String Matching Algorithms

Antonio Carzaniga

Faculty of Informatics
University of Lugano

December 5, 2008

Outline

- Problem definition
- Naïve algorithm
- Knuth-Morris-Pratt algorithm
- Boyer-Moore algorithm
Problem

Given the text

\begin{quote}
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
che la dritta via era smarrita…
\end{quote}

Find the string “trova”

A more challenging example: How many times does the string “110011” appear in the following text

0011110101101001100110111101101011
0110111111010101111011101110000101
10110001011111011100011111000100
100101001011101101010111101001100101
00101100100011111101001101101101011010
0100011011101001010101010000101010011110

String Matching: Definitions

Given a text \( T \)

- \( T \in \Sigma^* \): finite alphabet \( \Sigma \)
- \( |T| = n \): the length of \( T \) is \( n \)

Given a pattern \( P \)

- \( P \in \Sigma^* \): same finite alphabet \( \Sigma \)
- \( |P| = m \): the length of \( P \) is \( m \)

Both \( T \) and \( P \) can be modeled as arrays

- \( T[1 \ldots n] \) and \( P[1 \ldots m] \)

Pattern \( P \) occurs with shift \( s \) in \( T \) iff

- \( 0 \leq s \leq n - m \)
- \( T[s + i] = P[i] \) for all positions \( 1 \leq i \leq m \)
Example

Problem: find all \( s \) such that

- \( 0 \leq s \leq n - m \)
- \( T[s + i] = P[i] \) for \( 1 \leq i \leq m \)

\[
\begin{array}{c}
T & a & b & c & a & a & b & a & a & b & a & a & c & a \\
\end{array}
\]

\( m = 3 \) \( \uparrow \)

\[
\begin{array}{c}
P & a & b & a & a & b & a & a & b & a & a & b & a \\
0 & 4 & 7 & 9 & 3 & 6 & 0 & 3 & 6 & 0 & 3 & 6 & 0 \\
\end{array}
\]

Result

\( s = 4 \)
\( s = 7 \)
\( s = 9 \)

Naïve Algorithm

For each position \( s \) in \( 0 \ldots n - m \), see if \( T[s + i] = P[i] \) for all \( 1 \leq i \leq m \)

```plaintext
Naive-String-Matching(T, P)
1 \( n \leftarrow length(T) \)
2 \( m \leftarrow length(P) \)
3 for \( s \leftarrow 0 \) to \( n - m \)
4 do if Substring-At(T, P, s)
5 then output(s)

Substring-At(T, P, s)
1 for \( i \leftarrow 1 \) to \( length(P) \)
2 do if \( T[s + i] \neq P[i] \)
3 then return false
4 return true
```
Complexity of the Naïve Algorithm

- Complexity of Naive-String-Match is $O((n - m + 1)m)$

- Worst case example

  \[ T = a^n, \quad P = a^m \]

  i.e.,

  \[ T = \overline{aa \cdots a}, \quad P = \overline{aa \cdots a} \]

  So, $(n - m + 1)m$ is a tight bound, so the (worst-case) complexity of Naive-String-Match is

  $\Theta((n - m + 1)m)$

---

Improvement Strategy

- Observation

  \[ T \quad a \mid b \mid c \mid a \mid a \mid b \mid a \mid a \mid b \mid a \mid b \mid a \mid c \mid a \]

  \[ = = \neq \]

  \[ P \quad a \mid b \mid a \]

- What now?

  - the naïve algorithm tells us to go back to the second position in $T$ and to start from the beginning of $P$
  
  - can’t we simply move along through $T$?
  
  - why?
Improvement Strategy (2)

Here's a wrong but insightful strategy

Wrong-String-Matching($T$, $P$)

1. $n \leftarrow \text{length}(T)$
2. $m \leftarrow \text{length}(P)$
3. $q \leftarrow 0$  \text{▷ number of characters matched in $P$}
4. $s \leftarrow 1$
5. \text{while} $s \leq n$
6. \hspace{1em} do $s \leftarrow s + 1$
7. \hspace{2em} if $T[s] = P[q + 1]$
8. \hspace{3em} then $q \leftarrow q + 1$
9. \hspace{3em} if $q = m$
10. \hspace{4em} then output($s - m$)
11. \hspace{3em} $q \leftarrow 0$
12. \hspace{2em} else $q \leftarrow 0$

© 2007 Antonio Carzaniga

Example run of Wrong-String-Matching

<table>
<thead>
<tr>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>p</td>
<td>a</td>
<td>g</td>
<td>i</td>
<td>i</td>
<td>a</td>
<td>i</td>
<td>o</td>
<td>b</td>
<td>a</td>
<td>g</td>
</tr>
</tbody>
</table>

$P$  

\[ P \quad \text{Output: 10} \]

$q + q + q + 1$

Done. Perfect!

Complexity: $\Theta(n)$
Improvement Strategy (4)

- What is wrong with Wrong-String-Matching?

```
  T: a a b a a a b a b a b a c a
  P: a a b

  output(0)  missed!
```

- So Wrong-String-Matching doesn’t work, but it tells us something useful

© 2007 Antonio Carzaniga

Improvement Strategy (5)

- Where did Wrong-String-Matching go wrong?

```
  T: a a b a a a b a b a b a c a
  P: a a b

  q+q+q-1
```

- Wrong: by going all the way back to \( q = 0 \) we throw away a good prefix of \( P \) that we already matched

© 2007 Antonio Carzaniga
Improvement Strategy (6)

- Another example

\[ T \begin{array}{cccccccc}
  a & b & a & b & a & b & a & c \\
\end{array} \]

output (2)

\[ P \begin{array}{cccc}
  a & b & a & b & a & c \\
\end{array} \]

- We have matched “ababa”
  - suffix “aba” can be reused as a prefix

New Strategy

- \( P[1 \ldots q] \) is the prefix of \( P \) matched so far

- Find the longest prefix of \( P \) that is also a suffix of \( P[2 \ldots q] \)
  - i.e., find \( 0 \leq \pi < q \) such that \( P[q - \pi + 1 \ldots q] = P[1 \ldots \pi] \)
  - \( \pi = 0 \) means that such a prefix does not exist

- Restart from \( q \leftarrow \pi \)

- Iterate as usual

- In essence, this is the Knuth-Morris-Pratt algorithm
The Prefix Function

- Given a pattern prefix $P[1 \ldots q]$, the longest prefix of $P$ that is also a suffix of $P[2 \ldots q]$ depends only on $P$ and $q$
- This prefix is identified by its length $\pi(q)$
- Because $\pi(q)$ depends only on $P$ (and $q$), $\pi$ can be computed at the beginning by Prefix-Function
  - we represent $\pi$ as an array of length $m$
- Example

  $P$: a b a b a c
  \[ \pi: 0 \ 0 \ 1 \ 2 \ 3 \ 0 \]

The Knuth-Morris-Pratt Algorithm

```
KMP-String-Matching(T, P)
1   n ← length(T)
2   m ← length(P)
3   \(\pi\) ← Prefix-Function(P)
4   q ← 0               ▷ number of character matched
5   for i ← 1 to n     ▷ scan the text left-to-right
6     do while q > 0 and P[q + 1] ≠ T[i]
7       do q ← \(\pi[q]\)   ▷ no match: go back using $\pi$
8       if P[q + 1] = T[i]
9         then q ← q + 1    
10        if q = m
11          then output(i - m)
12         q ← \(\pi[q]\)   ▷ go back for the next match
```
Prefix Function Algorithm

- Computing the prefix function amounts to finding all the occurrences of a pattern \( P \) in itself.
- In fact, Prefix-Function is remarkably similar to KMP-String-Matching.

Prefix-Function Algorithm

1. \( m \leftarrow \text{length}(P) \)
2. \( \pi[1] \leftarrow 0 \)
3. \( k \leftarrow 0 \)
4. \( \text{for } q \leftarrow 2 \text{ to } m \)
   5. \( \text{do while } k > 0 \text{ and } P[k+1] \neq P[q] \)
   6. \( \text{do } k \leftarrow \pi[k] \)
   7. \( \text{if } P[k+1] = P[q] \)
   8. \( \text{then } k \leftarrow k + 1 \)
   9. \( \pi[q] = k \)

Prefix-Function at Work

Prefix-Function Algorithm

1. \( m \leftarrow \text{length}(P) \)
2. \( \pi[1] \leftarrow 0 \)
3. \( k \leftarrow 0 \)
4. \( \text{for } q \leftarrow 2 \text{ to } m \)
   5. \( \text{do while } k > 0 \text{ and } P[k+1] \neq P[q] \)
   6. \( \text{do } k \leftarrow \pi[k] \)
   7. \( \text{if } P[k+1] = P[q] \)
   8. \( \text{then } k \leftarrow k + 1 \)
   9. \( \pi[q] = k \)

© 2007 Antonio Carzaniga
Complexity of KMP

- $O(n)$ for the search phase
- $O(m)$ for the pre-processing of the pattern
- The complexity analysis is non-trivial
- Can we do better?

Comments on KMP

- Knuth-Morris-Pratt is $\Omega(n)$
  - KMP will always go through at least $n$ character comparisons
  - it fixes our “wrong” algorithm in the case of periodic patterns and texts
- Perhaps there’s another algorithm that works better on the average case
  - e.g., in the absence of periodic patterns
A New Strategy

We match the pattern right-to-left

- If we find a bad character $\alpha$ in the text, we can shift
  - so that the pattern skips $\alpha$, if $\alpha$ is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of $\alpha$
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

In essence, this is the Boyer-Moore algorithm

Comments on Boyer-Moore

- Like KMP, Boyer-Moore includes a pre-processing phase
- The pre-processing is $O(m)$
- The search phase is $O(nm)$
- The search phase can be as low as $O(n/m)$ in common cases
- In practice, Boyer-Moore is the fastest string-matching algorithm for most applications