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 $\Sigma$, 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).

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.

$\Sigma$ 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 $(b_{\ell}...b_1)$ 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 $1^30^4$). 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 $P_r(A)$.

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: $\triangleright$ Preprocessing
2: $p \leftarrow 0$
3: $t_0 \leftarrow 0$
4: $h \leftarrow d^{m-1} \bmod q$
5: for $i \leftarrow 1$ to $m$
6: $p \leftarrow (d \cdot p + P[i]) \bmod q$
7: $t_0 \leftarrow (t_0 <\beta \beta) \cdot (T[i] \bmod 0^{w-\beta,1})$
8: end for
9: $\triangleright$ Searching
10: for $s < 0$ to $n - m$
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: $t_{s+1} \leftarrow (d(t_s \cdot T[s+1] \cdot h) + T[s+m+1]) \bmod q$
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>
<thead>
<tr>
<th>Character</th>
<th>ASCII Value</th>
<th>Binary</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>76</td>
<td>1001100</td>
<td>01100</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
<td>1100001</td>
<td>00001</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100001</td>
<td>00101</td>
</tr>
<tr>
<td>i</td>
<td>108</td>
<td>1101100</td>
<td>01100</td>
</tr>
</tbody>
</table>

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

#### Algo1($T$, $n$, $P$, $m_s$, $|\Sigma|$)

1: $\triangleright$ Preprocessing
2: return an error if $m > w$
3: $\beta \leftarrow \left\lceil \frac{w}{m} \right\rceil$
4: $p \leftarrow 0$
5: $t_0 \leftarrow 0$
6: for $i \leftarrow 1$ to $m$
7: $p \leftarrow (p <\beta \beta) \cdot (P[i] \bmod 0^{w-\beta,1})$
8: $t_0 \leftarrow (t_0 <\beta \beta) \cdot (T[i] \bmod 0^{w-\beta,1})$
9: end for
10: $\triangleright$ Searching
11: for $s < 0$ to $n - m$
12: if $p = t_s$ then
13: if $|\Sigma| \leq 2^\beta$ 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: $t_{s+1} \leftarrow ((t_s <\beta \beta) \cdot 0^{w-\beta,1}) \bmod (T[s+m+1] \bmod 0^{w-\beta,1})$
19: end if
20: end for

#### Algo2($T$, $n$, $P$, $m$)

1: $\triangleright$ Preprocessing
2: $m' \leftarrow \min(w, 2^{\log_2 m})$
3: $\beta \leftarrow \frac{w}{m'}$
4: $p \leftarrow 0$
5: $t_0 \leftarrow 0$
6: for $i \leftarrow 1$ to $m'$
7: $p \leftarrow (p <\beta \beta) \cdot (P[i] \bmod 0^{w-\beta,1})$
8: $t_0 \leftarrow (t_0 <\beta \beta) \cdot (T[i] \bmod 0^{w-\beta,1})$
9: end for
10: $\triangleright$ Searching
11: for $s < 0$ to $n - m$
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: $t_{s+1} \leftarrow (t_s <\beta \beta) \cdot (T[s+m+1] \bmod 0^{w-\beta,1})$
19: end if
20: end for

The two algorithms share the same structure. They first calculate $\beta$ according to the size of $\Sigma$ and the length of the pattern $P$ (lines 1 ~ 3), then compute the fingerprints of the pattern and the first block of text (lines 4 ~ 9). At last they maintain a loop that iterates $s$ from 0 to $n - m$ (lines 10 ~ 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 Alg1, Alg2 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 $m' = 2^r$ ($r \in Z$), which avoids the use of the bit mask $0^{w-\beta}1^{\beta m}$ in Alg1 because every "$<$" operation in Alg2 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:

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

And equivalently

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

Then it makes no difference with line 18 of Alg1 except that Alg1 does $T[s + m + 1]$ & $0^{w-\beta}1^{\beta m}$ 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 $\Sigma$ occurs with the same probability.
- $m \leq 2^w$, which is the case of most applications.

Lemma 1: When Alg1 is executed, the probability of $t_s = \{p, s \in \{0, 1, \ldots, n - m - 1\}$

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

Proof: The fingerprints $t_s$ and $p$ contain $\beta m$ bits totally, with each character contributing a binary sequence of $\beta$ 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 Alg2 is executed, the probability of $t_s = \{p, s \in \{0, 1, \ldots, n - m - 1\}$

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

Proposition 1: The expected running time of Alg1 is $O(n + m)$.

Proof: The Alg1 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 $2^{\beta m} = 2^{\frac{\beta m}{m}} = 2^{w-1}$ 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 Alg2 is $O(n + m)$.

Theorem 1: When Alg1 is executed, the probability of a false match is

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

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

$$Pr(\text{collision}) = \frac{1}{\min(|\Sigma|^m, 2^{\beta m})} - \frac{1}{|\Sigma|^m}$$

$$= \max(0, \frac{1}{2^{\beta m}}) - \frac{1}{|\Sigma|^m}$$

$$\leq \frac{1}{2^{w-1}}.$$

Theorem 2: When Alg2 is executed, the probability of a false match is

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

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

$$Pr(\text{collision}) = \frac{1}{\min(|\Sigma|^m, 2^{\beta m})} - \frac{1}{|\Sigma|^m}$$

$$= \max(\frac{1}{|\Sigma|^m}, \frac{1}{2^{\beta m}}) - \frac{1}{|\Sigma|^m}$$

$$\leq \max(\frac{1}{|\Sigma|^m}, \frac{1}{2^w}).$$

From Fig. 2, we know that the property of the collision probability of Alg2, which is not stable in all cases, is different from that of Alg1.
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 \( \geq 0.02 \).

Therefore, in the cross comparison of the well known 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 \( \beta \) 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 \( \beta \) 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 \( q = 2^m \), 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.

| \( |\Sigma| = 4 \) | \( |\Sigma| = 8 \) |
|---|---|
| \( m \) | 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 28.6 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 |
| 10 | 16.7 8.3 41.9 27.9 23.5 18.2 | 7.3 5.7 35.8 27.8 23.2 18.1 |
| 12 | 16.4 4.8 41.9 27.8 23.2 18.0 | 6.6 3.4 35.8 27.8 23.3 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.

ACKNOWLEDGMENT

We greatly thank Mathieu Raffinot for his enthusiastic help.

REFERENCES