Efficient Algorithms and Problem Complexity
– Searching in Text –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. Knuth-Morris-Pratt Substring Search

2. Regular Expression Matching
Searching for a String in a Text

The problem

**Input:** A text \( t = t_0 \ldots n \) and a pattern \( p = p_0 \ldots m \), both in \( \Sigma^* \).

**Output:** The least \( i \) such that \( t_{i \ldots i+m} = p \) (and \(-1\) if not such \( i \) exists).

- Naive algorithm: Check for \( i = 0, \ldots, n-m \) whether \( t_{i \ldots i+m} = p \).
- Running time: \( O(mn) \) \( \Rightarrow \) quadratic if the pattern is large).
- Nowadays, texts and patterns are very large in some applications. \( \Rightarrow O(m + n) \) would be desirable.
The Observation by Knuth, Morris, and Pratt

Suppose the initial part of $p = \text{statistics}$ matches $t_{i...i+6}$, but $t_{i+7} \neq i$.

- The naive algorithm increases $i$ by 1 and tries to match $p$ again.
- But we could immediately shift $i$ by 5 positions!
- If the failure occurs after $\text{statistics}$, we can shift by 2 positions.

\[
t = \cdots \quad i \quad \cdots \quad i+s \quad \cdots \quad i+k \quad \cdots
\]
\[
p = \begin{array}{c|c|c|c|c|c}
\hline
0 & \cdots & s & \cdots & k & \cdots \\
\hline
\end{array}
\]
\[
p = \begin{array}{c|c|c|c|c|c}
\hline
0 & \cdots & k-s & \cdots \\
\hline
\end{array}
\]

$\Rightarrow$ if $p$ matched up to index $k \geq -1$ we can safely shift $p$ by

\[
\text{shift}[k] = \min\{s \geq 1 \mid p_{s...k} = p_{0...k-s}\}.
\]
The Knuth-Morris-Pratt Algorithm

\[ KMP-Search(p, t) \] where \( p = p[0, \ldots, m] \) and \( t = t[0, \ldots, n] \)

Compute shift table \( \text{shift}[-1, \ldots, m] \)

\[ i \leftarrow k \leftarrow 0 \]

\[ \text{while } i + m \leq n \text{ do} \]

\[ \text{if } t[i + k] = p[k] \text{ then} \]

\[ k \leftarrow k + 1 \]

\[ \text{if } k > m \text{ then return } i \]

\[ \text{else} \]

\[ i \leftarrow i + \text{shift}[k - 1] \]

\[ k \leftarrow \max(k - \text{shift}[k - 1], 0) \]

\[ \text{return } -1 \]

How efficient is this? And how do we compute the shift table?
Running Time of *KMP-Search*

As we shall see, the shift table can be computed in time $O(m)$. This yields:

**Theorem**

*KMP-Search* returns the position of the first occurrence of $p$ in $t$ (and $-1$ if no such position exists) in time $O(m + n)$.

Note that this is a *uniform* algorithm: not only the text, but even the pattern is part of the input.
Computing the Shift Table

How to compute the shift table?

We have to match (prefixes of) $p$ against $p$

$\Rightarrow$ run $KMP$-$Search(p, p)$ and record the matches in the shift table!
Knuth-Morris-Pratt Substring Search

Computing the Shift Table

\[ KMP-Shift(p) \text{ where } p = p[0, \ldots, m] \]

\[
\begin{align*}
\text{shift}[-1] &\leftarrow \text{shift}[0] \leftarrow 1 \\
i &\leftarrow 1 \\
k &\leftarrow 0 \\
\text{while } i + k \leq m \text{ do} \\
\quad \text{if } p[i + k] = p[k] \text{ then } &\leftarrow \text{found } p[0 \ldots k] = p[i \ldots i + k] \\
\qquad \text{shift}[i + k] &\leftarrow i \\
\qquad k &\leftarrow k + 1 \\
\quad \text{else} \\
\qquad \text{if } k = 0 \text{ then } &\leftarrow \text{no match here at all} \\
\qquad \quad \text{shift}[i] &\leftarrow i + 1 \\
\qquad i &\leftarrow i + \text{shift}[k - 1] \leftarrow \text{is this OK??} \\
\qquad k &\leftarrow \max(k - \text{shift}[k - 1], 0)
\end{align*}
\]
Recalling Regular Expressions

Regular expressions over an alphabet $\Sigma$ consist of

- the letters in $\Sigma$ (we omit $\emptyset$ and $\epsilon$),
- the unary operation $\ast$ ("Kleene star"), and
- the binary operations $|$ and $\cdot$.

A regular expression $E$ denotes a language $L(E) \subseteq \Sigma^*$ defined as follows:

<table>
<thead>
<tr>
<th>Expression $E$</th>
<th>Semantics $L(E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$E^*$</td>
<td>${u_1 \cdots u_n \mid n \geq 0, u_1, \ldots, u_n \in L(E')}$</td>
</tr>
<tr>
<td>$E \mid E'$</td>
<td>$L(E) \cup L(E')$</td>
</tr>
<tr>
<td>$E \cdot E'$</td>
<td>${uu' \mid u \in L(E), u' \in L(E')}$</td>
</tr>
</tbody>
</table>
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot (((a \cdot b) \mid d^*) \cdot c) \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[
  \text{match\_next}(E, t[i], true).
  \]

Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: \( adaabcc \)
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot (((a \cdot b) \mid d^*) \cdot c) \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[ \text{match\_next}(E, t[i], true). \]
  Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: \( adaabcc \)
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot ((a \cdot b) | d^*) \cdot c \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[
  \text{match\_next}(E, t[i], true).
  \]
  Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: adaabcc
Example: $E = a \cdot (((a \cdot b) | d^*) \cdot c)$ as a tree

- Matching $E$ to a text $t$ means to check whether $t = uvw$ with $v \in L(E)$.
- We do this by reading $t$ from left to right.
- Preprocessing: which subtrees match $\epsilon$?
- Match next symbol of $t$:
  $\text{match\_next}(E, t[i], \text{true})$.

  Is $t[i]$ allowed to be at the start of the substring matched by $E$?

Example: $adaabcc$
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot (((a \cdot b) | d^*) \cdot c) \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[
  \text{match\_next}(E, t[i], true).
  \]
  \( \uparrow \)
  Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: \( adaabcc \)
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot (((a \cdot b) \mid d^*) \cdot c) \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[
  \text{match\_next}(E, t[i], true).
  \]
  Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: \textit{adaabcc}
Regular Expression Matching by Repeated Tree Traversal

Example: \( E = a \cdot \(((a \cdot b) \mid d^*) \cdot c) \) as a tree

- Matching \( E \) to a text \( t \) means to check whether \( t = uvw \) with \( v \in L(E) \).
- We do this by reading \( t \) from left to right.
- Preprocessing: which subtrees match \( \epsilon \)?
- Match next symbol of \( t \):
  \[ \text{match\_next}(E, t[i], true) \]
  Is \( t[i] \) allowed to be at the start of the substring matched by \( E \)?

Example: \( adaabcc \)
Regular Expression Matching by Repeated Tree Traversal

\[
\text{match\_next}(E, a, \text{atStart}) \text{ where } E_1, E_2 \text{ are the children of } E \text{ (if present)}
\]

if \(E.\text{type} = \text{letter}\) then
\[
E.\text{match} \leftarrow \text{atStart} \land E.\text{val} = a
\]

else if \(E.\text{type} = |\) then
\[
E.\text{match} \leftarrow \text{match\_next}(E_1, a, \text{atStart}) \lor \text{match\_next}(E_2, a, \text{atStart})
\]

else if \(E.\text{type} = \ast\) then
\[
E.\text{match} \leftarrow \text{match\_next}(E_1, a, \text{atStart} \lor E_1.\text{match})
\]

else if \(E.\text{type} = \cdot\) then
\[
\text{boolean atStart\_Of2nd} = E_1.\text{match} \lor (E_1.\text{epsilon} \land \text{atStart})
\]
\[
\text{match\_next}(E_1, a, \text{atStart})
\]
\[
E.\text{match} \leftarrow \text{match\_next}(E_2, a, \text{atStart\_Of2nd}) \lor (E_1.\text{match} \land E_2.\text{epsilon})
\]

return \(E.\text{match}\)

Note: Strict evaluation of \(\lor\) in case \(E.\text{type} = |\) is important!
Regular Expression Matching by Repeated Tree Traversal

The matching procedure:

\[
\text{match}(E, t) \text{ where } t = T[0, \ldots, n]
\]

\[
\begin{align*}
\text{mark}_{-}\text{epsilon}(E) \\
\text{if } E.\text{epsilon} \text{ then return } \text{true} \\
\text{for } i = 0, \ldots, n \text{ do} \\
\quad \text{if } \text{match}_{-}\text{next}(E, t[i], \text{true}) \text{ then} \\
\quad \quad \text{return } \text{true} \\
\text{return } \text{false}
\end{align*}
\]

Procedure \text{match}_{-}\text{next} traverses \(E\) once

\(\Rightarrow\) running time \(O(mn)\), where \(m\) is the size of \(E\).