Aviv Farag

Professor Boady W. Mark

CS502 - Data Structures & Algorithms

3 February 2022

## Karp-Rabin String Matching Algorithm

String matching algorithms play an important role in finding occurrences of a specific patterns within a larger text. Nowadays, they are being utilized by Plagiarism programs as well as language processing projects in Data Science. Finding occurrences of a pattern within a text is not a simple task. There is a naive algorithm (sometimes called Brute-force algorithm) which is simple, but not efficient since bigger patterns require more memory and longer run-time. Over the years, computer scientists tried to find better algorithms that significantly reduce time and memory complexity. Among them are Richard M. Karp and Michael O. Rabin that published a paper in 1987 presenting a randomized algorithm for string matching that is both time and memory efficient. We will present the problem as well as the algorithm, its implementation, and complexity in this paper.

## Background

In June, 1977, Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt published a paper describing their linear time algorithm for the string matching problem. Their algorithm skips some iterations based on partial matches (false match) it found in previous iterations, and thus reduce run-time [Knuth et al.]. Later on, in October 1977, Boyer and Moore developed an algorithm for fast string searching. This algorithm prepossess a table based on the pattern. The table holds numeric values that define how many steps to skip. Finally, the algorithm start matching characters from the right-most of each window [Boyer and Moore]. If there is no match, the algorithm skips according to the table, otherwise it compares items to the left. Both algorithms have memory complexity of $O(n)$ for fast implementation, where n is the length of the pattern. Another linear algorithm, developed by Zvi Galil and Joseph Seifaras, was published in May 1980. This algorithm have a memory complexity of $O(log(n))$ [Galil and Seiferas, "Saving Space in Fast String-Matching" ]. Galil and Seifaras published another paper in June 1983 describing another algorithm

they developed that runs in real-time and requires a constant number of registers ["Time-space-optimal string matching" ].

Richard M. Karp and Michael O. Rabin developed an algorithm that uses hash functions with the rolling technique in order to match the pattern in real-time and constant number of registers [Karp and Rabin]. They published their work in 1987.

## The Problem

The string matching problem is to find the first occurrence of a pattern X within a large text Y (Y is longer than X). This problem could be extended to find all occurrences of the pattern in the text. A straightforward and simple algorithm iterates over the text and compares characters as shown below:


Figure 1: Brute-Force algorithm

Here we iterate over the full text with a sliding window whose size is the same as the pattern. We ensure that each character matches in both the pattern and the window. If not, we increment the index by 1, to check the next window.

The complexity of such algorithm is $O(n \cdot m)$ where n is the length of X(pattern) and m is the length of Y(text). Larger patterns as well as larger texts will increase both running time and memory required for that algorithm. Therefore, various algorithms were developed for that problem. Although Rabin-Karp algorithm's worst case is also $O(n \cdot m)$ [Karp and Rabin], its average and best case are much better due to computing fingerprints for both the pattern and the windows. The fingerprint is shorter than the string

itself, and is an output of a Hash function which maps a string into an integer. Karp-Rabin algorithm utilizes the rolling technique which significantly improves time complexity [Karp and Rabin].

## Applications

The string matching algorithms have various applications in different domains. A common application for such an algorithm is in plagiarism detection programs [KiranShivaji and S] as well as in databases that use fingerprints to efficiently query data [Bille et al.]. It has also been used for comparing biological (DNA) sequences [Wise]. Another application is spam filtering since the algorithms can easily trace certain words within an email [Baranwal et al.].

## Implementation

As mentioned earlier, the Karp-Rabin algorithms use a rolling Hash function which is presented in the next page. There is a while loop to iterate over the text and extract windows with the same length as the pattern. For both strings (window and pattern), it first computes the hash. Then, the algorithm checks whether the hash value of both strings match. If it doesn't, then the strings do not match and we can continue to the next window. This logic might save time and memory since the examination is done using a shorter representation of the strings.

If the fingerprints match, it doesn't necessarily indicate that the strings actually match. There could be a case of matching fingerprints for different strings which is called False Match. There is a small chance of such case which also depends on the prime number we choose. Therefore, it iterates over each character to ensure the strings match. If they don't, it breaks the while loop and continue to the next window. If all characters match in both strings, then the while loop stops, and the variable *occur_index* holds the index of the first character in the window that matches the pattern. As mentioned earlier, this algorithm finds the first occurrence, but can be easily modified to find all occurrences of the pattern.

**Algorithm 1:** Karp-Rabin string matching algorithm

```
Input: Y: text
       X: pattern
Output: First occurrence of X in Y. If not found then -1
match ← False
m ← length(Y)
n ← length(X)
i ← 0
p ← randomly chosen prime number
occur_index ← −1
patternHash ← Hash(X, p)

while match = False and i < m − n do
    windowHash ← Hash(Y[i, i + n], p)

    if windowHash = patternHash then
        valid ← True
        temp ← 0

        while valid = True and temp < n do
            /* Compare each character to make sure there is a match */
            if X[temp] != Y[i + temp] then
                valid ← False
            else
                temp ← temp + 1
            end
        end
        if valid = True then
            match = True
            occur_index ← i
        end
    end
    i ← i + 1
end
```

Another version[1] of this algorithm includes an additional randomization aspect. In the case of a false match, the extended version changes the prime number to a different prime number which is also randomly chosen. It ensures that similar false matches will not occur. Changing prime number should also be taken into account in the rolling hash function.

[1] Not covered in this paper

**Rolling Hash Function**

Rolling hash function improves time complexity because it computes a window's fingerprint based on the previous one. For example, for a string of length 10, it subtracts the value of the first character (in the former window) and adds the value of the new character. Thus, it saves the time that it would have taken to compute the hash value of 8 character. That is true for each instance of a sub-string besides the first one. Let define X as a string of length $n$. We can represent X as an integer using the formula:

$$I(X) = \sum_{i=1}^{n} x_i \cdot \underbrace{\sigma^{n-1}}_{base}$$

Where $x_i$ is the character in the i-th position, and $\sigma$ is the base in which x is represented. By using a prime number, which is chosen randomly, we could compute a possible fingerprint for a given string:

$$H(X) = I(X) \ mod \ p$$

For example, let define n (pattern's length) as 4, and the text is: "This is an example". Also, we will use the characters ASCII codes which is base-256 and the prime number 151 for our calculation. Our first Hash represents the word "This" and is computed below:

$$Hash(\text{"}This\text{"}) = (84 \cdot 256^3 + 104 \cdot 256^2 + 105 \cdot 256^1 + 115 \cdot 256^0)\%151 = 147 \quad (1)$$

$$Hash(\text{"}his\ \ \text{"}) = (104 \cdot 256^3 + 105 \cdot 256^2 + 115 \cdot 256^1 + 32 \cdot 256^0 +)\%151 = 31 \quad (2)$$

So far we have represented strings by a fingerprint which is an integer number computed by the hash function defined above and the prime number 113. Next, we will see where is the rolling part and why it is an efficient way to update the fingerprint function. Below is the formal notation:

$$H(Y_{i+1}) = (\ \underbrace{H(Y_i)}_{former\ hash} - \underbrace{\sigma^{n-i} \cdot y_i}_{former\ first\ char}\ ) \cdot \sigma + \underbrace{y_{i+n}}_{new\ char}$$

As mentioned earlier, the first sub-string will be computed by converting each character to its ASCII code:

$$Hash(\text{"}This\text{"}) = ((((\underbrace{84 \cdot 256^3}_{T})\%151 + \underbrace{104 \cdot 256^2}_{h})\%151) + \underbrace{105 \cdot 256^1}_{i})\%151 + \underbrace{115 \cdot 256^0}_{s})\%151$$

$$Hash(\text{"}This\text{"}) = 147$$

Then, we can use the rolling formula in order to compute the next sub-string ("his "). We need to subtract the hash value of "T" from 147 (last Hash) and add the new character's value:

$$Hash(\text{"}his\ \text{"}) = (\underbrace{(\underbrace{147}_{last} - \underbrace{84 \cdot 256^3}_{T}\%151 \cdot 256)\%151}_{Hash(his)} + \underbrace{32}_{space})\%151$$

$$Hash(\text{"}his\ \text{"}) = 31$$

As we can see, the hash value of the second sub-string matches in both examples shown above. The first example was a direct computation that required converting each character to its ASCII code and then applying mod 151, whereas the second example shows subtraction of the left-most character, and adding the new character. Thus, by utilizing a few arithmetic steps we can efficiently update the hash function of a sub-string which might significantly reduce time complexity. Furthermore, the fingerprint is much shorter than the string itself as can be seen below:

$$Hash(\text{"}his\ \text{"}) = \underbrace{1,751,741,216}_{without\ mod} > \underbrace{31}_{with\ mod}$$

Moreover, a prime number is used in order to avoid an overflow as well as reduce occurrences of false match. In their paper, Karp and Rabin also proved that any number in a specific range can be used (both prime and non-prime)[2].

[2]The algorithm for choosing a random prime number is not presented in this paper

## Example

To illustrate the functionality of the algorithm an example is shown in the table below. The pattern is "just" and the text is "This is just an example". The prime number is 151, so the hash value of "just" is 106. The algorithm computes the hash value of each window using the rolling technique and compares the value to the pattern's hash:

| iteration | Text (**window**) | window hash | pattern hash | match |
|---|---|---|---|---|
| 0 | **This** is just an example | This<br>147 | | False |
| 1 | T**his** is just an example | his<br>31 | | False |
| 2 | Th**is i**s just an example | is i<br>75 | | False |
| 3 | Thi**s is** just an example | s is<br>20 | | False |
| 4 | This **is** just an example | is<br>11 | just<br>106 | False |
| 5 | This **is j**ust an example | is j<br>76 | | False |
| 6 | This i**s ju**st an example | s ju<br>127 | | False |
| 7 | This is **jus**t an example | jus<br>4 | | False |
| 8 | This is **just** an example | just<br>106 | | True |

Table 1: An example of the Karp-Rabin algorithm

## Complexity

The brute-force algorithm is a straightforward solution with complexity of $O(m \cdot n)$. Karp-Rabin algorithm was developed to improve this complexity by computing fingerprints that are a shorter representation of the strings. Thus, reducing run-time as well as memory. Although the worst case complexity remains $O(m \cdot n)$, the average case is $O(m + n)$.

Figure 2 shows the growth of the best case of both the Brute-force and Karp-Rabin algorithms where the length of the pattern is 10. We can see a great improvement in run-time for the Karp-Rabin algorithm. Worst case could happen if there are multiple false matches (sometimes referred to as Spurious Hits). For that reason, a prime number is being utilized and a modulo is taken in the hash function.

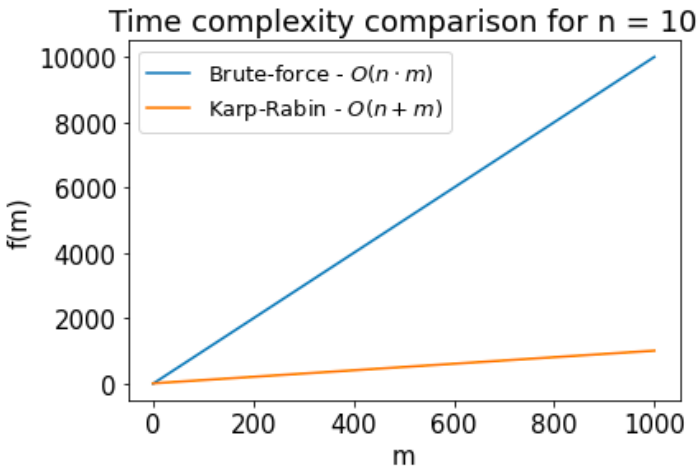


Figure 2: Compare best case growth of Brute-force and Karp-Rabin algorithms

## Current Methods

The Karp-Rabin algorithm presented in this paper use the same prime number while executing. Another version was proposed by Karp and Rabin in which the prime number is modified in the case that a false match occurs. It ensures that a similar false match is unlikely to occur by enhancing randomization. Run-time and memory usage is essentially the same as the algorithm presented in this paper. As mentioned in the background section, there are other algorithms such as KMP, and Boyer & Moore. Those are the current methods for string matching algorithms.

## Conclusions

String matching is not as easy task, but it is an important one in computer science. There are several algorithms developed for that task, all are trying to minimize both memory usage and total run-time. The algorithm that was reviewed in this paper was published in 1987 and has been widely used in various applications. Its worst case is the same as the brute-force algorithm, but it contains a randomized mechanism to avoid such case. Its average case (and also best case) is $O(n + m)$ which is a great improvement in run-time. Also, since it reduces string representations by computing hash value, it also requires less memory,so its memory complexity is constant $O(1)$.

Works Cited

Baranwal, Aastha, et al. "Spam Filtration using Boyer Moore Algorithm and Naïve Method." *International Journal of Computer Applications*, vol. 180, no. 42, May 2018, pp. 35–38. doi:10.5120/ijca2018917119.

Bille, Philip, et al. "Fingerprints in compressed strings." *Journal of Computer and System Sciences*, vol. 86, 2017, pp. 171–180. doi:https://doi.org/10.1016/j.jcss.2017.01.002.

Boyer, Robert S. and J. Strother Moore. "A Fast String Searching Algorithm." *Commun. ACM*, vol. 20, no. 10, Oct. 1977, pp. 762–772. doi:10.1145/359842.359859.

Galil, Zvi and Joel Seiferas. "Saving Space in Fast String-Matching." *SIAM Journal on Computing*, vol. 9, no. 2, 1980, pp. 417–438. doi:10.1137/0209032. *https://doi.org/10.1137/0209032*.

———. "Time-space-optimal string matching." *Journal of Computer and System Sciences*, vol. 26, no. 3, 1983, pp. 280–294. doi:https://doi.org/10.1016/0022-0000(83)90002-8.

Karp, Richard M. and Michael O. Rabin. "Efficient randomized pattern-matching algorithms." *IBM Journal of Research and Development*, vol. 31, no. 2, 1987, pp. 249–260. doi:10.1147/rd.312.0249.

KiranShivaji, Sonawane and Prabhudeva S. "Plagiarism Detection by using Karp-Rabin and String Matching Algorithm Together." *International Journal of Computer Applications*, vol. 115, Apr. 2015, pp. 37–41. doi:10.5120/20294-2734.

Knuth, Donald E, et al. "Fast Pattern Matching in Strings." *SIAM journal on computing*, vol. 6, no. 2, 1977, pp. 323–350. doi:10.1137/0206024.

Wise, M J. "Neweyes: a system for comparing biological sequences using the running Karp-Rabin Greedy String-Tiling algorithm." *Proceedings. International Conference on Intelligent Systems for Molecular Biology*, vol. 3, 1995, pp. 393–401.