

Biosequence Algorithms, Spring 2005

Lecture 3: Boyer-Moore Matching

Pekka Kilpeläinen
 University of Kuopio
 Department of Computer Science

Boyer-Moore Algorithm

The **Boyer-Moore algorithm** (BM) is the practical method of choice 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 $P[1 \dots n]$ to the right in longer steps. Typically less than m chars (often about m/n only) of $T[1 \dots m]$ are examined

BM is based on three main ideas:

Boyer-Moore: Main ideas

Longer shifts are based on

- examining P right-to-left, in order $P[n], P[n-1], \dots$
- "bad character shift rule"
 - avoids repeating unsuccessful comparisons against a target character
- "good suffix shift rule"
 - aligns only matching pattern characters against target characters already successfully matched

Either rule alone works, but they're more effective together

Right-to-left Scan and Bad Character Rule

The pattern is examined right-to-left:

```

      1       2       3
123456789012345678901234567890
T: maistuko kaima maisemaomaloma?
P: maisemaomal o ma (Legend: match / mismatch)
      ^
  
```

Bad character rule: Shift the next-to-left occurrence of 'i' below the mismatched 'i' of $T \rightsquigarrow$

```

      1       2       3
123456789012345678901234567890
T: maistuko kaima maisemaomaloma?
P:      maisemaomalom a
      ^
  
```

Bad Character Rule Formally

For any $x \in \Sigma$, let $R(x) = \max(\{0\} \cup \{i < n \mid P[i] = x\})$
 Easy to compute in time $\Theta(|\Sigma| + |P|)$ (Σ is the alphabet)

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 $P[1 \dots n-1]$ is at $j < i$, chars $P[j]$ and $T[h]$ get aligned
- if the right-most occurrence of x in $P[1 \dots n-1]$ is at $j > i$, the pattern is shifted to the right by one
- if x doesn't occur in $P[1 \dots n-1]$, $\text{shift} = i$, and the pattern is next aligned with $T[h+1 \dots h+n]$

(Strong) Good Suffix Rule

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

If the alphabet is small, occurrences of any char close to the end of P are probable. Especially in this case, additional benefit can be obtained from considering the *successfully matched suffix* of P

We concentrate to the so called **strong** good suffix rule, which is more powerful than the (weak) suffix rule of the original Boyer-Moore method

Good Suffix Rule: Illustration

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

```

      1       2       3
123456789012345678901234567890
T: maistuko kaima maisemaomaloma?
P: maisemaomal o ma
  
```

In an occurrence, $T[12 \dots 14] = \text{ima}$ must align with "xma", where x differs from $P[n-2] = 'o' \rightsquigarrow$

```

      1       2       3
123456789012345678901234567890
T: maistuko kaima maisemaomaloma?
P:      maisemaomalom a
      ^
  
```

Good Suffix Rule Formally

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

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

(See the previous slide for an example)

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

\rightsquigarrow Case 2

Good Suffix Rule: Case 2

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

```

      1       2       3
12345678901234567890123456789012
T: mahtava talomaisema omalomailuun
P: maisemaomaloma
    
```

No preceding occurrence of "aloma" in P , but a potential occurrence of P begins at $T[13 \dots 14] = \text{"ma"} \rightsquigarrow$

```

      1       2       3
12345678901234567890123456789012
T: mahtava talomaisema omalomailuun
P:      maisemaomaloma
      ^^
    
```

Case 2 Formally

Assume that $P[i \dots n]$ has been successfully matched against target substring t

Case 2: If Case 1 does not apply, shift P by the least amount possible 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, in which case P is shifted by $|P|$ positions

Preprocessing for the Good Suffix Rule (Case 1)

For $i = 2, \dots, n + 1$, define $L'(i)$ as the largest position of P that satisfies the following:

- 6 $L'(i)$ is the end position of an occurrence of $P[i \dots n]$ that is not preceded by char $P[i - 1]$;

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

NB 1: $0 \leq L'(i) < n$; If $L'(i) > 0$, it is the right endpoint of the closest-to-left copy of "good suffix" $P[i \dots n]$, which gives the shift $n - L'(i)$

NB 2: Since $P[n + 1 \dots n] = \epsilon$, $L'(n + 1)$ 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

```

      1
12345678901234
P: maisemaomaloma
    
```

Now $L'(15) = 13$

$L'(14) = 0$

$L'(13) = 7$ ($\leftarrow P[13..14] = P[6..7] = \text{ma}$, $P[5] \neq P[12]$)

$L'(12) = 10$, and

$L'(11) = L'(10) = \dots = L'(2) = 0$

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 $P[1 \dots j]$ and P ($\Rightarrow 0 \leq N_j(P) \leq j$)

Example: For

```

      1
12345678901234
P: maisemaomaloma
    
```

$N_0(P) = N_1(P) = 0$, $N_2(P) = 2$,
 $N_3(P) = \dots = N_6(P) = 0$, $N_7(P) = 2$,
 $N_8(P) = N_9(P) = 0$, $N_{10}(P) = 3$,
 $N_{11}(P) = \dots = N_{13}(P) = 0$,
 $N_{14}(P) = 14$

Computing the L' Values (2)

Now N_j (\sim longest common suffix) values and Z_i (\sim longest common prefix) values are reverses of each other, i.e.,

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

where P^r is the reverse of P

Example:

```

j: 123 45678      n - j + 1: 876 5432 1
P: aamunamu      P^r      : umanuma
      ↑                ↑
    
```

\rightsquigarrow the N_j values can be computed in time $O(|P|)$ by applying Algorithm Z to the reversal of P

Computing the L' Values (3)

How do the N_j values help?

Theorem 2.2.2 If $L'(i) > 0$, it is the largest $j < n$ for which $N_j(P) = |P[i \dots n]|$ ($= n - i + 1$)

Proof. Such j is the right endpoint of the closest-to-left copy of $P[i \dots n]$ which is not preceded by $P[i - 1]$ \square

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

for $i := 2$ **to** $n + 1$ **do** $L'(i) := 0$;
for $j := 1$ **to** $n - 1$ **do** $L'(n - N_j(P) + 1) := j$;

Preprocessing for Case 2 (1)

How to compute the smallest shift that aligns a matching prefix of P with a suffix of the successfully matched substring of $T = P[i \dots n]$?

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

Example: For $P = P[1..5] = \text{"ababa"}$,
 $l(6) = 0$ ($\leftarrow P[6 \dots 5] = \epsilon$),
 $l(5) = l(4) = 1$ ("a"), and
 $l(3) = l(2) = 3$ ("aba")

Preprocessing for Case 2 (2)

Now the following theorem holds

Theorem 2.2.4 $l(i) = \max\{0 \leq j \leq |P[i \dots n]| \mid N_j(P) = j\}$
Proof. (Left as an exercise) \square

This allows us to compute the $l(i)$ values in time $O(|P|)$
(\rightarrow Exercise)

BSA Lecture 3: BM Algorithm – p.1721

Shifts by the Good Suffix Rule

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

- ⑥ (Case 1) if $L'(i) > 0$, shift the pattern to the right by $n - L'(i)$ positions
- ⑥ (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

BSA Lecture 3: BM Algorithm – p.1821

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:

```
// Preprocessing:  
Compute  $R(x)$  for each  $x \in \Sigma$ ;  
Compute  $L'(i)$  and  $l(i)$  for each  $i = 2, \dots, n + 1$ ;
```

BSA Lecture 3: BM Algorithm – p.1921

BM Search Loop

```
// Search:
```

```
 $k := n$ ;
```

```
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  $T[h + 1 \dots k]$ ;
```

```
     $k := k + n - l(2)$ ;
```

```
  else // mismatch at  $P[i]$ 
```

```
    Increase  $k$  by the maximum shift given by the  
    bad character rule and the good suffix rule;
```

```
  endif;
```

```
endwhile;
```

BSA Lecture 3: BM Algorithm – p.2021

Final Remarks

The presented rules carefully 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) corrects this and leads to a provable worst-case linear time.

On natural language texts the running time is almost always sub-linear

BSA Lecture 3: BM Algorithm – p.2121