AN ANALYSIS OF THE KARP–RABIN STRING MATCHING ALGORITHM

Gaston H. GONNET and Ricardo A. BAEZA-YATES

Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

Communicated by D. Gries
Received 2 May 1989

We present an average case analysis of the Karp–Rabin string matching algorithm. This algorithm is a probabilistic algorithm, that adapts hashing techniques to string searching. We also propose an efficient implementation of this algorithm.

Keywords: String searching, signatures, average case analysis

1. Introduction

The string matching problem consists of finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet. It is well known that to search for a pattern of length \( m \) in a text of length \( n \) (where \( n > m \)) the search time is \( O(n + m) \) in the worst case (for fixed \( m \)) [4].

The classical algorithms used to search strings in linear time, are the Knuth–Morris–Pratt algorithm [3] and the Boyer–Moore algorithm [1].

A different approach to string searching is to use hashing techniques. All that we need to do is to compute the hash function of each possible \( m \)-character substring in the text and check if it is equal to the hash function of the pattern.

Karp and Rabin [2] found an easy way to compute these hash functions efficiently for the hashing function \( h(k) = k \pmod{q} \), where \( q \) is a large prime. Their method is based on computing the hash function for position \( i \) given its value for position \( i - 1 \). The algorithm then requires time proportional to \( n + m \) in almost all the cases, without using extra memory. Note that this algorithm finds positions in the text which have the same hashing value as the pattern, so, to be sure that there is a match, we should do a direct comparison of that text with the pattern. That is, the algorithm is probabilistic. Using a large value for \( q \) makes collisions unlikely (the probability of a random collision is \( O(1/q) \)).

Theoretically this algorithm could still require \( mn \) steps on the worst case, if we check each potential match and we have too many matches or collisions. In all our empirical results we observed only 3 collisions in \( 10^7 \) computations of the hash function (generally, for big alphabets).

2. The Karp–Rabin algorithm

The hash function represents a string as a base-\( d \) number, where \( d = c \) is the number of possible characters. To obtain the hash value of the next position, only a constant number of operations are needed. The code for the case \( d = 128 \) (ASCII) and \( q = 16647133 \) based in [5] for a word size of 32 bits is given in Fig. 1. By using \( d \) a power of 2 (\( d > c \)), all the multiplications by \( d \) can be computed as shifts. The prime \( q \) is chosen as large as possible, such that \( (d + 1)q \) does not cause overflow [5]. We also impose the condition that \( d \) is a primitive root mod \( q \). This provides that the hashing function has maximal cycle, that is,

\[
\min_k \left( d^k \equiv 1 \pmod{q} \right) = q - 1.
\]
rksearch(text, n, pat, m)  (;;) Search pat[1..m] in text[1..n]  * /
char text[], pat[];  (;;) (0 < m <= n)  * /
int n, m;
{
  int h1, h2, dM, i, j;
  dM = 1;
  for (i = 1; i <= m; i + +)  dM = (dM * D) % Q;  ++) Compute the hash value  * /
  h1 = h2 = 0;  ++) of the pattern and of  * /
  for (i = 1; i <= m; i + +)  ++) the beginning of the  * /
  {  ++) text  * /
      h1 = ((h1 * D) + pat[i]) % Q;
      h2 = ((h2 * D) + text[i]) % Q;
  }
  for (i = 1; i <= n - m + 1; i + +)  ++) Search  * /
  {
    if (h = = h2)  ++) Potential match  * /
    {
      for (j = 1; j <= m & & text[i - 1 + j] = = pat[j];  ++) check  * /
        if (j > m)  ++) true match  * /
          Report_match_at_position(i);
      }
      h2 = (h2 * (Q * D) - text[i] * dM) % Q;  ++) update the hash  * /
      h2 = ((h2 * D) + text[i + m]) % Q;  ++) value of the text  * /
  }
}

Fig. 1. The Karp–Rabin algorithm.

Thus, the period of the hashing function is much bigger than \( m \) for any practical case.

3. Average case analysis

Karp and Rabin [2] show that

\[
\text{Prob} \{ \text{collision} \} \leq \frac{\pi(m(n - m + 1))}{\pi(M)}
\]

where \( \pi(x) \) is the number of primes \( \leq x \), for \( m(n - m + 1) \geq 29 \), \( c = 2 \) and \( q \) a random prime \( \leq M \). They use \( M = O(mn^2) \) to obtain a probability of collision of \( O(1/n) \). However, in practice, the bound \( M \) depends on the word size used for the arithmetic operations, and not on \( m \) or \( n \). For this reason, our analysis follows a different approach, and is based in a more realistic model of computation.

**Lemma 3.1.** The probability of two different random strings of the same length having the same hash value is

\[
\text{Prob} \{ \text{collision} \} = \frac{1}{q} + O\left(\frac{1}{c^m}\right) \leq \frac{1}{q}
\]

for a uniform hashing function.

**Proof.** Let \( m \) be the length of the string and \( c \) the alphabet size. There are \( c^m \) different strings of length \( m \). Let \( d = c \) be a primitive root mod \( q \). The hashing function is

\[
h(s) = \left( \sum_{i=0}^{m-1} s_i c^i \right) (\text{mod} \ q)
\]

and all possible values from 0 to \( c^m - 1 \) are possible before taking modulus. Let \( k = \lfloor c^m/q \rfloor \) the number of times that \( q \) is contained in \( c^m \). If we choose a string at random, we have two cases:
(1) with probability \( p = (k + 1)(c^m - qk)/c^m \) we have other \( k \) possible hash values equal to the one chosen, and

(2) with probability \( 1 - p \) we only have \( k - 1 \) other values that can be equal.

Then, the probability of a collision is

\[
\text{Prob} \{ \text{collision} \} = \frac{kp}{c^m - 1} + \frac{(k + 1)(1 - p)}{c^m - 1}
\]

\[
= \frac{k(k + 1)q}{c^m(c^m - 1)}
\]

\[
= \frac{1}{q} - \frac{1}{c^m - 1} + O(q^{-1}c^{-m})
\]

\[
< \frac{1}{q}
\]

**Theorem 3.2.** The expected number of text-pattern numerical comparisons performed by the Karp–Rabin algorithm to search a pattern of length \( m \) in a text of length \( n \) is

\[
\bar{C}_n \approx H + m \frac{1 - \frac{1}{q}}{c^m} + O(1/c^m),
\]

where \( H \) is the cost of computing the hash function expressed as comparison units.

**Proof.** We have that

\[
\bar{C}_n = (n + m)H + \frac{(n - m + 1)m}{c^m}
\]

\[
+ \left(1 - \frac{1}{c^m}\right)(n - m + 1)m\text{Prob} \{ \text{collision} \}
\]

\[
+ O(1),
\]

where the first term is the cost of computing of all the hashing values and the second term is the number of comparisons used to check all the matches found on average. For every position that does not match, we have to perform \( m \) comparisons if the two random strings have the same hashing value. We bound this probability by using Lemma 3.1.

The empirical results are compared with our theoretical results in Table 1 using \( H = 1 \). They agree very well for all alphabet sizes. In practice, the value of \( H \) is bigger, due to the multiplications and the modulus operations of the algorithm. However, this algorithm becomes competitive for long patterns.

### 4. An efficient implementation

We can avoid the computation of the modulus function at every step, by using the implicit modular arithmetic given by the hardware. That is, we use the maximum value of an integer (word size) for \( q \).

The value of \( d \) is selected such that \( d^k \mod 2^r \) has maximal cycle (cycle of length \( 2^{r-2} \)) for \( r \) from 8 to 64, where \( r \) is the size in bits of a word. For example, an adequate value of \( d \) is 31.

With these changes, the evaluation of the hashing function in every step (see Fig. 1) is

\[
h2 = h2 \times D - \text{text}[j-m] \times dM + \text{text}[i+m];
\]

\[
\text{// update the hash value}
\]

and overflow is ignored. In this way, we use two multiplications instead of one multiplication and two modulus operations.

### References


