The Boyer-Moore Algorithm

The string matching algorithm of choice.

Achieves expected sublinear run time using three ideas:
- Right-to-left scan
- Bad character rule
- Good suffix rule

Right-to-left scan

When a mismatch occurs, shift P right by some amount.

If we always shift by one, run time is $O(nm)$. Instead, we try to shift by more than one, if possible.

Definition. For each $x$ in the alphabet, $R(x)$ is the index of the rightmost occurrence of $x$ in $P$. $R(x)$ is 0 if $x$ does not occur in $P$.

Example. $\Sigma = \{a, b, c, d\}$  

\[
\begin{array}{cccc}
 x & a & b & c & d \\
 R(x) & 5 & 6 & 0 & 4 \\
\end{array}
\]

The bad character shift rule

Suppose that for some alignment of $P$ against $T$, the rightmost $n - i$ characters of $P$ are matched, but $P[i]$ mismatches. Assume the mismatch is with $T[k]$. Then, shift $P$ right by $\max[1, i - R(T[k])]$ places.
The extended bad character shift rule

When a mismatch occurs at $P[i]$ and $T[k]$ and $T[k] = x$, shift $P$ to the right so that the closest $x$ in $P$ to the left of $P[i]$ is aligned with $T[k]$.

\[
\begin{array}{cccccccccc}
T & x & & & & & & & & \\
p & x & x & x & y & x & x \\
p & x & x & x & y & x & x \\
\end{array}
\]

The good suffix rule.

Suppose that for some alignment of $P$ and $T$, substring $t$ of $T$ matches a suffix of $P$, but a mismatch occurs at the next position. Find the rightmost copy $t'$ of $t$ in $P$ such that $t'$ is not a suffix of $P$ and the character to the left of $t'$ in $P$ differs from the character to the left of $t$ in $P$. Shift $P$ so that $t'$ in $P$ is aligned with $t$ in $T$.

\[
\begin{array}{cccccccccc}
T & x & t & & & & & & & \\
p & z & t' & y & t \\
p & z & t' & y & t \\
\end{array}
\]

... If there is no such $t'$, shift the left end of $P$ past the left end of $t$ in $T$ by the least amount so that a prefix of the shifted pattern matches a suffix of $t$ in $T$.

\[
\begin{array}{cccccccccc}
T & x & t & & & & & & & \\
p & s & & & & & & & \\
p & s & & & & & & & \\
\end{array}
\]

... If no such shift is possible, shift $P$ by $n$ places to the right (i.e., past $t$).

\[
\begin{array}{cccccccccc}
T & x & t & & & & & & & \\
p & y & t \\
p & y & t \\
\end{array}
\]
If an occurrence of $P$ is found, shift $P$ by the least amount so that a proper prefix of the shifted $P$ matches a suffix of the occurrence of $P$ in $T$. If no such shift is possible, shift $P$ by $n$ places to the right (i.e., past $t$).

**Definition.** For each $i$, $L(i)$ is the largest position less than $n$ such that $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$. $L(i)$ is 0 if no such position exists.

**Definition.** For each $i$, $L'(i)$ is the largest position less than $n$ such that $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$ and such that the character preceding that suffix is not equal to $P[i-1]$. $L'(i)$ is 0 if no such position exists.
Definition. $N_j(P)$ is the length of the longest suffix of $P[1 \ldots j]$ that is also a suffix of the full string $P$.

Example

\begin{align*}
  j & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
  P[j] & \quad c \quad a \quad b \quad d \quad a \quad b \quad d \quad a \quad b \\
  N_j(P) & \quad 0 \quad 0 \quad 2 \quad 0 \quad 0 \quad 5 \quad 0 \quad 0 \quad * \\
\end{align*}

Observation. $N(P)$ is the “reverse” of $Z(P)$. That is, $N_j(P) = Z_{n-j+1}(P')$, where $P'$ is the reversal of $P$.

Example

\begin{align*}
  j & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
  P[j] & \quad c \quad a \quad b \quad d \quad a \quad b \quad d \quad a \quad b \\
  N_j(P) & \quad 0 \quad 0 \quad 2 \quad 0 \quad 0 \quad 5 \quad 0 \quad 0 \quad * \\
  P'[j] & \quad b \quad a \quad d \quad b \quad a \quad d \quad b \quad a \quad c \\
  Z_j(P') & \quad * \quad 0 \quad 0 \quad 5 \quad 0 \quad 0 \quad 2 \quad 0 \quad 0 \\
\end{align*}

Thus, $N(P)$ can be computed in $O(n)$ time.

Theorem.

- $L(i)$ is the largest $j$ such that $N_j(P) \geq |P[i \ldots n]|$.
- $L'(i)$ is the largest $j$ such that $N_j(P) = |P[i \ldots n]|$.

Computing the $L'(i)$’s

\begin{verbatim}
for i ← 1 to n do 
  L'(i) ← 0 
for j ← 1 to n-1 do 
  i ← n - N_j(P) + 1 
  L'(i) ← j 
\end{verbatim}

Thus, $L'$ can be computed in $O(n)$ time.

Definition. $l'(i)$ is the length of the longest suffix of $P[i \ldots n]$ that is also a prefix of $P$. If no such suffix exists, $l'(i) = 0$.

\begin{align*}
  P & \quad c \quad a \quad b \quad d \quad a \quad b \quad d \quad a \quad b \\
  l'(i) & \quad 0 \quad 0 \quad 2 \quad 0 \quad 0 \quad 5 \quad 0 \quad 0 \quad * \\
\end{align*}

Theorem. $l'(i)$ equals the largest $j \leq |P[i \ldots n]|$ such that $N_j(P) = j$.
Using $L'$ and $l'$ for the good suffix rule

◊ If a mismatch occurs at position $i - 1$ of $P$, then
  • If $L'(i) > 0$, shift $P$ right by $n - L'(i)$ places
  • If $L'(i) = 0$, shift $P$ right by $n - l'(i)$ places

◊ If an occurrence of $P$ is found, then shift $P$ by $n - l'(2)$ places.

◊ If $P[n]$ mismatches, shift $P$ one place to the right.

Boyer-Moore($P,T$)
compute $L'(i)$ and $l'(i)$ for each position $i$ of $P$
compute $R(x)$ for each $x$ in the alphabet
$k \leftarrow n$
\while $k \leq m$ \do
  $i \leftarrow n$; $h \leftarrow k$
  \while $i > 0$ \and $P[i] = T[h]$ \do
    $i$--; $h$--
  \if $i = 0$ \then
    report occurrence of $P$ in $T$ ending at $T[k]$
    $k \leftarrow k + n - l'(2)$
  \else
    shift $P$ (increase $k$) by the maximum amount determined by the bad character rule and the good suffix rule

With the strong good suffix rule alone, the worst-case run time of Boyer-Moore is

• $O(m)$ if $P$ is not in $T$ (Knuth, Morris, Pratt 1977, Guibas & Odlyzko 1980, Cole 1994)
• $O(nm)$ if $P$ is in $T$, but can be modified to achieve $O(n+m)$ time in all cases (Galil 1979, Apostolico and Giancarlo 1986)

With the bad character rule alone

• Worst-case time is $O(nm)$
• Expected time on random strings is sublinear
• Sublinear time observed in practice