}$ is

$$Pr(t_s = p) = \frac{1}{\min(|\Sigma|^{m'},\ 2^{\beta m'})}. \tag{2}$$

Proposition 1: The expected running time of Algo1 is $O(n+m)$.

Proof: The Algo1 uses $O(n+m)$ time to compute the fingerprints of the pattern and the text.

When $|\Sigma| \leq 2^\beta$, the information of the strings is all packed in the fingerprints as we mentioned before, no extra check needed, thus the running time is $O(n+m)$. Otherwise, we take $O(m)$ time to check a match, and the expected times of such checking is $\frac{1}{2^{\beta m}} \cdot (n-m+1)$ according to Lemma 1. Notice that $2^{\beta m} = 2^{\lfloor \frac{w}{m} \rfloor \cdot m} > 2^{w-1}$ and $m \leq 2^w$, so the expected running time is

$$O(n+m) + O(m) \cdot \frac{1}{2^{\beta m}} \cdot (n-m+1) = O(n+m). \tag{3}$$

∎

Proposition 2: The expected running time of Algo2 is $O(n+m')$.

Theorem 1: When Algo1 is executed, the probability of a false match is

$$Pr(collision) < \frac{1}{2^{w-1}}. \tag{4}$$

Proof: The probability of a match that $P[1..m] = T[s+1..s+m]$ is $\frac{1}{|\Sigma|^m}$. And by Lemma 1, we get the probability of a false match

$$\begin{aligned} Pr(collision) =& \frac{1}{\min(|\Sigma|^m,\ 2^{\beta m})} - \frac{1}{|\Sigma|^m} \\ =& \max(0,\ \frac{1}{2^{\lfloor \frac{w}{m} \rfloor \cdot m}} - \frac{1}{|\Sigma|^m}) \\ <& \frac{1}{2^{\lfloor \frac{w}{m} \rfloor \cdot m}} \\ \leq& \frac{1}{2^{w-1}}. \end{aligned}$$

∎

Theorem 2: When Algo2 is executed, the probability of a false match is

$$Pr(collision) < \max(\frac{1}{|\Sigma|^{m'}},\ \frac{1}{2^w}). \tag{5}$$

Proof: The probability of a match that $P[1..m] = T[s+1..s+m]$ is $\frac{1}{|\Sigma|^m}$. By Lemma 2, we get the probability of a false match

$$\begin{aligned} Pr(collision) =& \frac{1}{\min(|\Sigma|^{m'},\ 2^{\beta m'})} - \frac{1}{|\Sigma|^m} \\ =& \max(\frac{1}{|\Sigma|^{m'}},\ \frac{1}{2^w}) - \frac{1}{|\Sigma|^m} \\ <& \max(\frac{1}{|\Sigma|^{m'}},\ \frac{1}{2^w}). \end{aligned}$$

∎

From Fig. 2, we know that the property of the collision probability of Algo2, which is not stable in all cases, is different from that of Algo1.

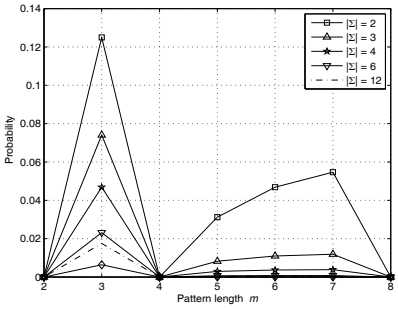

Fig. 2. Characteristics of m and the collision probability of Algo2 in different alphabet, where $w = 32$. See that $\frac{1}{2^w}$ is negligible here, so the curves are nearly the same as $\frac{1}{|\Sigma|^{m'}} - \frac{1}{|\Sigma|^m}$. Therefore, they go to 0 when w mod $m = 0$ ($m = m'$ in other words).

V. Experimental results

The previous sections indicate that Algo2 has a larger probability of a false match while its hash process is more desirable. Actually, in the experiments, there are only two cases that Algo1 has a superior performance: (1) $m \leq 7$ with $|\Sigma| = 2$. (2) $m \leq 3$ with $|\Sigma| \leq 6$. And it can be seen in Fig. 2 that the two cases are exactly the situation where the collision probability of Algo2 ≥ 0.02.

Therefore, in the cross comparison of the well know algorithms, we used a combination of two algorithms called **Algo3**, which runs Algo1 in above two cases, and otherwise Algo2. Additionally, from line 2, 3 of the Algo2, we know that β has only $\log_2 w$ different values. To maximize the performance of the algorithm, our implementation uses $\log_2 w$ subfunctions in which different values of β are treated as constants.

The referential algorithms are implemented according to the suggestion in [11]. Their implementation of Karp-Rabin algorithm is a particular fast version, which choose $d = 2$ and $q = 2^w$, rendering bitwise operations applicable.

The experiments were performed on a computer under Linux 2.6.22 and all the programs were compiled by gcc 4.3 using O1 optimization. We first built a text of 10M randomly. Then different algorithms were employed to search 300 random patterns of length m for all matches in the text, respectively. The comparison among their total running time is given in Table II and Fig. 3.

TABLE II
Experimental results in running time on alphabets of size 4, 8, 16, 32. (time in seconds). Hor is abbreviated from Horspool, KR is Karp-Rabin and SO is short for Shift-Or.

	$\|\Sigma\| = 4$						$\|\Sigma\| = 8$					
m	Hor	BOM	KMP	KR	SO	Algo3	Hor	BOM	KMP	KR	SO	Algo3
2	32.9	40.6	40.8	34.9	23.2	**18.4**	26.1	30.4	35.4	30.9	23.3	**19.1**
4	23.0	26.4	42.2	30.0	23.2	**18.5**	**15.2**	19.0	36.0	28.6	23.3	18.3
6	19.1	18.6	42.0	28.4	23.3	**18.3**	**11.5**	13.7	36.0	28.1	23.2	18.1
8	17.5	**14.9**	41.8	27.9	23.2	18.1	**9.8**	10.7	35.9	27.8	23.3	18.1
16	16.7	**8.3**	41.9	27.9	23.5	18.2	7.3	**5.7**	35.8	27.8	23.2	18.1
32	16.4	**4.8**	41.9	27.8	23.2	18.0	6.6	**3.4**	35.8	27.8	23.3	18.1
	$\|\Sigma\| = 16$						$\|\Sigma\| = 32$					
m	Hor	BOM	KMP	KR	SO	Algo3	Hor	BOM	KMP	KR	SO	Algo3
2	23.3	25.9	33.2	29.1	23.3	**18.3**	22.0	23.6	31.9	28.4	23.3	**18.1**
4	**12.5**	14.9	33.1	28.1	23.3	18.1	**11.5**	12.8	32.0	28.0	23.3	18.1
6	**8.9**	10.9	33.1	27.9	23.2	18.1	**7.9**	9.1	32.0	27.8	23.3	18.1
8	**7.1**	8.4	33.1	27.8	23.3	18.1	**6.2**	6.9	32.0	27.8	23.3	18.1
16	**4.5**	4.8	33.2	27.8	23.3	18.1	**3.6**	4.2	32.0	27.9	23.3	18.1
32	3.5	**2.8**	33.2	27.8	23.3	18.1	**2.3**	2.5	31.9	27.8	23.2	18.1

Our implementation performed well in the situation in which the pattern length and the alphabet size are both small, and about 28% faster than Shift-Or algorithm in all cases. This is because the Shift-Or algorithm needs to refer one more table in the matching process.

VI. Conclusion

Building on Karp-Rabin algorithm, we presented a string matching algorithm which outperforms other prefix-based approaches. In addition, it is even more efficient for small alphabets, which allows it a faster algorithm in searching on DNA and amino acid sequences.

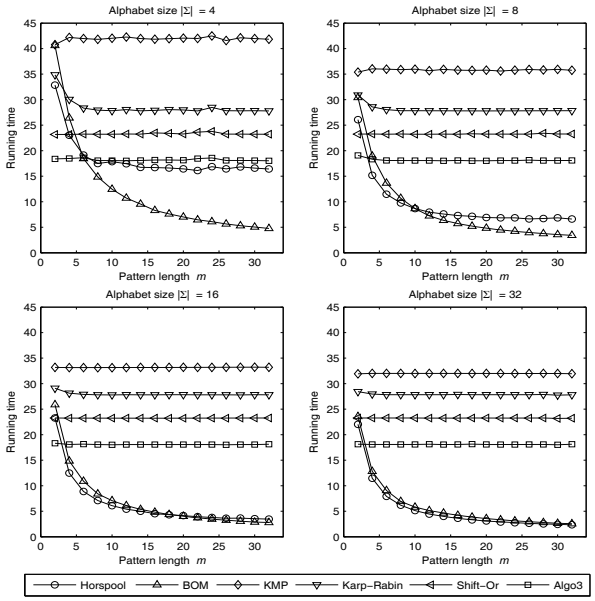

Fig. 3. Experimental results in running time on alphabets of size 4, 8, 16, 32. The X-axis represents the length of the pattern and the Y-axis stands for the total search time in seconds.

Future work is to extend the hash function. For example, the key parameter β could be calculated differently for different applications. Moreover, having noted that the fingerprints produced by our algorithms still preserve the information of characters (LSB) orderly, the idea may also serve to solve approximate string matching problems efficiently.

Acknowledgment

We greatly thank Mathieu Raffinot for his enthusiastic help.

References

[1] D. E. Knuth, J. James H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[3] G. Navarro and M. Raffinot, *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. New York, NY, USA: Cambridge University Press, 2002.

[4] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *SIGIR Forum*, vol. 23, no. SI, pp. 168–175, 1989.

[5] R. N. Horspool, "Practical fast searching in strings." *Software Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[6] G. Navarro and M. Raffinot, "Fast and flexible string matching by combining bit-parallelism and suffix automata," *J. Exp. Algorithmics*, vol. 5, p. 4, 2000.

[7] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching," in *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*. London, UK: Springer-Verlag, 1999, pp. 295–310.

[8] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 2001.

[10] G. H. Gonnet and R. A. Baeza-Yates, "An analysis of the Karp-Rabin string matching algorithm," *Inf. Process. Lett.*, vol. 34, no. 5, pp. 271–274, 1990.

[11] C. Charras and T. Lecroq, *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004.