RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

DATA STRUCTURES & ALGORITHMS TERM PAPER

Rabin-Karp Algorithm for Pattern matching to detect plagiarism

Shounak Rangwala

April 5, 2020

Contents

Abstract:
In this paper we will discuss one of the more efficient algorithms for string matching which comes under the broad purview of pattern matching. Rabin-Karp algorithm has an average time complexity of O(m) + O(n+m-1) and a worst-case scenario of O(n*m), but the medium time of operation itself is increased because of the novel rolling hash function used for the comparison operation. We will be analyzing the mathematics behind the hash function, the different parameters that can be tweaked to change the performance of the Rabin-Karp algorithm according to the application and its performance wrt. other prominent string-matching algorithms. Lastly, we will be studying the various real-life applications of this algorithm in plagiarism detection, network intrusion detection etc.

A. Introduction:

1. ***Strings-searching algorithms***:

These types of algorithms are also called as string- matching algorithms. These try to find the positions in a large text/string where a particular target string occurs. We can also find the patterns that occur in a text and then draw analytical conclusions from it as to the number of times it appears in the text. One particular basic assumption made is that the target string to be searched and the large text in which we will find the target have to be made by the same set of elements. For example, both have to be alpha-numeric (A-Z,0-9) if performed on normal texts or DNA alphabet (A, C, G, T) if we are using the algorithm for bioinformatics.
String algorithms are usually divided into 3 categories:

1) Algorithms searching for a single pattern: The most common application of string algorithms. We search for a single target string in the large body of text for all of its occurrences.
2) Algorithms searching for a finite set of patterns: We look for 2 or more finite number of target strings in the large text body for all their occurrences.
3) Algorithms searching for infinite set of patterns: Usually achieved by regular expressions when you do not know the clear structure of the target string to be searched.

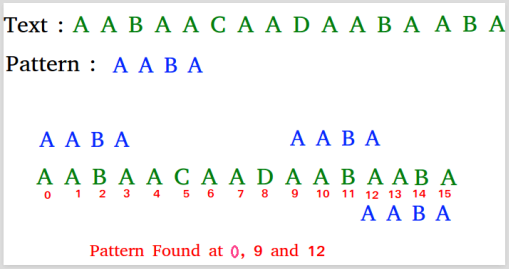
2. ***Strings***:

We should understand what exactly the different ways are to interpret strings before we can apply these algorithms to them. Strings essentially are arrays of characters and these characters can be converted to integers using the ASCII encoding. This will be the general strategy of comparing strings – by comparing them to an array of integers. Naïve or intuitive algorithm that strikes us is comparing each element of the target string to each element in the large text. Assuming the length of the target string is n and the large text is M, the time complexity of this search is usually $O(n*(M-n+1)) \sim O(n*M)$.

As you can see this from the simple example shown below:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```



In order to improve the performance from this naïve implementation to a more linear complexity, O(n), we use algorithms like Rabin-Karp, KMP, Boyer-Moore etc. We will be analyzing Rabin-Karp algorithm because of its unique use of the hash function. Usually the worst case in these algorithms is the same as the naïve implementation, but the preprocessing operations in these algorithms help to modify the string values such that the probability of the worst case occurring are miniscule.

B. Rabin – Karp Algorithm:

This algorithm was created by Richard Karp and Michael Rabin in 1987 for efficient randomized pattern matching. As mentioned before this algorithm uses a "rolling hash function" an improvement on the otherwise simple hashing procedure we described before to improve string matching.
We will now attempt to retrace the footsteps the creators of this algorithm took to go from naïve string comparisons to using hash functions to convert the string to integer to compare more efficiently and finally to using the rolling hash as part of this algorithm.

1. Normal Hash function:

Let the target string be *P* and the large body of text be *S*. We have to search for an occurrence of P in S using hash functions. The following steps will be required to be performed:

1) Calculate the hash of P. Let this be h(P). Assuming that the length of P is L, the complexity of computing the hash will be O(L) since the function will have to read through all the characters in the string to compute the collective hash output.

2) Iterate through all the L- length substrings in S, compute their hash outputs. Assuming that the length of the S string is N, the total number of substrings to be calculated will be N-L+1. The total complexity will be O(N-L+1) * O(L). This is approximately O(NL).

3) We compare each of these calculated hash outputs to the hash of the string P. If there is a match, we then begin element-wise comparison of that substring with the string P. This again will take O(L) complexity because the length of the substring is L.

The overall complexity of this process is O(NL). To reduce this, we will use rolling hashes. The reason there is a scope of improvement on this method is because there is a lot of overlap occurring in the hash computation which increases the redundancy and thus the complexity. This can be explained by a small example. Suppose you wanted to calculate the hashes of substrings of size 4 from the string S = "abcdefghijk". The first substring will be "abcd" and the hash computed will be say, h1. The second substring will be "bcde" and the hash will be computed again for this from scratch and will have the value of h2. The fact that the hash component for the "bcd" part of the string was computed twice shows the overlap that exists in this method.

2. Rolling Hash Function:

I would like to preface this with an illustration of the rolling hash in practice. We will use integer arrays for simplicity.
Imagine P = [1,2,3,4] and S= [9,1,2,3,4,5,6,7,8,9].
If we have to search for P in S, we do the following:

1) Identify all the substrings in S that have the same length as P (in this case len(P)=4). These will be [9,1,2,3], [1,2,3,4], [2,3,4,5] etc.

2) The hash function is as follows:

   $$h(k) = (k[0] * 10^3 + k[1] * 10^2 + k[2] * 10^1 + k[3] * 10^0) \bmod m$$

   Basically, we are taking each substring/subarray and concatenating the integers to make a bigger integer. The base is taken to be 10 because in this case the elements happen to be integers, but the base can differ for characters or for special purpose applications like bioinformatics where the domain size is 4 (A, C, G, T).
   Hence h(P) = 1234 mod m. The first substring of S has a hash $h(S_0)$ = 9123 mod m the second substring has a hash of $h(S_1)$ = 1234 mod m and so on.

3) The main feature is getting from $h(S_0)$ to $h(S_1)$ in constant time. For this if we observe, we need to remove the first element in substring S_0 and add the next element from string S to make S_1.
   So, in this case, the transformation will be as follows:
   S_0= 9123
   S' = 9123-9000 = 123 */*we remove the first element i.e. 9*/*
   S'' = 1230 */*we need to move forward by adding the new element to the back of the original string. Thus, we must increase the base multiplier by 1 for each element and add the new element at the *100 position*/*
   S_1 = S'' + 4 = 1234 */*adding the new element to the modified string*/*

4) This makes the operation of calculating the hash of the next substring to be of order O(1). This is because all we have to do is, remove one element and add an element. Earlier we had to iterate through all the elements in the new substring to calculate the hash making the order O(n). using already calculated results helps in increasing efficiency.

The more generic expression for the hash function is using a base b and len(P) = L:

$$h(k) = (k[0] * b^{L-1} + k[1] * b^{L-2} + \ldots\ldots + k[L-1] * b^{0}) \bmod m$$

And the expression for the next hash is as follows:

$$h(S_{i+1}) = b\,(h(S_i) - L^{l-1} * S[i]) + S[i+L] \bmod m$$

Coming back to string matching, we can incorporate the rolling hash in our algorithm as follows:

1) Calculate the hash of P. Let this be h(P). Assuming that the length of P is L, the complexity of computing the hash will be O(L) since the function will have to read through all the characters in the string to compute the collective hash output.

2) We calculate the hash of the first substring of S. This can take O(L) time. We implement the rolling hash for all the subsequent substring of S which may be in the order O(N). Since each calculation takes O(1) time, this entire comparison operation will take O(N*1) = O(N) complexity.

3) We compare each of these calculated hash outputs to the hash of the string P. If there is a match, we then begin element-wise comparison of that substring with the string P. This again will take O(L) complexity because the length of the substring is L.

This makes the overall complexity O(N) in the average case instead of the previous O(N*L). The one issue we may encounter is that when all the substrings are giving the same hash value which will make us run iteratively on each element of these substrings and this can cause the complexity to increase to O(NL).
We must ensure that the times we enter the step 4 where we compare the target string to the original string should be in the order of O(1). This can be made easier if we use a hash table to record the values of the hash and the corresponding substring.

***3. String signatures*:**
Sometimes to further reduce the chances of the problem degenerating into a O(NL) complexity problem, we can introduce a special hash function which records the substring signature in $O(N^2)$ space so that when we compare the hash values we also compare this secondary hash feature. Iff it matches then we will proceed with the string comparison. Further research can be done into devising the perfect or most effective hash signature function that can perform better.

PS: The Rabin-Karp algorithm is slower than other string searching algorithms like Boyer-Moore, KMP etc. when searching for a single pattern in the text body. However, the efficiency of this algorithm is seen when it compares multiple patterns at once to be present in the text body. To implement this, we can compare the new hash calculated of each substring to the collection of pattern hashes that will be stored in a set-like data structure. This is not possible to implement in the other faster algorithms.

C. Implementation:

We will be implementing the algorithm in a small Python code. As mentioned above we will have to select a base b and a modulo parameter m. According to research, it is best to try to

select the value of *m* such that the multiplication of *b* and *m* is largest possible to fit into the register, whose length will be the same as the big string over which we will be implementing the algorithm. Here *b* is the domain set of the elements that will be constituting the 2 strings over which this algorithm is going to be implemented.

```
# main string = B, target string  = A
#  implementing a code for searching the DNA sequencing thus the domain set will
# be only comprising of A,C,G,T.
A = "TACG"
B = "AATACCGATACGAACGTACGTT"
lengthTarget = len(A)
lenText = len(B)
m = 1073741789
b = 4
Hcommon = b**(lengthTarget-1) % m
result=[]
# convert each element into a integer
def referenceTable (x):
    dictionary = {'A':0,'C':1,"G":2,"T":3}
    return dictionary[x]
# assert the string comparison process
def equal (a):
    for i in range(0,lengthTarget):
        if B[a+i] != A[i]:
            return False
    return True
# calculate the hash for a string
def hashCalc(s):
    h=0
    for i in range(0,len(s)):
        h = (h * b + referenceTable(s[i])) % m
    return h

targetHash = hashCalc(A)
# Rabin-Karp Algo
def RKA():
    substringHash = hashCalc(B[:lengthTarget])
    if substringHash == targetHash:
        if equal(0):
            result.append(0)
    for i in range (0,lenText-lengthTarget):
        substringHash = (b* (substringHash-
referenceTable(B[i])* Hcommon)+ referenceTable(B[i+lengthTarget]))% m
        if substringHash==targetHash:
            if equal(i+1):
                result.append(i+1)
```

```
RKA()
for i in range(len(result)):
# result has the index where the target string can be found
    print(result[i])
```

The result as expected is as follows:

```
shounakrangwala@nbp-208-136 ~
8
16
```

D. Applications of Rabin-Karp algorithm:

There are many research and industrial applications of this string searching/ matching algorithm. Some of the more prominent ones are:

1) Plagiarism detection: The grading of essays can be done using Rabin-Karp by running through key words/phrases through the text and noting the similarity. Here some additional terminologies would be:
    a. K-Gram: this is the way of converting the text into effective sized words by removing the spaces from the text to create a giant string and then dividing the string into smaller strings each of size K. K is also the size of the target string. This is an important step in data preprocessing.
    b. Similarity coefficient: This is done using Dices similarity coefficient equation.

2) Intrusion detection and prevention systems (IDPs): Communication networks which have to test their security and vulnerability stores the signatures of different known attacks in a set data structure. Each time a network attack is detected, the pattern matching algorithm must quickly get the signature of the attack and compare it with the known signatures. This comparison procedure is most crucial because the detection must happen before the attack get over so that corresponding safety precaution measures can be implemented by the system right away. Many researchers have worked on using the faster string-searching algorithms. This is understandable because while Rabin-Karp is an efficient string-matching algorithm, it is a better performer when the size of the patterns to be compared is large.

3) Bioinformatics: As shown in the example above, detecting the same sequencing of DNA can be important in finding different strains of viruses and thus finding an effective vaccine for that. Also, can help in genetic engineering through which better crops can be grown, which are more durable and of better quality.

D. Conclusion and future of Rabin-Karp algorithm:

To summarize, this is not the most efficient algorithm in comparing strings/patterns, but this is more effective than the naïve method. Also, this is the only algorithm that can run pattern matching for multiple patterns in the same body of the text. Researchers are working on finding the effect of changing the lengthTarget and the modulus operand, and how the performance of the algorithm can be changed. Another important development is by using the multiprocessor power in modern-day computers to run different instances of Rabin-Karp on the same body of text but in parallel. This is found to decrease the runtime of the algorithm by a factor of 4-8. Some researchers are working on improving the performance of string-matching algorithms by combining different aspects of each algorithm to make a hybrid algorithm which should be better than both. Some progress has been made in this direction by combining the KMP and the Rabin-Karp algorithms.

E. References:

1) C. Chang and H. Wang, "Comparison of Two-Dimensional String-Matching Algorithms," 2012 International Conference on Computer Science and Electronics Engineering, Hangzhou, 2012, pp. 608-611.
2) V. Gupta, M. Singh and V. K. Bhalla, "Pattern matching algorithms for intrusion detection and prevention system: A comparative analysis," 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), New Delhi, 2014, pp. 50-54.
3) P. Brođanac, L. Budin and D. Jakobović, "Parallelized Rabin-Karp method for exact string matching," Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces, Dubrovnik, 2011, pp. 585-590.
4) Y. Jeong, M. Lee, D. Nam, J. Kim and S. Hwang, "High Performance Parallelization of Boyer-Moore Algorithm on Many-Core Accelerators," 2014 International Conference on Cloud and Autonomic Computing, London, 2014, pp. 265-272.
5) M. M. Musthofa and A. Yaqin, "Implementation of Rabin Karp Algorithm for Essay Writing Test System on Organization xyz," 2019 International Conference on Information and Communications Technology (ICOIACT), Yogyakarta, Indonesia, 2019, pp. 502-507.
6) A. D. Hartanto, A. Syaputra and Y. Pristyanto, "Best Parameter Selection Of Rabin-Karp Algorithm In Detecting Document Similarity," 2019 International Conference on Information and Communications Technology (ICOIACT), Yogyakarta, Indonesia, 2019, pp. 457-461.
7) https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
8) https://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf
9) https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
10) https://people.csail.mit.edu/alinush/6.006-spring-2014/rec06-rabin-karp-spring2011.pdf
11) https://en.wikipedia.org/wiki/String-searching_algorithm