Academic report

Data Structures and Algorithms

Challenge 1: String matching

Student name	**Student ID**	**Role**	**Progress**
Nguyen Minh Uyen	19120154	Brute-Force, Comparison table	100%
Pham Duc Tu	19120043	Rabin-Karp, Coding	100%
Nguyen Dang Tien Thanh	19120036	Knuth-Morris-Pratt, Coding	100%
Le Cong Binh	19120176	Definitions, Report	100%

December 16^{th}, 2020

Contents

1 Introduction to string matching

1.1 Definition

Given an alphabet A, a text T (an array of n characters in A) and a pattern P (another array of $m \leq n$ characters in A), we say that P occurs with shifts in T (or P occurs beginning at position $s+1$ in T) if $0 \leq s \leq (n-m)$ and $T[s+j] = P[j]$ for j from 1 to m. A shift is valid if P occurs with shifts in T and invalid otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Input: text string T and pattern string P
Requirement: Find all occurrences of string P in string T

Example:

$$T = tadadattaetadadadafa$$

$$P = dada$$

Valid shifts are two, twelve and fourteen.

1.2 Applications

1.2.1 Plagiarism detection

The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.

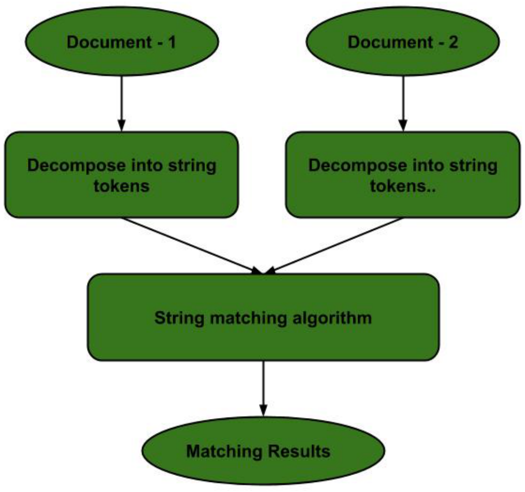

Document - 1
Document - 2
Decompose into string tokens
Decompose into string tokens..
String matching algorithm
Matching Results

Figure 1: Plagiarism detection

1.2.2 Spam filter

Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.

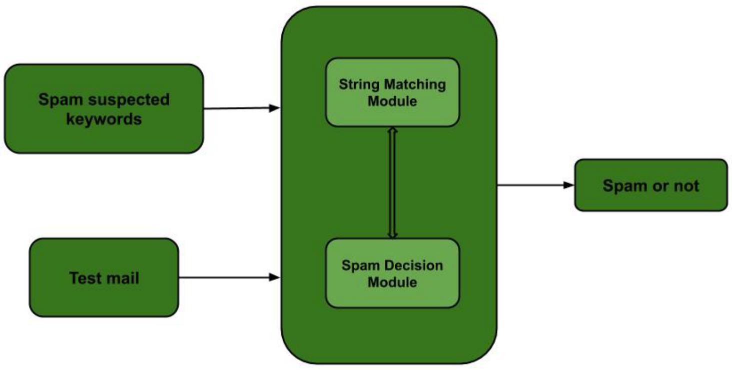


Figure 2: Spam filter

2 String matching algorithms

2.1 Brute-force

2.1.1 Idea

Given a text $T[0...n-1]$ and a pattern $P[0...m-1]$, write a function $search(char\ P[], char\ T[])$ that prints all occurrences of $P[]$ in $T[]$. You may assume that $n > m$. Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

**** Psuedo code:***

```
function Bruteforce(string T, string P):
    m := length of P
    n := length of T

    for i from 0 to (n - m) do:
        j := 0
        while (j < m) do:
            if (T[i + j] != P[j]) then:
                break
            j := j + 1
        if (j == m) then:
            print j

    return not found
```

2.1.2 Step-by-step example

Pattern string P : "$TRING$" with length of 5
Text string S : "$STRINGMATCHING$" with length of 14

Step	Compared substring	Explain
1	"STRIN"	At index 0: "T" and "S". Not a match
2	"TRING"	At index 0: "T" and "T". Match. At index 1: "R" and "R". Match. At index 2: "I" and "I". Match. At index 3: "N" and "N". Match. At index 4: "G" and "G". Match. First match at index 1 in the text string
3	"RINGM"	At index 0: "T" and "R". Not a match
4	"INGMA"	At index 0: "T" and "I". Not a match
5	"NGMAT"	At index 0: "T" and "N". Not a match
6	"GMATC"	At index 0: "T" and "G". Not a match
7	"MATCH"	At index 0: "T" and "M". Not a match
8	"ATCHI"	At index 0: "T" and "A". Not a match
9	"TCHIN"	At index 0: "T" and "T". Match. At index 1: "R" and "C". Not a match
10	"CHING"	At index 0: "T" and "C". Not a match

2.1.3 Time and space complexity

1. **Time complexity:**
   - **Best case:**
     - Happen when the first character of the pattern string doesn't appear in the text string at all. For example, the pattern string is "ABC", the text string is "$DCBHEDB$".
     - Complexity: $O(n)$
   - **Worst case:**
     - Both the pattern string and the text string consist of the same digit. For example, the pattern string is "AAA", the text string is "$AAAAAAAAAAAA$".
     - Complexity: $O(m \times (n - m + 1))$
   - **Average case:**

   - The substrings of the text string are different from the patttern string. Therefore, it does not take much time to stop comparing one substring and move to another.
   - Complexity: $O(m+n)$

   The time complexity of brute-force algorithm is $O(m \times n)$.

2. **Space complexity:** $O(1)$ due to no additional memory needed.

2.2 Rabin-Karp

2.2.1 Idea

The disavantage of the brute force algorithm compares the given pattern against all positions in the given text and cuts short the comparison at each position as soon as a mismatch is found. Therefore, the worst-case time for such a method is proportional to the product of the two lengths. The Rabin–Karp algorithm instead achieves its speedup by using a hash function to quickly perform an approximate check for each position, and then only performing an exact comparison at the positions that pass this approximate check.
In order for the algorithm to work well, the hash function should not produce many false positives, positions of the text which have the same hash value as the pattern but do not actually match the pattern. These positions contribute to the running time of the algorithm unnecessarily, without producing a match.
The good and widely used way to define the hash of a string S of length n is:

$$hash(S) = S_1 \times b^{n-1} + S_2 \times b^{n-2} + ... + S_n \times b^0 \;\; mod \;\; q \tag{1}$$

q: prime modulus

b: the base

For example:

- String "tus", the base is 256 for ASCII code and prime modulus is 101 then the hash value would be:

$$\begin{aligned} Hash(\text{"}tus\text{"}) &= ((116 \times 256 + 117) \times 256 + 115) \;\; mod \;\; 101 \\ &= (116 \times 256^2 + 117 \times 256 + 115) \;\; mod \;\; 101 \\ &= 7632243 \;\; mod \;\; 101 = 77 \end{aligned}$$

Technically, this hash value is an approximate to the true number in a non-decimal system representation. For example we could have the "base" less than one of the "digits" if the text string only contains 3 characters 'a', 'b' and 'c':

- Then hash values are just the value of the text string in ternary numerical system.

$$\text{"}cab\text{"} \longrightarrow 201_3 = 163_{10}$$

**** Psuedo code:***

```
function RabinKarp(string T, string P)
   m := length of P
   n := length of T

   hpattern := hash_function(P)
   for i from 0 to n - m do:
      hs := hash_function(T[i..i+m-1])
      if hs == hpattern then:
         if T[i...i + m - 1] == P[1..m] then:
            return i

   return not found
```

2.2.2 Step-by-step example

In the following example, the base is 256 and prime modulus is 1000000007.
Pattern string P: "TRING" with hash value 157786209
Text string T: "STRINGMATCHING"

Step	**Compared substring**	**Explain**
1	"STRIN"	Substring's hash value is 896961931, which is not equal pattern's hash value
2	"TRING"	Substring' hash value is equal pattern's hash value. Perform an exact comparison and found a matched substring
3	"RINGM"	Substring's hash value is 417182630, which is not equal pattern's hash value
4	"INGMA"	Substring's hash value is 845906091, which is not equal pattern's hash value
5	"NGMAT"	Substring's hash value is 203692068, which is not equal pattern's hash value
6	"GMATC"	Substring's hash value is 238802917, which is not equal pattern's hash value
7	"MATCH"	Substring's hash value is 808520763, which is not equal pattern's hash value
8	"ATCHI"	Substring's hash value is 586567841, which is not equal pattern's hash value

9	"TCHIN"	Substring's hash value is 906061167, which is not equal pattern's hash value
10	"CHING"	Substring's hash value is 975570535, which is not equal pattern's hash value. End algorithm

2.2.3 Time and space complexity

1. **Time complexity:**
    - **Best case:**
        - When there aren't any matched substrings and false positives of the hash function. To compute and compare the substring's hash value with the pattern's hash value in $O(1)$.
        - Complexity: $O(m+n)$
    - **Worst case:**
        - When each substring is a matched substring or a false positive of the hash function, the algorithm performs exact comparisons for each position.
        - Complexity: $O(m \times (n-m+1))$
    - **Average case:**
        - With a good choice of hash function, in normal cases, the pattern substrings seldom appears in the text string.
        - Complexity: $O(m+n)$

    The time complexity of brute-force algorithm is $O(m \times n)$.
    In many applications, we expect few matched substrings:

    - We can expect that the number of false positives is $O(n/q)$ with q is the prime modulus. Then the expected matching time taken by the Rabin-Karp algorithm is: $O(n)+O(m\times(v+n/q))$ with v is number of matched substrings. Hence, if we choose the prime p to be large enough and $v=O(1)$ then the expected complexity is $O(n)$.
2. **Space complexity:** $O(1)$ for rolling hash, a hash function whose value can be quickly updated from each position of the text to the next.

2.2.4 Multiple pattern search

To find any of a large number, say k, fixed length patterns in a text, a simple variant of the Rabin–Karp algorithm uses a Bloom filter or a set data structure to check whether the hash of a given string belongs to a set of hash values of patterns we are looking for.

2.3 Knuth-Morris-Pratt

2.3.1 Idea

The basic ideal behind KMP string matching algorithm is by employing the obveration that after some matches, whenever we detect a mismatch, we already know some information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. We take advantage of this information to avoid matching the characters that we know will anyway match.
Given a string S of length n. The prefix function for this string is defined as an array $\pi[1..n]$, where $\pi[i]$ equal to the length of the longest proper prefix of $S[1..i]$ which is also a suffix of $S[1..i]$. The KMP algorithm starts with computing the array π for the pattern string P. Each pair of character in P and T is compared in a similar fashion to the brute-force approach, during which i and j are increased by 1 simultaneously. If a mismatch ($T[i] \neq P[j]$) occur:

- We have the information: $P[1...j-1] = T[i-j+1...i-1]$
- We also know that $\pi[i-1]$ is the length of both the prefix array and the suffix array P[1...i-1], meaning that $P[1...\pi[j-1]] = P[j-\pi[i-1]...j-1]$
- Based on the above information, we can determine the next matching position, where the suffix $T[1..i]$ and prefix $P[1..j]$ has the longest common substring.
- We have $T[1...i] = T[1...i-1] + T[i]$ and $P[1...j] = P[1...j-1] + P[j]$. Repeat the step of finding j until $T[i] = P[j]$, since $P[1...j-1] = T[i-j+1...i-1]$. So $P[1...j] = T[i-j+1...i]$. Therefore, we find leftmost position of j satisfying the basic idea. If a position for j cannot be found (j=0), stop the loop.
- We find the first occurrence of P in T when $j = |P|$, when $T[i-|P|+1...i] = P$, meaning that P appears at $i-|P|+1$.

**** Psuedo code:***

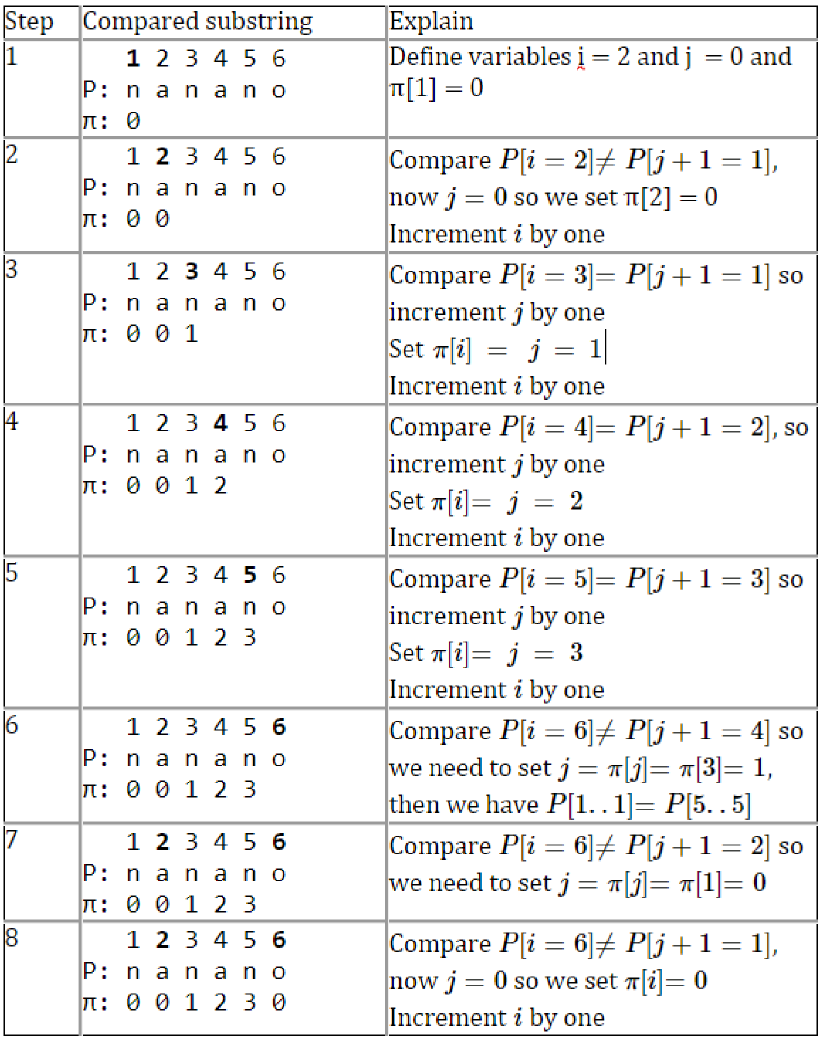
```
function ComputePrefix(string P):
    n[1] := 0
    k := 0
    for i from 2 to m do:
        while k > 0 and S[k+1]   S[i] do:
            k := n[k]
        if S[k+1] == S[i] then:
            k := k + 1
        n[i] := k
    return the string n

function KMP(string T, string P, string ):
    j := 1
    k := 0
    while j + m - 1 <= n do:
        while k <= m and P[k+1] = T[j+k] do:
            k := k + 1
```

```
if k == m then:
    print "Match at position " j
if k == 0 then:
    j := j + 1
else:
    j := j + k - n[k]
    k := n[k]
```

2.3.2 Step-by-step example

Pattern string P : "*nanano*"
Text string S : "*banananano*"

*** Compute the prefix function:***



Step	Compared substring	Explain
1	**1** 2 3 4 5 6 P: n a n a n o π: 0	Define variables i = 2 and j = 0 and π[1] = 0
2	1 **2** 3 4 5 6 P: n a n a n o π: 0 0	Compare $P[i=2] \neq P[j+1=1]$, now $j=0$ so we set π[2] = 0 Increment i by one
3	1 2 **3** 4 5 6 P: n a n a n o π: 0 0 1	Compare $P[i=3] = P[j+1=1]$ so increment j by one Set $\pi[i] = j = 1$ Increment i by one
4	1 2 3 **4** 5 6 P: n a n a n o π: 0 0 1 2	Compare $P[i=4] = P[j+1=2]$, so increment j by one Set $\pi[i] = j = 2$ Increment i by one
5	1 2 3 4 **5** 6 P: n a n a n o π: 0 0 1 2 3	Compare $P[i=5] = P[j+1=3]$ so increment j by one Set $\pi[i] = j = 3$ Increment i by one
6	1 2 3 4 5 **6** P: n a n a n o π: 0 0 1 2 3	Compare $P[i=6] \neq P[j+1=4]$ so we need to set $j = \pi[j] = \pi[3] = 1$, then we have $P[1..1] = P[5..5]$
7	1 **2** 3 4 5 **6** P: n a n a n o π: 0 0 1 2 3	Compare $P[i=6] \neq P[j+1=2]$ so we need to set $j = \pi[j] = \pi[1] = 0$
8	1 **2** 3 4 5 **6** P: n a n a n o π: 0 0 1 2 3 0	Compare $P[i=6] \neq P[j+1=1]$, now $j=0$ so we set $\pi[i] = 0$ Increment i by one

Figure 3: **Compute the prefix function**

The prefix function of pattern string P is $\pi = [0, 0, 1, 2, 3, 0]$.

**** Search pattern:***

Step	Compared substring	Explain
1	1 2 3 4 5 6 7 8 9 10 T: **b** a n a n a n a n o P: **n** a n a n o 1 2 3 4 5 6	Compare $T[i=1] \neq P[j+1=1]$, mismatch occurred, so shift pattern P one position by increment i by 1
2	1 2 3 4 5 6 7 8 9 10 T: b **a** n a n a n a n o P: **n** a n a n o 1 2 3 4 5 6	Compare $T[i=2] \neq P[j+1=1]$, mismatch occurred, so shift pattern P one position by increment i by 1
3	*1 2 **3** 4 5 6 7 8 9 10* T: b a **n** a n a n a n o P: **n** a n a n o *__1__ 2 3 4 5 6*	Compare $T[i=3] = P[j+1=1]$, so increment j by 1
4	1 2 3 **4** 5 6 7 8 9 10 T: b a n **a** n a n a n o P: n **a** n a n o 1 **2** 3 4 5 6	Compare $T[i=4] = P[j+1=2]$, so increment j by 1
5	1 2 3 4 **5** 6 7 8 9 10 T: b a n a **n** a n a n o P: n a **n** a n o 1 2 **3** 4 5 6	Compare $T[i=5] = P[j+1=3]$, so increment j by 1
6	1 2 3 4 5 **6** 7 8 9 10 T: b a n a n **a** n a n o P: n a n **a** n o 1 2 3 **4** 5 6	Compare $T[i=5] = P[j+1=4]$, so increment j by 1
7	1 2 3 4 5 6 **7** 8 9 10 T: b a n a n a **n** a n o P: n a n a **n** o 1 2 3 4 **5** 6	Compare $T[i=6] = P[j+1=5]$, so increment j by 1
8	1 2 3 4 5 6 7 **8** 9 10 T: b a n a n a n **a** n o P: n a n a n **o** 1 2 3 4 5 **6**	Compare $T[i=7] \neq P[j+1=6]$, mismatch occurred, so we need to consider $\pi[j=5] = 3$, set $j=3$, that mean shift the pattern 3 position, and that we have $P[1..3] = T[5..7]$
9	1 2 3 4 5 6 7 **8** 9 10 T: b a n a n a n **a** n o P: n a n **a** n o 1 2 3 **4** 5 6	Compare $T[i=8] = P[j+1=4]$, so increment j by 1.
10	1 2 3 4 5 6 7 8 9 10 T: b a n a n a n a **n** o P: n a n a **n** o 1 2 3 4 5 6	Compare $T[i=8] = P[j+1=4]$, so increment j by 1.
11	1 2 3 4 5 6 7 8 9 **10** T: b a n a n a n a n **o** P: n a n a n **o** 1 2 3 4 5 **6**	Compare $T[i=10] = P[j+1=5]$, so increment j by 1. Now $j = 6 = \|P\|$ so pattern found in text at position $i - \|S\| + 1 = 5$

Figure 4: **Searching pattern**

2.3.3 Time and space complexity

1. **Time complexity:**
   (a) Computing the prefix array for string P: Since this process traverse through each character in the P string, variable j only increase by 1 in each iteration, while the *while loop* decrease j by 1 until it is 0. Because j increase at most m units and decrease at most m units, the algorithm does not perform more than $2 \times m$ comparison operations. So the complexity of this stage is $O(m)$.
   (b) Searching for the pattern: During the traversal through each character of string n, variable j only increase by 1 or less, whilst the *while loop* find the j postion based on the π prefix array previously computed. j will decrease while remaining positive, the number of decrements do not exceed the number of incerments. Overall, j increase at most n times and decrease at most n times, meaning that the *loop* do not iterate more than $2 \times n$ times. Therefore, the time complexity of this step is $O(n)$.
   (c) To sum up, the KMP algorithm has the time complexity of $O(m + n)$.
2. **Space complexity:** O(m) because the KMP algorithm requires additional memory to store the π prefix array.

2.4 Algorithms comparison

Criteria	**Brute-force**	**Rabin-karp**	**KMP**
Best time complexity	$O(n)$	$O(m + n)$	$O(n + m)$
Worst time complexity	$O(m \times (n - m + 1))$	$O(m \times (n - m + 1))$	$O(n + m)$
Average time complexity	$O(m + n)$	$O(m + n)$	$O(n + m)$
Space complexity	$O(1)$	$O(1)$	$O(m)$
Pros	Doesn't require pre-processing of the text	Performs well in multiple pattern search with fix length patterns	Guaranteed worst-case efficient, which is very fast
Cons	Can be ineffective if perform the search more than once	Depend on the performance of the hash function	Can be difficult to implement and debug

2.5 Programming

We can view ***Crossword game*** as a multiple pattern search problem, with different length patterns and each pattern string does not have to be found more than once.

The brute-force algorithm is too slow and inefficient to solve such a problem. Therefore, we decided to choose between the Rabin-Karp and Knuth–Morris–Pratt algorithm. In the end, Knuth–Morris–Pratt algorithm is a better option for our project because of the following reasons:

- The pattern strings do not have fixed lengths, a situation in which the Rabin-Karp algorithm does not perform very well.
- The Rabin-Karp algorithm's performance depends on the performance of hash function. With bad input data, we might suffer from a longer runtime.
- The Knuth–Morris–Pratt algorithm solves the problem stably with low time complexity irrespective of the input data.

3 Reference

- https://www.geeksforgeeks.org/applications-of-string-matching-algorithms
- https://dzone.com/articles/algorithm-week-brute-force
- https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching
- https://en.wikipedia.org/wiki/String-searching_algorithm
- https://en.wikipedia.org/wiki/Rabin-Karp_algorithm
- https://cp-algorithms.com/string/rabin-karp.html
- https://vnoi.info/wiki/translate/wcipeg/kmp.md
- https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching
- https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm