String Matching Algorithms

Topics
- Basics of Strings
- Brute-force String Matcher
- Rabin-Karp String Matching Algorithm
- KMP Algorithm

In string matching problems, it is required to find the occurrences of a pattern in a text.

These problems find applications in text processing, text-editing, computer security, and DNA sequence analysis.

*Find and Change* in word processing

Sequence of the human cyclophilin 40 gene

CCCAGTCTGG AATACAGTGG **CGCGATCTCG** GTTCACTGCA
ACCGCCGCCT **CCCGGG** TTCA AACGATTCTC CTGCCTCAGC

**CGCGATCTCG** : DNA binding protein GATA-1
**CCCGGG** : DNA binding protein Sma 1

C: Cytosine, G: Guanine, A: Adenosine, T: Thymine
Text:  $T[1..n]$ of length $n$ and Pattern $P[1..m]$ of length $m$. The elements of $P$ and $T$ are characters drawn from a finite alphabet set $\Sigma$. For example $\Sigma = \{0,1\}$ or $\Sigma = \{a,b, \ldots, z\}$, or $\Sigma = \{c, g, a, t\}$. The character arrays of $P$ and $T$ are also referred to as strings of characters. Pattern $P$ is said to occur with shift $s$ in text $T$ if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$ or $T[s+j] = P[j]$ for $1 \leq j \leq m$, such a shift is called a valid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern $P$ occurs in a given text $T$.

Brute force string-matching algorithm

To find all valid shifts or possible values of $s$ so that $P[1..m] = T[s+1..s+m]$; There are $n-m+1$ possible values of $s$.

Procedure $BF\_String\_Matcher(T,P)$

1. $n \leftarrow$ length $[T]$;
2. $m \leftarrow$ length$[P]$;
3. for $s \leftarrow 0$ to $n-m$
4. do if $P[1..m] = T[s+1..s+m]$
5. then shift $s$ is valid

This algorithm takes $\Theta((n-m+1)m)$ in the worst case.
Rabin-Karp Algorithm

Let $\Sigma = \{0,1,2,\ldots,9\}$. We can view a string of $k$ consecutive characters as representing a length-$k$ decimal number. Let $p$ denote the decimal number for $P[1..m]$. Let $t_s$ denote the decimal value of the length-$m$ substring $T[s+1..s+m]$ of $T[1..n]$ for $s = 0, 1, \ldots, n-m$.

$t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$, and $s$ is a valid shift.

$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10(P[1])))$ We can compute $p$ in $O(m)$ time.

Similarly we can compute $t_0$ from $T[1..m]$ in $O(m)$ time.


$6378 = 8 + 7 \times 10 + 3 \times 10^2 + 6 \times 10^3 \quad m = 4$

$= 8 + 10 (7 + 10 (3 + 10(6)))$

$= 8 + 70 + 300 + 6000$

$p = \text{P}[m] + 10(\text{P}[m-1] + 10(\text{P}[m-2] + \ldots + 10(\text{P}[2] + 10(\text{P}[1])))$

$t_s+1$ can be computed from $t_s$ in constant time.

$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$

Example: $T = 314152$

$t_s = 31415, s = 0, m = 5$ and $T[s+m+1] = 2$

$t_{s+1} = 10(31415 - 10000*3) + 2 = 14152$

Thus $p$ and $t_0, t_1, \ldots, t_{n-m}$ can all be computed in $O(n+m)$ time.

And all occurrences of the pattern $P[1..m]$ in the text $T[1..n]$ can be found in time $O(n+m)$.

However, $p$ and $t_s$ may be too large to work with conveniently.

Do we have a simple solution!!
Computation of $p$ and $t_0$ and the recurrence is done modulus $q$.

In general, with a d-ary alphabet $\{0,1,\ldots,d-1\}$, $q$ is chosen such that $d \times q$ fits within a computer word.

The recurrence equation can be rewritten as
\[ t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q, \]
where $h = d^{m-1}(\mod q)$ is the value of the digit “1” in the high order position of an m-digit text window.

Note that $t_s \equiv p \mod q$ does not imply that $t_s = p$.
However, if $t_s$ is not equivalent to $p \mod q$, then $t_s \neq p$, and the shift $s$ is invalid.

We use $t_s \equiv p \mod q$ as a fast heuristic test to rule out the invalid shifts.
Further testing is done to eliminate spurious hits.
- an explicit test to check whether
\[ P[1..m] = T[s+1..s+m] \]

Example:

\[ T = 31415; \quad P = 26, n = 5, m = 2, q = 11 \]

\[ p = 26 \mod 11 = 4 \]
\[ t_0 = 31 \mod 11 = 9 \]
\[ t_1 = (10(9 - 3(10) \mod 11) + 4) \mod 11 \]
\[ = (10 (9 - 8) + 4) \mod 14 = 14 \mod 11 = 3 \]
Procedure **RABIN-KARP-MATCHER**(T,P,d,q)  
**Input**: Text T, pattern P, radix d (which is typically = \(|\Sigma|\)), and the prime q.  
**Output**: valid shifts \(s\) where P matches

1. \(n \leftarrow \text{length}[T]\);  
2. \(m \leftarrow \text{length}[P]\);  
3. \(h \leftarrow d^{m-1} \mod q\);  
4. \(p \leftarrow 0\);  
5. \(t_0 \leftarrow 0\);  
6. for \(i \leftarrow 1\) to \(m\)  
7. do \(p \leftarrow (dp + P[i]) \mod q\);  
8. \(t_0 \leftarrow (dt_0 + T[i]) \mod q\);  
9. for \(s \leftarrow 0\) to \(n-m\)  
10. do if \(p = t_s\)  
11. then if \(P[1..m] = T[s+1..s+m]\)  
12. then "pattern occurs with shift " \(s\)  
13. if \(s < n-m\)  
14. then \(t_{s+1} \leftarrow (dt_s - T[s+1]h + T[s+m+1]) \mod q\);  

---

**Comments on Rabin-Karp Algorithm**

- All characters are interpreted as radix-d digits  
- \(h\) is initiated to the value of high order digit position of an \(m\)-digit window  
- \(p\) and \(t_0\) are computed in \(O(m+m)\) time  
- The loop of line 9 takes \(\Theta((n-m+1)m)\) time  

The loop 6-8 takes \(O(m)\) time  
The overall running time is \(O((n-m)+m)\)
Knuth Morris Pratt (KMP) Algorithm

Pseudocode:

KMP-Matcher (T, P)

n ← length (T)
m ← length (P)
π ← Compute-Prefix-Function (P)
q ← 0
for i = 1 to n
    while q > 0 and P[q+1] ≠ T[i];
    do q ← π[q]
    if P[q+1] = T[i]
        then q ← q + 1
    if q = m
        then print "Pattern occurs with shift" (i - m)
    q ← π[q]
Compute-Prefix-Function (P)

Compute Prefix Function (P)
m ← length [P]
π[1] ← 0
k ← 0
for q ← 2 to m
    do while k > 0 and P[k+1] ≠ P[q]
        do k ← π[k]
        if P[k+1] = P[q]
            then k ← k + 1
    π[q] ← k
return π

• Given the pattern P [1..q] matches text chs T[s+1.. S+q]
What is the least shift s’ > s such that
P[1..k] = T[s’+1, .. s’+k], s’+k = s+q
Given pattern P[1..m], the prefix function for the pattern P is the function
π : {1,2, ... m } → {0,1, ...m-1} such that
π [q] = max { k: k < q and P_k is a suffix of P_q}
\[
\begin{array}{cccccccccccc}
  & b & a & c & b & a & b & a & b & a & b & a & c & b & a & b \\
  s & \hline
  & a & b & a & b & a & c & a \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
  & b & a & c & b & a & b & a & b & a & b & a & a & b & c & b & a & b \\
  s' = s + 2 & \hline
  & a & b & a & b & a & c & a \\
\end{array}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[i]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>π[i]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
KMP algorithm (contd..)

Running time analysis of KMP yields O(m+n), because the call of the function takes O(m) time and the remainder KMP matcher algorithm takes O(n) time.

KMP is among the fastest algorithms for large sizes of P and T

Boyer Moore Algorithm

Pseudocode

\[
\begin{align*}
n & \leftarrow \text{length } [T] \\
m & \leftarrow \text{length } [P] \\
\partial & \leftarrow \text{COMPUTE-LAST-OCCURRENCE-FUNCTION}(P, m, \xi) \\
\Phi & \leftarrow \text{COMPUTE-GOOD-SUFFIX-FUNCTION}(P, m) \\
S & \leftarrow 0 \\
\text{While } s \leq n - m & \\
\text{do } j & \leftarrow m \\
\text{while } j > 0 \text{ and } P[j] = T[s+j] & \\
\text{do } j & \leftarrow j - 1 \\
\text{if } j = 0 & \\
\text{then print } "\text{Pattern Occurs at shift } " s & \\
S & \leftarrow s + \Phi[0] \\
\text{else } s & \leftarrow s + \max(\Phi[j], j - \partial[T[s+j]]) \\
S & \leftarrow s + 1 \\
\text{else } s & \leftarrow s + 1
\end{align*}
\]
Boyer Moore Algorithm (contd.)

This algorithm is considered as the most efficient algorithm for most of the general applications of string matching.

♦ This algorithm scans the pattern from right to left
♦ In case of a mismatch it uses 2 pre computed functions
  (a) Good-Suffix Shift (b) Bad Character shift (occurrence shift)

Assume that a mismatch occurs between the character \( x[i] = a \) of the pattern and the character \( y[i+j] = b \) of the text during an attempt at position \( j \). Then, \( x[i+1 .. m-1] = y[i+j+1 .. j+m-1] = u \) and \( x[i] \neq y[i+j] \). The good-suffix shift consists in aligning the segment \( y[i+j+1 .. j+m-1] = x[i+1 .. m-1] \) with its rightmost occurrence in \( x \) that is preceded by a character different from \( x[i] \).

Harish Kumar String matching 19

Boyer Moore Algorithm (contd.)

If there exists no such segment, the shift consists in aligning the longest suffix \( v \) of \( y[i+j+1 .. j+m-1] \) with a matching prefix of \( x \).

The bad-character shift consists in aligning the text character \( y[i+j] \) with its rightmost occurrence in \( x[0 .. m-2] \).

Harish Kumar String matching 20
Boyer Moore Algorithm (contd.)

First attempt

G C A T C G C A G A G T A T A C A G T A C G

Second attempt


Third attempt


Fourth attempt


Fifth attempt


Total number of character comparisons 17