Biosequence Algorithms, Spring 2007
Lecture 3: Boyer-Moore Matching

Pekka Kilpeläinen
University of Kuopio
Department of Computer Science
The **Boyer-Moore algorithm** (BM) is a practically efficient method for exact matching. It is especially suitable if

- the alphabet is large (as in natural language)
- the pattern is long (as often in bio-applications)

The speed of BM comes from shifting the pattern $P[1 \ldots n]$ to the right in longer jumps. Typically less than $m$ chars (often about $m/n$ only) of $T[1 \ldots m]$ are examined

BM is based on three main ideas...
Boyer-Moore: Main ideas

Longer shifts result from

1. matching $P$ against $T$ right-to-left, in order $P[n], P[n - 1], \ldots$

2. “bad character shift rule”
   - to avoid repeating unsuccessful comparisons against a mismatched target character

3. “good suffix shift rule”
   - to align only matching pattern characters against target characters already successfully matched

Either rule alone works, but they’re more effective together
Characters of $P$ and $T$ are compared right-to-left:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
1234567890 & 1234567890 & 1234567890
\end{array}
\]

$T$: maistuko kaima maisemaomaloma?
$P$: maisemaomaloma  (legend: match/mismatch)

**Bad character rule:** Align the next-to-left ’i’ of $P$ with the mismatched $T[12] = ’i’$  \[ \sim \]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
1234567890 & 1234567890 & 1234567890
\end{array}
\]

$T$: maistuko kaima maisemaomaloma?
$P$: maisemaomaloma  ^
Bad Character Rule Formally

For each character \( x \in \Sigma \), let

\[
R(x) = \max \{ i < n \mid P[i] = x \} \cup \{0\}
\]

\( R(x) \): right-most occurrence of \( x \) in \( P[1..n - 1] \), or 0

Easy to compute in time \( \Theta(|\Sigma| + |P|) \):

\[
\text{for each } x \in \Sigma \text{ do } R[x] := 0; \\
\text{for } i := 1 \text{ to } n - 1 \text{ do } \\
R[P[i]] := i;
\]
Bad Character Shift

When $P[i] \neq T[h] = x$, shift $P$ to the right by $\max\{1, i - R(x)\}$. This means:

- if the right-most occurrence of $x$ in $P[1 \ldots n - 1]$ is at $j < i$, chars $P[j]$ and $T[h]$ get aligned
- if the right-most occurrence of $x$ in $P[1 \ldots n - 1]$ is at $j > i$, the pattern is shifted to the right by one
- if $x$ doesn’t occur in $P[1 \ldots n - 1]$, shift $= i$, and the pattern is next aligned with $T[h + 1 \ldots h + n]$
(Strong) Good Suffix Rule

Bad character rule is effective, e.g., in searching natural language text (because mismatches are probable)

Horspool’s version of BM applies the bad-char rule only

With a small alphabet, occurrences of $x$ at $P[j]$ for $i < j < n$ are probable ($\rightarrow i - R(x) < 0$; doesn’t help)

Extended bad-char rule helps by recording for each $x \in \Sigma$ and each $i$ the right-most occurrence of $x$ in $P[1..i - 1]$

Additional benefit can be obtained from considering the successfully matched suffix of $P$

We concentrate to so called strong good suffix rule, which is more powerful than the original BM suffix rule
Good Suffix Rule: Illustration

Consider a mismatch at $P[n - 2]$:  

```
  1 2 3
123456789012345678901234567890
```

T: maistuko kaima maisemaomaloma?
P: maisemaomaloma

In an occurrence, $T[12...14] = ima$ must align with “xma”, where $x \neq P[n - 2] = 'o'$  

```
  1 2 3
123456789012345678901234567890
```

T: maistuko kaima maisemaomaloma?
P: maisemaomaloma

`^-^`
Good Suffix Rule Formally

Suppose that $P[i \ldots n]$ has been successfully matched against $T$

**Case 1:** If $P[i - 1]$ is a mismatch and $P$ contains another copy of $P[i \ldots n]$ which is *not preceded by char $P[i - 1]$*, shift $P$ s.t. the closest such copy is aligned with the substring already matched by $P[i \ldots n]$

(See the previous slide for an example)

What if no preceding copy of $P[i \ldots n]$ exists?

$\rightsquigarrow$ **Case 2**
Good Suffix Rule: Case 2

Consider a mismatch at $P[n - 5]$:

$$
\begin{array}{ccc}
1 & 2 & 3 \\
12345678901234567890123456789012 \\
\end{array}
$$

T: mahtava talomaisema omalomaluiun
P: maisemaomaloma

No preceding occurrence of “aloma” in $P$, but a potential occurrence of $P$ begins at $T[13\ldots14] = “ma” \sim$

$$
\begin{array}{ccc}
1 & 2 & 3 \\
12345678901234567890123456789012 \\
\end{array}
$$

T: mahtava talomaisema omalomaluiun
P: maisemaomaloma

^^
Assume that $P[i \ldots n]$ has been successfully matched against substring $t$ of the target

**Case 2:** If Case 1 does not apply, shift $P$ by the smallest amount s.t. a suffix of $t$ matches a prefix of $P$.

**NB 1:** Case 2 applies when an occurrence of $P$ has been found

**NB 2:** As a special case, the longest suffix of $t$ that matches a prefix of $P$ can be empty; then $P$ is shifted by $|P|$ positions
For $i = 2, \ldots, n + 1$, define $L'(i)$ as the largest position of $P$ that satisfies the following:

1. $P[i'..L'(i)] = P[i..n]$ (thus $i' = L'(i) - n + i$) such that $i' = 1$ or $P[i' - 1] \neq P[i - 1]$

if no such copy of suffix $P[i \ldots n]$ occurs in $P$, let $L'(i) = 0$

**NB 1:** If $L'(i) > 0$, then $P[i'..L'(i)]$ is the closest copy of “good suffix” $P[i \ldots n]$, and gives the shift $n - L'(i)$

**NB 2:** Since $P[n + 1 \ldots n] = \epsilon$, $L'(n + 1)$ is the right-most position $j$ s.t. $P[j] \neq P[n]$ (or 0 if all chars are equal).
Example of $L'(i)$

Consider

\[
\begin{array}{c}
1 \\
12345678901234
\end{array}
\]

$P$: maisemaomaloma

Now $L'(15) = 13$
$L'(14) = 0$
$L'(12) = 10$, and
$L'(11) = L'(10) = \cdots = L'(2) = 0$

The $L'$ values can be computed in time $O(n)$; See next
Computing the $L'$ Values (1)

Define $N_j(P)$ to be the length of the longest common suffix of $P[1\ldots j]$ and $P$ ($\Rightarrow 0 \leq N_j(P) \leq j$)

Example:

$$
\begin{array}{c}
\text{1} \\
12345678901234 \\
P: \text{maisemaomaloma}
\end{array}
$$

\[
N_0(P) = N_1(P) = 0, \quad N_2(P) = 2, \\
N_3(P) = \cdots = N_6(P) = 0, \quad N_7(P) = 2, \\
N_8(P) = N_9(P) = 0, \quad N_{10}(P) = 3, \\
N_{11}(P) = \cdots = N_{13}(P) = 0, \\
N_{14}(P) = 14
\]
Computing the $L'$ Values (2)

Remember: $Z_i \sim$ longest repeat of prefix.
Now $N_j$ ($\sim$ longest common suffix) are reverses of $Z_i$:

$$N_j(P) = Z_{n-j+1}(P^r),$$

where $P^r$ is the reverse of $P$

Example:

\[
\begin{align*}
  j &: 123 \ 45678 \quad 123 \ 4567 \ 1 \\
  P &: \text{aamunamua} \quad P^r &: \text{umanumaa}
\end{align*}
\]

$\uparrow$ $\uparrow$

$\sim$ the $N_j$ values can be computed in time $O(|P|)$ by applying the $Z$ algorithm to the reversal of $P$
Computing the $L'$ Values (3)

How do the $N_j$ values help?

**Theorem 2.2.2** If $L'(i) > 0$, then

$$L'(i) = \max\{j < n \mid N_j(P) = |P[i \ldots n]|\}.$$ 

**Proof.** Such $j$ is the right endpoint of the closest copy of $P[i \ldots n]$ which is not preceded by $P[i - 1]$. □

The $L'(i)$ values can be computed in $O(n)$ time by locating the largest $j$ s.t. $N_j(P) = |P[i \ldots n]| = n - i + 1$ ($\Rightarrow$ such $j$ is $L'(i)$ for $i = n - N_j(P) + 1$):

```plaintext
for i := 2 to n + 1 do L'(i) := 0;
for j := 1 to n - 1 do L'(n - N_j(P) + 1) := j;
```
How to compute the smallest shift that aligns a matching prefix of $P$ with a suffix of the successfully matched substring $t$ of $T$ ($= P[i \ldots n]$)?

For $i \geq 2$, let $l(i)$ be the maximum length such that $P[1 \ldots l(i)]$ is equal to a suffix of $P[i \ldots n]$

**Example:** For $P = P[1..5] = “ababa”$, $l(6) = 0$ ($\leftarrow P[6 \ldots 5] = e$), $l(5) = l(4) = 1$ (“a”), and $l(3) = l(2) = 3$ (“aba”)
Now the following theorem holds

**Theorem 2.2.4** \( l(i) = \max\{0 \leq j \leq |P[i\ldots n]| \mid N_j(P) = j\} \)

**Proof.** (Left as an exercise)

This allows us to compute the \( l(i) \) values in time \( O(|P|) \)
(→ Exercise)
**Shifts by the Good Suffix Rule**

When $P[i - 1]$ is a mismatch (after matching $P[i \ldots n]$ successfully)

- (Case 1) if $L'(i) > 0$, shift the pattern to the right by $n - L'(i)$ positions
- (Case 2) if $L'(i) = 0$, shift the pattern to the right by $n - l(i)$ positions

**NB** If already $P[n]$ fails to match, $i = n + 1$, which also gives correct shifts

When an occurrence of $P$ has been found, shift $P$ to the right by $n - l(2)$ positions. Why? To align a prefix of $P$ with the longest matching proper suffix of the occurrence
Which Shift to Use?

Since neither the bad character rule nor the good suffix rule misses any occurrence, we can use the maximum of alternative shift values.

Complete Boyer-Moore Algorithm:

// I: Preprocess P[1...n]:
Compute \( R(x) \) for each \( x \in \Sigma \);
Compute \( L'(i) \) and \( l(i) \) for each \( i = 2, \ldots, n + 1 \);
// II: Search in $T[1 \ldots m]$:
$k := n$;
while $k \leq m$ do
  $i := n$; $h := k$;
  while $i > 0$ and $P[i] = T[h]$ do
    $i := i - 1$; $h := h - 1$;
  endwhile;
  if $i = 0$ then
    Report an occurrence at $T[h + 1 \ldots k]$;
    $k := k + n - l(2)$;
  else // mismatch at $P[i]$
    Increase $k$ by the maximum shift given by the bad character rule and the good suffix rule;
  endif;
endwhile;
Final Remarks

The presented rules avoid performing unnecessary comparisons that would fail.

They can be shown to lead to linear-time behavior, but only if \( P \) does not occur in \( T \). Otherwise the worst-case complexity is still \( \Theta(nm) \).

A simple modification (“Galil rule”; Gusfield, Sect. 3.2.2) leads to a provably linear worst-case time.

On natural language texts the running time is typically sub-linear, and normally BM searches for longer patterns faster.