Exact String Matching

TTAAAACTTTGAAAAAATTAACCGCTCTTCGGAGCGGGGGGGGGGGGGGGGGGTTTTTATTATCTAGAATTITAATT
TACGCTCTAAAAATGAACAAGGGATCACTAAAAATTTAAAACAATCAATATTCTTCT
AGCTTTTTATTCCTATTTTAAGATTATAATTAGCGCAACAACTGCTCAATGAAAATCAAA
AATAGGGTTAATATGAATCTCGATCTCCATTTTTGTTCATCGTATTTCAACAACAAGCCAAA
ACTCGTACAAATATGACCCGACTTTCGCTATAAAGAAACACGGCTTGTGGCGAGATATCTCT
TGGAAAAACCTTTCAAGAGCAACTCAATCAACTTTCTCGAGCATTGCATTGCTCACAATATT
GACGTACAAGATAAAAATCGCCATTATTTTGCCCATATAATATGGAACGGTTGGACAATCGGTGAC
ATTGCGACCTTACAATTCGAGCAATACAGTGCTATTCTCAGCAACTACAACAGGGGCAG
CAAGCAGAATTTATCTCTAAATCACGCGGATGAAAAATTCTCCTTCTCGGCGATCAAGAG
CAATACGATCAAACATTGGAATTGCTCATCTATTTGCTCACAATATTACAAAAATTGTAAC
ATGAAATCACCATTCAATTCACAAAGATCTCCTTCTTTCGACTTTGGGAAAGTTTTATT
AAAACAGGTTCACAAAGCCCAACAAAGATGAATCAATACCCACGCTCAAACCAAAAAACATTA
TCGGATTTTATTTACGATTATTATTATAATTACGCGAGACAAACACGGGAGAGCCTAAAGGGTGTCTAG
TTAGATTACGCTAATCTCGCTCAACAAATAGAAACACAGTATCTTTACCTAATATGTGACA
Outline

• Exact Matching Problem
• Naïve Algorithm
• Boyer-Moore Heuristics
• Knuth-Morris-Pratt Algorithm
• Z-algorithm
Books

Chapter 9

Chapter 1-3
Exact Matching Problem

- Given a string \( P \) called \textit{pattern} and a longer string \( T \) called \textit{text}, find all occurrences, if any, of pattern \( P \) in text \( T \).
- A \textit{string} \( S \) is an ordered list of characters.
- A \textit{substring} of \( S \) is a contiguous sequence \( S[i..j] \).
- A \textit{subsequence} of \( S \) may be not consecutive.

TTA\textcolor{red}{AAACTTTTGAA}AAAAT\textcolor{green}{TTAACCGC}
ACTTTTG\textcolor{red}{A}A substring
TTAAA\textcolor{green}{AATCG} subsequence
Example

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 \\
\end{array} \]

\[ T = \text{T A T A T T T A T A T A T A T A} \]

\[ P = \text{T A T A A} \]

Output 0, ?
Example

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 \\
T &=& T & A & T & A & T & T & A & T & A & T & A \\
P &=& T & A & T & A \\
Output &=& 0, 5, 7, 9 \\
\end{array}
\]
Naïve Algorithm (brute force)

- The brute-force pattern matching algorithm compares the pattern $P$ with the text $T$ for each possible shift of $P$ relative to $T$.
- Brute-force pattern matching runs in time $O(nm)$.
- Worst case?

Algorithm $\text{BruteForceMatch}(T, P)$

Input text $T$ of size $n$ and pattern $P$ of size $m$

Output starting indices of substrings of $T$ equal to $P$

for $i \leftarrow 0$ to $n - m$

{ test shift $i$ of the pattern }

$j \leftarrow 0$

while $j < m \land T[i+j] = P[j]$

$j \leftarrow j + 1$

if $j = m$

output $i$ { match at $i$ }
Naïve Algorithm (brute force)

- The brute-force pattern matching algorithm compares the pattern $P$ with the text $T$ for each possible shift of $P$ relative to $T$
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = \text{aaa} \ldots \text{ah}$
  - $P = \text{aaah}$
  - may occur in images and DNA sequences
  - unlikely in English text

Algorithm $\mathbf{BruteForceMatch}(T, P)$

- **Input** text $T$ of size $n$ and pattern $P$ of size $m$
- **Output** starting indices of substrings of $T$ equal to $P$

```
for $i \leftarrow 0$ to $n - m$
{
    test shift $i$ of the pattern 
    $j \leftarrow 0$
    while $j < m \land T[i + j] = P[j]$
    {
        $j \leftarrow j + 1$
    }
    if $j = m$
    {
        output $i$ { match at $i$ }
    }
```

Boyer-Moore Heuristics

- The Boyer-Moore’s pattern matching algorithm is based on two heuristics
  
  **Looking-glass heuristic:** Compare $P$ with a subsequence of $T$ moving backwards

  **Character-jump heuristic:** When a mismatch occurs at $T[i] = c$
  
  - If $P$ contains $c$, shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
  - Else, shift $P$ to align $P[0]$ with $T[i + 1]$

- Example
Last-Occurrence Function

• Boyer-Moore’s algorithm preprocesses the pattern $P$ and the alphabet $S$ to build the last-occurrence function $L$ mapping $S$ to integers, where $L(c)$ is defined as
  • the largest index $i$ such that $P[i] = c$ or
  • -1 if no such index exists

• Example:
  • $S = \{a, b, c, d\}$
  • $P = abacab$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>–1</td>
<td></td>
</tr>
</tbody>
</table>

• The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
• The last-occurrence function can be computed in time $O(m + s)$, where $m$ is the size of $P$ and $s$ is the size of $S$
Boyer-Moore's Algorithm

Algorithm BoyerMooreMatch(T, P, Σ)

\[ L \leftarrow \text{lastOccurenceFunction}(P, \Sigma) \]
\[ i \leftarrow m - 1 \]
\[ j \leftarrow m - 1 \]

repeat
    if \( T[i] = P[j] \)
        if \( j = 0 \)
            output \( i \) \{ match at \( i \) \}
        else
            \( i \leftarrow i - 1 \)
            \( j \leftarrow j - 1 \)
    else
        \{ character-jump \}
        \( l \leftarrow L[T[i]] \)
        \( i \leftarrow i + m - \min(j, 1 + l) \)
        \( j \leftarrow m - 1 \)

until \( i > n - 1 \)

Case 1: \( j \leq 1 + l \)

Case 2: \( 1 + l \leq j \)
Example

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{d} & \text{c} & \text{a} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\end{array}
\]
Analysis

- Boyer-Moore’s algorithm runs in time $O(nm + s)$
- Worst case?

```
   a   a   a   a   a   a   a   a   a   a
   6   5   4   3   2   1
   b   a   a   a   a   a
   12  11  10   9   8   7
   b   a   a   a   a   a
   18  17  16  15  14  13
   b   a   a   a   a   a
   24  23  22  21  20  19
   b   a   a   a   a   a
```
Analysis

- Boyer-Moore’s algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = \text{aaa ... a}$
  - $P = \text{baaa}$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore’s algorithm is significantly faster than the brute-force algorithm on English text
The KMP Algorithm - Motivation

- Knuth-Morris-Pratt’s algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..k]$ that is a suffix of $P[1..k]$.

No need to repeat these comparisons
Resume comparing here
KMP Failure Function

- Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

- The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$.

- Knuth-Morris-Pratt’s algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$, we set $j \leftarrow F(j - 1)$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[j]$</td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
</tr>
<tr>
<td>$F(j)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- Running time?

```
Algorithm KMPMatch(T, P)
    \[ F \leftarrow \text{failureFunction}(P) \]
    \[ i \leftarrow 0 \]
    \[ j \leftarrow 0 \]
    \[ \text{while } i < n \]
        \[ \text{if } T[i] = P[j] \]
            \[ i \leftarrow i + 1 \]
            \[ \text{if } j = m - 1 \]
                \[ \text{output } i - j \{ \text{ match } \} \]
                \[ j \leftarrow F[j] \]
            \[ \text{else} \]
                \[ j \leftarrow j + 1 \]
        \[ \text{else} \]
            \[ j \leftarrow F[j - 1] \]
        \[ \text{else} \]
            \[ i \leftarrow i + 1 \]
```
The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop.
- Thus, KMP’s algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            i ← i + 1
            if j = m - 1
                output i - j { match }
                j ← F[j - 1]
            else
                j ← j + 1
        else
            if j > 0
                j ← F[j - 1]
            else
                i ← i + 1
```
Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time.
- The construction is similar to the KMP algorithm itself.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop.

Algorithm `failureFunction(P)`

\[
\begin{align*}
    F[0] & \leftarrow 0 \\
    i & \leftarrow 1 \\
    j & \leftarrow 0 \\
    \text{while } i < m \\
    \quad \text{if } P[i] = P[j] \\
    \quad \quad \{ \text{we have matched } j + 1 \text{ chars} \} \\
    \quad \quad F[i] & \leftarrow j + 1 \\
    \quad i & \leftarrow i + 1 \\
    \quad j & \leftarrow j + 1 \\
    \quad \text{else if } j > 0 \text{ then} \\
    \quad \quad \{ \text{use failure function to shift } P \} \\
    \quad \quad j & \leftarrow F[j - 1] \\
    \quad \text{else} \\
    \quad F[i] & \leftarrow 0 \{ \text{ no match } \} \\
    \quad i & \leftarrow i + 1
\end{align*}
\]
Example

```
Example

1 2 3 4 5 6
a b a c a b
7
a b a c a b
8 9 10 11 12
a b a c a b
13
a b a c a b

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>F(j)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Aug 26, 2015
Z-algorithm

• Dan Gusfield’s Z-values
• KMP-algorithm has two subroutines similar to each other
• Z-algorithm combines two runs
• KMP is older and more popular
• Warning: strings in the Gusfiled’s book start with 1.
**Z-values of a string**

- **Z(i), i > 1** of a string S is the largest integer d such that $S[1...d] = S[i...i+d-1]$

```
S   1    i    i+d-1
```

- **Example**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| i | = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 |

$P = T\ A\ T\ A\ T\ T\ A\ T\ A\ T\ A\ T\ A\ T$  
$Z = -\ 0\ 3\ 0\ 1\ 5\ 0\ 4\ 0\ 2\ 0$
Question

• How do we find all occurrences of $P$ in $T$ using $Z$ values (of what)?
Exact String Matching with Z values

- Idea: use $PT$ (or $P\$T$ as book suggests)

Algorithm $Z$-matching($P$, $T$)

**Input** text $T$ of size $n$ and pattern $P$ of size $m$

**Output** starting indices of substrings of $T$ equal to $P$

- compute $Z$ values of $PT$
- for $i = 1$ to $n$
  - if $Z[i+m] \geq m$ then
    - output $i$
Time complexity

- \( O(n) \) + time for computing the \( Z \) values of \( PT \).
- We show that \( Z \) values can be found in \( O(n) \) time.
**Z-algorithm**

**Z-box** is the interval \([i \ldots i+Z_i-1]\) if \(Z_i > 0\).

\(r_i\) is rightmost endpoint of **Z-boxes** \(Z_2, Z_3 \ldots Z_{i-1}\).

\(l_i\) is its left endpoint (leftmost endpoint if there many \(r_i\)).

We need to store only one \(r=r_i\) and one \(l=l_i\).

0. Find \(Z_2\) by comparing \(S[2..]\) and \(S[1..]\). Set \(r\) to \(Z_2+1\) and \(l\) to 2.

Suppose \(Z_i\) is computed for all \(i < k\). We find \(Z_k\) as follow.

1. If \(k > r\) then find \(Z_k\) as above (compare \(S[k..]\) and \(S[1..]\)).
   - If \(Z_k > 0\) set \(r\) to \(k+Z_k-1\). Set \(l\) to \(k\).

2. If \(k \leq r\) then \(k'=k-l+1\) and \(\beta=S[k..r]\). Two cases:
   a) If \(Z_{k'} < |\beta|\) then \(Z_k \leftarrow Z_{k'}\) and \(l, r\) unchanged.
   b) If \(Z_{k'} \geq |\beta|\) compare characters starting at positions \(r+1\) and \(|\beta|+1\) until mismatch at \(q>r\) found. Then \(Z_k \leftarrow q-r, r \leftarrow q-1, l \leftarrow k\).
Z-algorithm

Figure 1.3: String $S[k..r]$ is labeled $\beta$ and also occurs starting at position $k'$ of $S$.

Figure 1.4: Case 2a. The longest string starting at $k'$ that matches a prefix of $S$ is shorter than $|\beta|$. In this case, $Z_k = Z_{k'}$.

Figure 1.5: Case 2b. The longest string starting at $k'$ that matches a prefix of $S$ is at least $|\beta|$.
Z-algorithm

Given $Z_i$ for all $1 < i \leq k - 1$ and the current values of $r$ and $l$, $Z_k$ and the updated $r$ and $l$ are computed as follows:

Begin

1. If $k > r$, then find $Z_k$ by explicitly comparing the characters starting at position $k$ to the characters starting at position $l$ of $S$, until a mismatch is found. The length of the match is $Z_k$. If $Z_k > 0$, then set $r$ to $k + Z_k - 1$ and set $l$ to $k$.

2. If $k \leq r$, then position $k$ is contained in a $Z$-box, and hence $S(k)$ is contained in substring $S[l..r]$ (call it $\alpha$) such that $l > 1$ and $\alpha$ matches a prefix of $S$. Therefore, character $S(k)$ also appears in position $k' = k - l + 1$ of $S$. By the same reasoning, substring $S[k..r]$ (call it $\beta$) must match substring $S[k'..Z_i]$. It follows that the substring beginning at position $k$ must match a prefix of $S$ of length at least the minimum of $Z_{k'}$ and $|\beta|$ (which is $r - k + 1$). See Figure 1.3.

We consider two subcases based on the value of that minimum.

2a. If $Z_{k'} < |\beta|$ then $Z_k = Z_{k'}$ and $r$, $l$ remain unchanged (see Figure 1.4).

2b. If $Z_{k'} \geq |\beta|$ then the entire substring $S[k..r]$ must be a prefix of $S$ and $Z_k \geq |\beta| = r - k + 1$. However, $Z_k$ might be strictly larger than $|\beta|$, so compare the characters starting at position $r + 1$ of $S$ to the characters starting a position $|\beta| + 1$ of $S$ until a mismatch occurs. Say the mismatch occurs at character $q \geq r + 1$. Then $Z_k$ is set to $q - k$, $r$ is set to $q - 1$, and $l$ is set to $k$ (see Figure 1.5).

End

Theorem 1.4.1. Using Algorithm $Z$, value $Z_k$ is correctly computed and variables $r$ and $l$ are correctly updated.