VNUHCM – University of Science

Knowledge Engineering Department

Subject: DATA STRUCTURES AND ALGORITHMS

PROJECT REPORT:

KABIN-KARP AND KNUTT-MORRIS-PRATT ALGORITHM IN STRING MATCHING

Team information – Group 1:

1. Lê Trọng Anh Tú ID: 20127091
2. Đoàn Ánh Dương ID: 20127474
3. Phan Tuấn Khải ID: 20127524
4. Nguyễn Thái Hiệp ID: 20127496

Instructors: Nguyễn Thanh Phương

Bùi Huy Thông

Nguyễn Ngọc Thảo

TP.HCM, Aug 2nd 2021

TABLE OF CONTENTS

Contents

INTRODUCTION

In recent years, there are many processing algorithms, the contribution of which has significantly influenced computer science and string processing algorithm is one of those. Being an important class of string algorithm, string-matching (sometimes called string searching) finds all the occurrences of a given substring in a pattern or a text.

With popularities and many practical applications of string matching, our group is building two most common string-matching algorithm, which is Rabin-Karp and KMP algorithm. They will be descripted step by step and how it works.

1. Rabin-Karp Algorithm

- Basic problem: Given a text and a small pattern, you are required to write a function that shows all occurrence of pattern in text (assume the length of text is n, the length of pattern is m and $n > m$).
- Example: $text[..] = \{abcdabdabcadb\}$,

$$pattern[..] = \{ab\}$$

The answer now should be "The pattern found at index 0, 4, 7".

- General idea: Rabin Karp uses one data structure named "**Hash**". This algorithm calculates the hash value of the pattern and all the hash values of all the substrings of the text.

Since we need to efficiently calculate hash values, we must have a hash function to do this mission. Because hash values can be very large so we mod them with $10^9 + 7$.

Afterwards, we slide the pattern over text one by one, check the hash values of current window of text and pattern. Then calculate hash value for the next window of the text with following step "Remove leading digit, add trailing digit". Because we might get negative value of hash value, we convert it into positive number.

- Trace the algorithm by using a simple example:
- Example: Given a text and a pattern:

$text[..] = \{ababadcadb\}$,

$pattern[..] = \{aba\}$

+ Step 1: We declare and compute all following variables:

. base = 311 as an integer,

. MOD = $10^9 + 7$ as an long long integer,

. patLen = 3 which is equal to the length of a pattern $\{aba\}$,

. strLen = 10 which is equal to the length of a text $\{ababadcadb\}$,

. hashP = 0 (as the hash value of a pattern), hashS = 0 (as the hash value of a string), POW = 1 (uses for taking off the most significant digit in hash value).

+ Step 2: We calculate the value of POW in the first loop of this algorithm which is equal to $pow(base(311), patLen(3) - 1)$ in theory. Because this value can be very large so we MOD with $10^9 + 7$.

. $i = 0 < patLen(3) - 1 = 2, POW = (POW * base)\ \%\ MOD =$

$(1 * 311)\ \%\ (10^9 + 7) = 311.$

. $i = 1 < 2, POW = (POW * base)\ \%\ MOD = (311 * 311)\ \%\ (10^9 + 7) = 96721.$

. $i = 2$, the loop stops right here.

So after the first loop in this algorithm, we calculated the value of POW which is equal to 96721.

+ Step 3: Next, we calculate the hash value of pattern and first window of text:

i	$HashP$	$HashS$	$Running\ code$
0	0	0	$hashP = (hashP * base + pat[i])\ \%\ MOD$ $= (0 * 311 + 97('a'))\ \%\ (109 + 7) = 97$ $hashS = (hashS * base + str[i])\ \%\ MOD$ $= (0 * 311 + 97('a'))\ \%\ (109 + 7) = 97$
1	97	97	$hashP = (hashP * base + pat[i])\ \%\ MOD$ $= (97 * 311 + 98('b'))\ \%\ (109 + 7) = 30265$ $hashS = (hashS * base + str[i])\ \%\ MOD$ $= (97 * 311 + 98('b'))\ \%\ (109 + 7) = 30265$
2	30265	30265	$hashP = (hashP * base + pat[i])\ \%\ MOD$ $= (30265 * 311 + 97('a'))\ \%\ (109 + 7)$ $= 9412512$ $hashS = (hashS * base + str[i])\ \%\ MOD$ $= (30265 * 311 + 97('a'))\ \%\ (109 + 7)$ $= 9412512$
3	9412512	9412512	The loop stops here

+ Step 4: In the last loop of Rabin-Karp algorithm, we check the hash value of current window of text and pattern. If the hash values match, then print i and calculate the hash value for next window of text.

i	$HashP$	$HashS$	Explaination
0	9412512	9412512	$hashP = hashS$, print index 0 $hashS\ afte = ((hashS - str[0] * POW) * base + str[0 + 3])\ \%\ MOD$ $= ((9412512 - 97 * 96721) * 311 + 98)\ \%\ (109 + 7) = 9508923 > 0$
1	9412512	9508923	$hashP \neq hashS$, don't do anything $hashS\ afte = ((hashS - str[1] * POW) * base + str[1 + 3])\ \%\ MOD$ $= ((9508923 - 98 * 96721) * 311 + 97)\ \%\ (109 + 7) = 9412512 > 0$
2	9412512	9412512	$hashP = hashS$, print index 2 $hashS\ afte = ((hashS - str[2] * POW) * base + str[2 + 3])\ \%\ MOD$ $= ((9412512 - 97 * 96721) * 311 + 100)\ \%\ (109 + 7) = 9508925 > 0$

3	9412512	9508925	$hashP \neq hashS$, don't do anything $hashS\ afte = ((hashS - str[3] * POW) * base + str[3 + 3]) \% MOD$ $= ((9508925 - 98 * 96721) * 311 + 99) \% (109 + 7) = 9413136 > 0$
4	9412512	9413136	$hashP \neq hashS$, don't do anything $hashS\ afte = ((hashS - str[4] * POW) * base + str[4 + 3]) \% MOD$ $= ((9413136 - 97 * 96721) * 311 + 97) \% (109 + 7) = 9702986 > 0$
5	9412512	9702986	$hashP \neq hashS$, don't do anything $hashS\ afte = ((hashS - str[5] * POW) * base + str[5 + 3]) \% MOD$ $= ((9702986 - 100 * 96721) * 311 + 100) \% (109 + 7) = 9605646 > 0$
6	9412512	9605646	$hashP \neq hashS$, don't do anything $hashS\ afte = ((hashS - str[6] * POW) * base + str[6 + 3]) \% MOD$ $= ((9605646 - 99 * 96721) * 311 + 98) \% (109 + 7) = 9413135 > 0$
7	9412512	9413135	$hashP \neq hashS$, don't do anything $i = 7$ so don't compute hashS again
8	9412512	9413135	$i = 8 > 7$, the loop stops here

And after this loop, the output will be "The pattern found at index 0, 2".

- Analyze the algorithm's time complexity:

+ First let's assume that n is the length of $text$ and m is the length of $pattern$.

+ In our source code, the complexity in best-case, average-case and also the worst case of Rabin-Karp's algorithm is $O(n + m)$ because:

. In the first loop we calculate the value of POW, so it takes $(m - 1)$ steps.

. We need to calculate the hash value of pattern and the hash value of the first substring of text in the second loop of our source code, so it takes m steps.

. In the last loop, we compare between $hashP$ and $hashS$, calculate the hash value for the next window of the $text$ and check if the hash value is negative, so it takes $3 * (n - m + 1)$ steps.

→ The time complexity is $O(n + m)$.

- Proof the exactness of Rabin Karp Algorithm [9]:
  - Example Q approximately 10^5, and T = 100 test.
    .With 2 different strings, the probability that it has the same Hash is approximately $1/10^9$. Thus, the probability of correctly answering a query is: $1 - 1/10^9$.

.In the worst case, we have Q queries where each query is a different pair of strings. The probability that we answer all the queries correctly is: $(1- 1/10^9)^Q$.

.The probability that we will correctly answer all the queries of all the tests is: $(1- 1/10^9)^{Q*T}$.

- Substituting numbers in, the probability of correctly answering all queries is 0.99.

2. Knutt – Morris – Pratt Algorithm

- Idea [5]:
  - The KMP algorithm uses degenerating property of the pattern (pattern having same sub-pattern appearing more than once in the pattern) to change the worst case complexity to $O(n)$.
  - Whenever we detect a mis match (after some matches), we already know some of the character in the text of the next window, take the advantage of this information to avoid matching the characters that we know will anyway match.
- To walk through this algorithm, we need obviously to know some definitions and build relating functions.

a) Computing Prefix Function [8]:

- **Border** of string S is a prefix of S which is equal to a suffix of S, but not equal to entire S.
  - Ex: "ab" is a border of string "$abcdefab$", "ab" is not a border of "ab".
- **Prefix Function of a string S** (with π table) is a function *s(i)* that for each i return the length of the longest border of the prefix S into table π.
  - Ex: Assume that we have a string $P =$ "$ababadracehuababadra$" with the length 20.

a	b	a	b	a	d	r	a	c	e	h	u	a	b	a	b	a	d	r	a

With first "a", we have no border, so s(0) return 0.

π array →

0	0	1	2	3	0	0	1	0	0	0	0	1	2	3	4	5	6	7	8

Next to $i = 1$, the s(i) has no border neither, so s(1) returns 0.

With $i = 2$, s(i) has a border "a" with the length 1. So s(2) returns 1.

+ $i = 3$, we have border of "ab" with the length 2. So s(3) returns 2.

+ $i = 4$, we have border "aba" with length of 3. So s(4) returns 3. and so on…

To compute the prefix table π, we are following some principles below.

- Relying on the prefix function, we have two lemmas and corollaries:

  First:

  - Lemma: $P[0..i]$ has a border of length $s(i+1) - 1$

    Assume that P has the longest border w of $P[0..i+1]$ at the position i + 1. Let's remove the last character of w, then we have a new border w' at the position i.

    Take the example above, $P =$ "$ababadracehuababadra$".

a	b	a	b	a	d	r	a	c	e	h	u	a	b	a	b	a	d	r	a
			↑	↑															
			i	i+1															
0	0	1	2	3	…														

  - Corollary:
    - $s(i+1) \leq s(i) + 1$.

      As the example, we take the suffix at position $i + 1$ with length $s(i+1) = 3$

and remove the last character from it. We end up with a suffix ending in position I with the length $s(i+1)-1=2$. Now we see $s(i)=2$ and $s(i+1)-1 \leq s(i)$ or $s(i+1) \leq s(i)+1$

- The value of the prefix function cannot increase by more than one from the position to the next position, otherwise it could be stayed the same or decreased some.

Second:

- Lemma: If $s(i) > 0$ then all borders of $P[0..i]$, but for the longest one is also border of $P[0..i-1]$.
- Corollary: All borders of $P[0..i]$ can be enumerated by taking the longest border $b1$ of $P[0..i]$, then the longest border $b2$ of $b1$, and so on.
- Rely on this lemma and corollary, we could compute $s(i+1)$ by following each step [7][8]:
  - If $s[i+1] = s[\pi[i]]$, obviously $\pi[i+1] = \pi[i]+1$ (first lemma).
  - Other case, if $s[i+1] \neq s[\pi[i]]$. We would like to immediately move backward the longest border with length $j < \pi[i]$.
  - Then compare between $s[i+1]$ and $s[j]$. If they are equal, we have $\pi[i+1] = j+1$. Otherwise, we keep track of the smaller border which has value smaller than j, and so on until $j = 0$.
  - If $s[i+1] = s[0]$, let's assign $\pi[i+1] = 1$, and $\pi[i+1] = 0$ in remaining case.

- Computing Prefix Function Pseudo Code: Explained base on two lemmas above.

```
1.  ComputePrefixFunction(P):
2.  s ← array of integers with length(P)
3.  s[0] ← 0
4.  j ← 0
5.  for from i = 1 to length(P):
6.      while (j > 0 & s[i]≠ s[j]):
7.          j ← π[j]
8.      if (s[i] = s[j]) :
9.          j ← j + 1
10.     s[i] ← j
11. return s array
```

b) Build Knuth- Morris-Pratt Algorithm[7][8]:

- In this section, we will build **KMP** algorithm by applying Prefix function to find all the occurrence of a substring in a pattern.

Solution 1 [7][8]:

- Pseudocode:

```
1.  KMPSearchOccurrence(P, Sub):
2.  S←Sub+"#"+P
3.  Pi[] ← ComputPrefixFunction(S)
4.  for i from length(Sub)+1 to length(S) - 1:
5.      if(Pi[i] = length(Sub)):
6.          print i -2 length(Sub)
```

- Explain:

- For all value i, $Pi[i] \leq length(Sub)$ because of the special character "#".
- When we find a value that $Pi[i] = length(Sub)$, it means existing a suffix that is equal to prefix Sub. In other words, $Sub = S[0..length(Sub) - 1] = S[i - length(Sub) + 1..i]$, and Sub is also equal to $P[i - 2length(Sub)..i - length(Sub) - 1]$, so that we returns $i - 2length(Sub)$.

Solution 2:

- Pseudocode:

```
1.  KMPSearchOccurence(P, Sub):
2.  Pi[]←ComputePrefixFunction(Sub).
3.  Pi[0] = 0
4.  int j ← 0.
5.  for i from 1 to length(P):
6.      while (j>0 & Sub[j] ≠P[i]):
7.          j ← Pi[j].
8.      if(Sub[j] = P[i]):
9.          j ← j+1
10.     if(j = length(Sub):
11.         print i-j+1
12.         j← Pi[j]
```

- Explain:
  - This solution 2, we might compute prefix function twice.
  - If $Sub[j] = P[i]$, we increase j by one, it seems like we assign $Pi[i + 1] = P[i] + 1$.
  - Otherwise, if $Sub[j] \neq P[i]$, we move backward to find the smaller value $j = Pi[j]$, which is also the smaller length of previous border. If $Sub[j] = P[i]$, we break out the loop with $j = Pi[j]$, otherwise we keep track of each previous charater of Sub and so on until $j = 0$. Then if $P[i] \neq Sub[0]$, we assign $j = 0$.
  - If we find a value $j = length(Sub)$, we return the position, which found Sub in string P.

c) Complexity [8]:

- Both of the solution executed in linear time $\theta(length(Sub) + length(P))$.
- The complexity of ComputePrefixFunction(P) runs in linear time $\theta(length(P))$. Because we need to compare between two string runs in $O(1)$ for each iterations.
  - The number of the inner "while" loop was bounded by $\theta(length(P))$ because border is always greater than 0, and for each iterations of the "for" loop, it may increase at most by 1 or decrease at least by 1. The length of the $border < length(P)$ so that it is increased by $\theta(length(P))$ time.
- In KMPSearchOccurence, we use ComputePrefixFunction with $\theta(length(P) + length(Sub))$, and it starts finding occurences in linear time $\theta(length(P) + length(Sub))$.
- In total, both worst case and average case are running with linear time $\theta(length(P) + length(Sub))$.

3. Algorithm analysis:

In this section, we continue to compare the distinguish between **Rabin-Karp** and **KMP** algorithm, and so does the identical. We convent that m is the length of substring and n is the length of the pattern or text.

- Analogy:
  - Both are the string-matching algorithm.
  - Both executed in linear time $O(m + n)$ in average case.
  - Preprocessing time of both is $O(m)$.
- Difference:

Rabin-Karp	**Knutt – Morris – Pratt**
- Technique: Using Hashing function. - Time complexity: Worst case: $O(mn)$. - Space complexity: $O(1)$. - Exactness: approximatly 100%.	- Technique: Using prefix array. - Time complexity: Worst case: $O(m + n)$. - Space complexity: $O(m)$. - Exactness: 100%.

4. References

[1] (n.d.). Retrieved from VNOI: https://vnoi.info/wiki/translate/wcipeg/kmp.md

[2] (2011, December 11). Retrieved from Wcipeg: http://wcipeg.com/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

[3] (2020, December 22). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

[4] (2021, July 22). Retrieved from Geeksforgeeks: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

[5] (2021, March 24). Retrieved from Geeksforgeeks: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[6] Bari, A. (2018, March 31). *Abdul Bari*. Retrieved from Youtube: https://www.youtube.com/watch?app=desktop&v=qQ8vS2btsxI

[7] Eng, E. m. (n.d.). Retrieved from CP Algorithm: https://cp-algorithms.com/string/prefix-function.html

[8] Michael Levin, A. S. (n.d.). Retrieved from Coursera: https://www.coursera.org/lecture/algorithms-on-strings/knuth-morris-pratt-algorithm-73RS1

[9] Hash: A String Matching Algorithm (vnoi.info)