Optimizing Rolling Hash in Rabin-Karp Pattern Matching Algorithm with Randomized Modulo

Afrizal Fikri / 13513004
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
afrizal_f@students.itb.ac.id

***Abstract*—Pattern matching has widespread application from the word processing, to the biometric recognition. Many algorithm's been developed. One of those is Rabin-Karp algorithm which easy to implement and relatively fast than other algorithm. Since this algorithm using hash as the main component and also limitation of amount of hash key and data representation in computer, we need to minimizing collision between different patterns. We will look that randomized modulo give many advantages.**

***Keywords*—rolling hash, collision, hash value, false match.**

I. INTRODUCTION

From the very beginning, pattern has appear in various things. Those pattern data need to be analyzed to become valuable information. Then this information can be used for many purpose. For example, using goods prices pattern government could make proper policies in monetary and economic.

One of most important operation in pattern recognizing is matching operation. We look for an already known pattern in a long pattern which found, then make conclusion from the data that gotten.

The most simple and widely studied pattern matching problem is the following: given a pattern *X* of length *n* and a text *Y* of length $m \geq n$, find all occurrences of *X* as a consecutive substring of *Y*. The easiest solution is to brute force all position of pattern *Y* in *X* and check whether *Y* is substring of *X* in this position.

In 1974, Donald Knuth and Vaughan Pratt, also independently James H. Morris developed more efficient algorithm called Knuth-Morris-Pratt (KMP) algorithm. Then Aho-Corasick (1975) and Boyer-Moore (1977) invented algorithm. *O(n)* registers to store a table of pointers. The characters of the text *Y* can come in a stream and require no storage. But for fast implementation it is useful to have portions of *Y* in main memory [1].

In 1987, Richard M. Karp and Michael O. Rabin propose another linear time algorithm which need only constant number of register and a substring of length *n* of the text in main memory [1]. The algorithm using hash function based on polynomial which similar to number basis concept.

A. Pigeonhole Principle

In mathematics, pigeonhole principle is obvious fact which occurred in many counting problems. This principle originally states that if there are *n* item put into *m* containers, with $m < n$, there must be minimal a container which contain more than one item inside**.**

This theorem can be generalized by considering each container can obtain more than one items. So, for *n* items put into *m* containers, there must be $\lceil n/m \rceil$ item(s) for each container.

B. Number Theory

1. Prime Number

Briefly, prime number is a positive number more than 1 whose only 2 divisor, 1 and itself. Positive numbers other than prime number called composite numbers. So, all composite number is consist of one or more prime number(s).

From the fundamental theorem of arithmetic, we know that all of positive integer more than 1 can be factorized to one or more prime numbers. Simply, it can be derived from the composite number's properties above. Since the composite number consist of prime numbers and, obviously prime number consist of prime one itself, so the theorem above valid.

2. Modular arithmetic

Integers can be written in the division form with divisor, quotient, and residual (remainder). Modular arithmetic concern to this remainder properties of integer. Formally suppose we have a number *n* and *m*. We can rewritten *m* as form of division by *n* become

$$m = nq + r \qquad (1)$$

with *q* and $r < n$ are non-negative integers. Equation above called Euclid Theorem [3].

With Euclid Theorem we can find greatest common divisor of two number. Euclid also marks that $GCD(m, n) = GCD(n, r)$. This implies a recurrence relation of GCD between two numbers. Also we can

determine coprimality between two numbers from the fact that 2 coprime numbers have GCD equal 1.

There are another way to represent modulo of a number from another number. That's congruency. Slightly different from the previous form, congruency has the form

$$m \equiv r \bmod n \qquad (2)$$

Since we only concern about the division remainder, we can throw out q (quotient).

There are some identity about modular congruency:

- $(a + b) \equiv (p + q) \bmod n$ (3)
- $(a - b) \equiv (p - q) \bmod n$ (4)
- $ab \equiv pq \bmod n$ (5)

with $a \equiv p \bmod n$ and $b \equiv q \bmod n$.

C. Hashing

Hash is a storing and searching method which run in complexity near constant [1]. Almost all of database system nowadays using varies hash function to store their data.

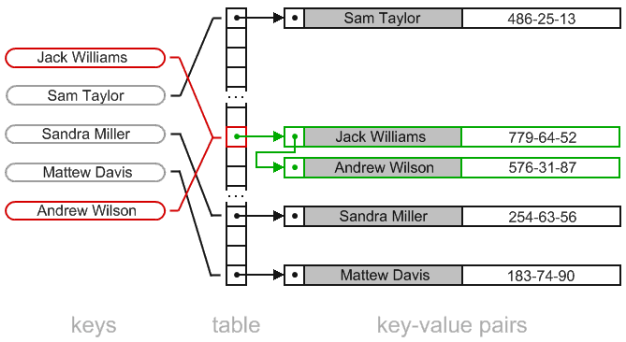

Figure 1.2 Hash table illustration with hash chaining resolution
Source: http://www.algolist.net/img/hash-table-chaining.png

Besides hashing there are many ways to store dataset efficiently. For example, using tree data compression. But we only concern to the hash function using here.

Usually hash is generated using some function which mapping a key to a specific unique value, sometimes index of storage table. Hash function generated value based on key certain properties.

For example, the simplest hash function is by mapping key based on modulo by a numerical value

$$h(n) = n \bmod m, \qquad m > 0 \qquad (6)$$

As can be seen from the equation above, this function only provide m distinct hash value (from 0 to m-1). By pigeonhole principle mention above, must be a slots which filled by 2 or more different key. This phenomenon also known as collision.

Varied ways to get rid this collision. Those usually called collision resolution. Most often resolution used is open addressing / close hashing and hash chaining. Open addressing is resolution by moving entry whose origin place had taken. While hash chaining is by making list of elements in collision indices so every entry can occupy the proper place.

These two method cannot used to our string matching problem because these method take action on storage management. Since our algorithm only need the hash value to be compared, we need another resolution method.

II. Rabin-Karp Algorithm

Before we talk about the Rabin-Karp algorithm we need to know how the string matching normally works. Suppose we have a pattern T with length m and string S with length $n \geq m$. First, we will put T sticking to beginning of S. Then we check whether part of S which beside the T is equal with T or not. If not, slide pattern T to next character. So on until the end of S or the pattern has found in S. This operation run in $O(n)$.

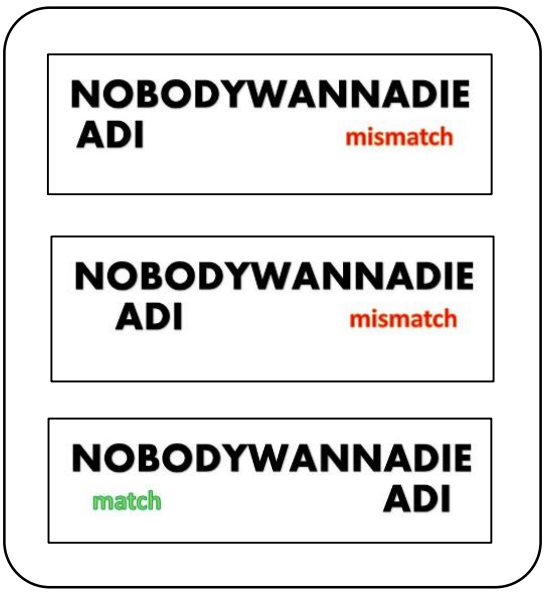

Figure 2.1 String matching illustration from first to last

Each comparison of substring, we have to check every letter of substring with pattern one by one. This operation take $O(m)$ since we have iterate at most m characters of pattern. So overall comparison of this naïve method take running time $O(nm)$.

Inefficiency occurred because we check the same character over and over. We could reduce running time by

save last position which have same character with last character checked before. KMP algorithm used this approach with preprocessing table to save return position after each checking in current position failed.

Another way to make a fast comparison is by turning the string into an integer value then compare these two integers to check equality of strings. Each character can be assigned to a different value of integer. So there are no two different string have the same integer value.

One of the way of turn the string into integer is using polynomial hash. Suppose we want to turn a string *S* consist only of uppercase alphabet. We could assign each alphabet character with a number 0 – 25. Then each alphabet depend of the position has a multiplier factor to keep position of each character in the integer value distinguished. From the last character multiplier factor is p^0, then previous character multiplied by p^1, and so on until the first character multiplied by $p^{|S|-1}$.

$p^{\|S\|-1}$	$p^{\|S\|-2}$	$p^{\|S\|-3}$	…	p^2	p^1	p^0

Figure 2.2 Multiplier factor for each character position

Hence, the hash formula:

$$h(S) = \sum_{i=1}^{|S|} S_i . p^{|S|-i} \qquad (7)$$

with S_i refers to value assigned to *i*-th character is *S* instead of character itself and *p* is multiplier base.

Hash value of the string can be very large when the string is long. Even for string with length = 15, the hash value can reach 26^{15} = 1677259342285725925376. Compare to maximum value of unsigned 64-bit integer which is only $2^{64} - 1$ = 18446744073709551615 the hash value cannot be represented by 64-bit integer.

To make hash value can be covered by 64-bit integer we have to modulo that formula with a 64-bit integer number too, supposed by *k*. Hence, the formula become:

$$h(S) = \left(\sum_{i=1}^{|S|} S_i . p^{|S|-i} \right) mod\ k \qquad (8)$$

and by modulo identity formula above can be rewritten:

$$h(S) = \left(\sum_{i=1}^{|S|} (S_i . p^{|S|-i}\ mod\ k) \right) mod\ k \quad (9)$$

Now we start discuss the main algorithm. Suppose we want to check a substring of a string *S* with a pattern *T*. We have to calculate the hash value of substring and compare to hash of pattern. If does not match, we get to the next substring and calculate again the hash value. Until the pattern found or reach the end of string. But calculate hash value every substring take extra time even compared to naïve method mention before.

Note that while going to next character, we only need subtract hash value with first character, shifting hash value to the left one, and add the last character. Subtracting the last character can be done by compute the proper multiplier factor than multiply with first character value, then subtracting it from current hash value. Operation can be optimized more by storing the multiplier factor for deletion because the pattern length is constant. So, we don't need to compute many times. Shifting left one can be done only by multiply current hash with multiplier base (*p*). Because when we shifting left, component which changed are only multiplier factor. Then the last one just add the value of last character. Nothing to do with the multiplier factor because the last character have multiplier factor equal $p^0 = 1$. This strategy also known as rolling hash.

This is pseudo code of the "raw" Rabin-Karp algorithm

```
function Rabin-Karp(S, T : string) : boolean
{ returning whether T occurred in the S
substring or not }
{ h : hash of current substring of S }
{ hp : hash of string T }
  h ← 0
  hp ← 0
  for i : 1 to |T| do
    h ← (h × p + Si) mod k
  endfor
  for i : 1 to |T| do
    hp ← (hp × p + Ti) mod k
  endfor
  for i : |T| to |S| do
    if h = hp then
      return true
    endif
    if i < |S| then
      h ← (h + k – p^|T|-1 mod k) mod k
      h ← (h × p + Si) mod k
    endif
  endfor
  return false
```

Ideally, we assumed that if hash of substring and pattern equal, than those two is same too. But notice that possible hash value is limited only to *k*. While the hash value before modulo should be $26^{|T|}$. So, for some length of *T* the hash value before the modulo will be excess the limit *k*. By the

pigeonhole principle, must be a hash value which represented two or more different string, also known as false match.

Hence, we add an additional checking. When the hash value of two is equal, we consider to check whether two string is same or not instead just conclude the two is same.

Here is the "improved" rolling hash pseudo code:

```
function Rabin-Karp(S, T : string) : boolean
{ returning whether T occurred in the S
substring or not }
{ h : hash of current substring of S }
{ hp : hash of string T }
  h ← 0
  hp ← 0
  for i : 1 to |T| do
    h ← (h × p + Si) mod k
  endfor
  for i : 1 to |T| do
    hp ← (hp × p + Ti) mod k
  endfor
  for i : |T| to |S| do
    if h = hp then {do additional check}
      same ← true
      for j : 1 to |T| do
        if Tj ≠ Si-1+j then
          same ← false
        endif
      endfor
      return same
    endif
    if i < |S| then
      h ← (h + k – p|T|-1 mod k) mod k
      h ← (h × p + Si) mod k
    endif
  endfor
  return false
```

The worst case is when for each comparison, we have equal hash value. So in worst case, running time of this algorithm still *O(nm)***.** We need to optimize more to suppress the collision.

III. REDUCING FALSE MATCH

Suppose for any fixed location *i*, the probability of an incorrect match is δ. Then by a union bound over $t - p + 1$ locations we perform the equality test, the probability of outputting some false positive is at most $(t - p + 1).\ \delta \leq t\delta$. If we want the final probability of error to be at most 1/2, we should ensure that $\delta \leq 1/(2t)$.

Notice the only randomness is in the choice of the random prime *k*. We make a mistake when the number represented by the p-bit substring $S[i \ldots (i+p-1)]$ (call this number a) is not equal to the number represented by the *p*-bit pattern *T* (call this second number b), but these two numbers are the same modulo the random *k*. By the definition of being equivalent modulo *k*, this means that *k* divides $|a - b|$. Now $|a - b|$ is also a p-bit number, so it can have less than p distinct prime divisors. (Each prime divisor is at least 2, and $|a - b| < 2^p$). And if $k \mid (a - b)$, then *k* must have been one of these "bad" values, these prime divisors.

We would like to claim that choosing a uniformly random prime number *k* in the range {2, … , M}, the chance that we choose one of (at most) *p* bad values is smaller than *1/(2t)*. For this, it suffices to choose *M* large enough such that there are at least *2pt* primes between 2 and *M*.

And while this is just an asymptotic statement, we also know that

$$\pi(n) \geq \frac{7n}{8 \ln n} \qquad (10)$$

with $\pi(n)$ is number of primes less or equal by *n*. This theorem was proved by Chebyshev back in 1848. Now setting M to equal, say, *10pt ln pt* ensures that $\pi(M) \geq 2pt$ for large enough *pt*, which proves the result.

That method is good enough. But if we want to reduce the error probability more, we could either pick several independent primes *k* or perform the string matches in parallel (claiming that there is a match at location *i* only when all the hash values match).

IV. CONCLUSION

The idea of randomize sometimes can be very useful. Since the randomize value can be varies, we also consider the worst case of using this algorithm. Anyway, the rolling hash matching still have important role in various field. Maybe in future, more efficient method will be developed and make a more productive world.

VII. ACKNOWLEDGMENT

I would thank to God for all resource He give to me for helping me finishing this paper. Also I would like to thank my lecturer, Mr. Rinaldi Munir and Mrs. Harlili for giving me knowledge in computer science, and discrete mathematic field especially. And to everybody who give me inspiration and support in this paper work.

REFERENCES

[1] Karp, Richard M.; Rabin, Michael O. (March 1987). "Efficient randomized pattern-matching algorithms". *IBM Journal of Research and Development* **31** (2). Retrieved on 8 Dec 2014. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.9502&rep=rep1&type=pdf.

[2] Paulo Ribenboim. (1995). *The New Book of Prime Number Records* (3rd ed.). New York: Springer-Verlag.

[3] K. H. Rosen. (2012), Discrete Mathematics and Its Applications 7th. New York: McGraw-Hill.

[4] Knuth, Donald (1998). *'The Art of Computer Programming'*. 3: *Sorting and Searching* (2nd ed.). Addison-Wesley.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 10 Desember 2014

Afrizal Fikri
13513004