



***Biosequence Algorithms, Spring 2005***  
***Lecture 5: The Shift-And Method***

Pekka Kilpeläinen

University of Kuopio

Department of Computer Science

# Seminumerical String Matching

Most matching methods based on **char comparisons**

“Seminumerical” methods that apply **bit-level** and/or **arithmetic operations**:

- ⑥ *FFT*-based  $O(m \log m)$  solution to the *match count problem*:
  - △ For each target position  $j$ , find the number of matching characters in corresponding positions of  $P[1 \dots n]$  and  $T[j - n + 1 \dots j]$
- ⑥ *Karp-Rabin* pattern matching, based on comparing *hash values* of  $P$  and of substrings of  $T$

We discuss **Shift-And**, which is a practical and efficient matching method for short patterns

# Shift-And: Basic idea

Consider locating exact occurrences of pattern  $P[1 \dots n]$  in target  $T[1 \dots m]$ , by going through positions  $j = 1, \dots, m$

**Basic idea:** The state of the search is represented as a **state vector**, which is a **bit vector**  $S[1 \dots n]$

The vector records, simultaneously, occurrences of **any prefix** of  $P$  that end at the current target position  $j$ :

$$S[i] = 1 \text{ iff } P[1 \dots i] = T[j - i + 1 \dots j]$$

(for  $i = 1, \dots, n$ )

# Shift-And State Vector

**Example:** Consider pattern  $P = \text{“ennen”}$ , and the state of search at position 6 of text  $\text{“mennentullen”}$

Now prefixes  $P[1 \dots 2] = \text{“en”}$  and  $P[1 \dots 5] = \text{“ennen”}$  of  $P$  both match a suffix of  $T[1 \dots 6] \rightsquigarrow$  the state vector  $S$  is

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

**NB:** We've found an occurrence of  $P$  iff  $S[n] = 1$

How to compute  $S$  for each  $j = 1, \dots, m$ ?

# Computing the State Vectors

Consider values of  $S$  as *columns*  $M(1), M(2), \dots, M(m)$  of an  $n \times m$  matrix  $M$ :

|     |   | $T$ : |   |   |   |   |   |   |   |   |    |    |    |
|-----|---|-------|---|---|---|---|---|---|---|---|----|----|----|
|     |   | m     | e | n | n | e | n | t | u | l | l  | e  | n  |
| $P$ |   | 1     | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| e   | 1 | 0     | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0  | 1  | 0  |
| n   | 2 | 0     | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0  | 0  | 1  |
| n   | 3 | 0     | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| e   | 4 | 0     | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| n   | 5 | 0     | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0  | 0  | 0  |

Now  $M(j)[i] = 1$  iff  $P[1 \dots i]$  matches a suffix of  $T[1 \dots j]$

$M(j)[1] = 1$  iff  $T[j] = P[1]$ , and  $M(1)[i] = 0$  for  $i > 0$

# Computing the Next Column of $M$

How about  $M(j)[i]$  for  $i, j > 1$ ?

$$\begin{aligned} \text{Now } M(j)[i] = 1 &\Leftrightarrow P[1 \dots i] = T[j - i + 1 \dots j] \\ &\Leftrightarrow P[1 \dots i - 1] = T[j - i + 1 \dots j - 1] \text{ and } P[i] = T[j] \\ &\Leftrightarrow M(j - 1)[i - 1] = 1 \text{ and } P[i] = T[j] \end{aligned}$$

So, we can compute the columns in order  $j = 1, \dots, m$

But this takes clearly time  $\Theta(nm)$  (and character comparisons)!

If  $|P|$  is bounded by a constant, we can compute each column in **constant time**  $\rightsquigarrow \Theta(m)$  time matching algorithm

Is this a practical observation?

# Computing State Vector Fast

(Remember: State vector = the current column of  $M$ )

Assume that  $|P| \leq w$ , where  $w$  is the processor word length (normally 32 or 64). Then we can represent the state vector as a **single number**, and update it really fast

↪ a linear-time matching method **fast in practice**, too

For updating the state vector in a single step, compute for each  $a \in \Sigma$  an  $n$ -bit **occurrence mask**  $U(a)$ , which indicates the occurrences of char  $a$  in  $P$ :

$$U(a)[i] = \begin{cases} 1 & \text{if } P[i] = a \\ 0 & \text{otherwise} \end{cases}$$

# Example of Occurrence Masks

Occurrence masks for pattern “ennen”:

|   | U | a | b | c | d | e | f | ... | m | n | o | ... | z |
|---|---|---|---|---|---|---|---|-----|---|---|---|-----|---|
| e | 1 | 0 | 0 | 0 | 0 | 1 | 0 | ... | 0 | 0 | 0 | ... | 0 |
| n | 2 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 | ... | 0 |
| n | 3 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 | ... | 0 |
| e | 4 | 0 | 0 | 0 | 0 | 1 | 0 | ... | 0 | 0 | 0 | ... | 0 |
| n | 5 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 | ... | 0 |

These can be computed in time  $\Theta(|\Sigma|n)$   
(=  $\Theta(|\Sigma|)$  if  $n$  is bounded by a constant)

# Updating the State Vector

Define an operation which shifts elements of bit-vector  $S$  forward by one, and inserts 1 as the first element:

$$\text{BitShift}(S)[i] = \begin{cases} 1 & \text{for } i = 1 \\ S[i - 1] & \text{for } i = 2, \dots, n \end{cases}$$

This can be implemented in C-like languages as follows:

$$S = (S \ll 1) \mid 1$$

Now **AND**ing  $\text{BitShift}(S)$  with the occurrence mask of the current text character  $T[j]$  updates the state vector correctly, from column  $M(j - 1)$  to  $M(j)$ :

# Correctness of State Vector Updates

**Theorem** If  $S = M(j - 1)$ , then

$$\text{BitShift}(S) \text{ AND } U(T[j]) = M(j).$$

**Proof.** Assume that  $S[i] = M(j - 1)[i]$  for  $i = 1, \dots, n$ , and let  $S' = \text{BitShift}(S) \text{ AND } U(T[j])$ .

$$S'[1] = 1 \Leftrightarrow U(T[j])[1] = 1 \Leftrightarrow T[j] = P[1] \Leftrightarrow M(j)[1] = 1$$

For  $i > 1$ ,

$$\begin{aligned} S'[i] = 1 &\Leftrightarrow S[i - 1] = 1 \text{ and } U(T[j])[i] = 1 \\ &\Leftrightarrow M(j - 1)[i - 1] = 1 \text{ and } T[j] = P[i] \\ &\Leftrightarrow M(j)[i] = 1 \end{aligned}$$

So,  $S'[i] = M(j)[i]$  for each  $i = 1, \dots, n$ . □

# Updating the State Vector: Example

Consider moving from text position 3 to 4, while searching for pattern “ennen” in text “mennentullen” ( $T[4] = n$ ):

| $S = M(3)$ | $\rightsquigarrow$ | BitShift( $S$ ) | <b>AND</b> | $U(n)$ | $=$ | $M(4)$ |
|------------|--------------------|-----------------|------------|--------|-----|--------|
| 0          |                    | 1               |            | 0      |     | 0      |
| 1          |                    | 0               |            | 1      |     | 0      |
| 0          |                    | 1               |            | 1      |     | 1      |
| 0          |                    | 0               |            | 0      |     | 0      |
| 0          |                    | 0               |            | 1      |     | 0      |

# Shift-And Implementation

Sketch of a full algorithm using C-like bit operations:

```
compute_U_masks( $P$ );  
S := 0;  
one_at_row_n := 1 << ( $n-1$ );  
for  $j := 1$  to  $m$  do  
    S := BitShift(S) & U( $T[j]$ );  
    if (S & one_at_row_n) then  
        Report an occurrence at  $j - n + 1$ ;  
    endif;  
endfor;
```

## Remarks

This is sometimes called “*bit(-level) parallelism*”

The original authors (Baeza-Yates & Gonnet, CACM 10/1992) complemented the role of bits, and presented the method as **Shift-Or**. Shift-Or is a bit more efficient, but less intuitive to explain

Experiments on English text indicate *Shift-Or* to be about 2.5 times faster than the naive method, and more efficient than Boyer-Moore with patterns shorter than 4–10 characters (depending on the BM implementation)

**Shift-And** easily **generalizes to matching with wild-cards** either in  $P$  or  $T$  (or both) (Exercise)

# Extension to Inexact Matching

Wu and Manber (CACM 10/1992) extended Shift-And to **inexact matching** or **matching with errors**:

Find all target positions where  $P$  occurs *with at most  $k$  errors* (single-char mismatches, insertions or deletions)

(“mismatch” = “*epävastaavuus*” tai “*yhteensopimattomuus*”)

Now we discuss mismatches only.

(We’ll dwell into inexact matching in general later.)

**Example:** *atcgaa* occurs with 2 mismatches at posn 4 of

|          |          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        | 10       | 11       |
| <i>a</i> | <i>a</i> | <i>t</i> | <i>a</i> | <i>t</i> | <i>c</i> | <i>c</i> | <i>a</i> | <i>c</i> | <i>a</i> | <i>a</i> |

and with 4 mismatches at positions 2 and 6

# Shift-And with Mismatches: Main ideas

Wu-Manber method is efficient for small  $|P|$  ( $\leq w$ ) and  $k$  ( $\leq 4$ ). It is included in the approximate search tool called **agrep**

Instead of a single table  $M$  (as before), compute tables  $M^0, M^1, \dots, M^k$ , each of size  $n \times m$ :

$M^l(j)[i] = 1$  iff  $P[1 \dots i]$  matches  $T[j - i + 1 \dots j]$   
with at most  $l$  mismatches

Obs 1:  $M^0 = M$

Obs 2:  $M^k(j)[n] = 1$  iff  $T[j - n + 1 \dots j]$  is an occurrence of  $P$  with at most  $k$  mismatches

Obs 3: For  $l \geq 1$ ,  $M^l(1)[1] = 1$ , and  $M^l(1)[i] = 0$  when  $i > 1$

# How to Compute $M^k$

Compute column  $j - 1$  of each table before column  $j$ , and for each  $j$  tables in order  $M^0, M^1, \dots, M^k$

- ⑥ column  $M^0(j)$  from  $M^0(j - 1)$  as in Shift-And

When computing column of  $M^l(j)$  for  $l > 0$  and  $j > 1$ ,

- ⑥ column  $M^{l-1}(j - 1)$ ,
- ⑥ column  $M^l(j - 1)$ , and
- ⑥ column  $M^{l-1}(j)$

have been computed

(Instance of *dynamic programming*)

# Computing Table Entries

When is  $M^l(j)[i] = 1$  (for  $l > 0$ )?  $\Leftrightarrow$  When does  $P[1 \dots i]$  match  $T[j - i + 1 \dots j]$  with at most  $l$  mismatches? (\*)

- (1) If  $P[1 \dots i]$  matches  $T[j - i + 1 \dots j]$  with at most  $l - 1$  mismatches ( $\Leftrightarrow M^{l-1}(j)[i] = 1$ )
- (2) If  $P[1 \dots i - 1]$  matches  $T[j - i + 1 \dots j - 1]$  with at most  $l$  mismatches and  $P[i] = T[j]$  ( $\Leftrightarrow M^l(j - 1)[i - 1] = 1$  and  $P[i] = T[j]$ )
- (3) If  $P[1 \dots i - 1]$  matches  $T[j - i + 1 \dots j - 1]$  with at most  $l - 1$  mismatches ( $\Leftrightarrow M^{l-1}(j - 1)[i - 1] = 1$ )

Conversely, (\*) holds only if at least one of (1)–(3) holds

# Computing Columns of $M^l$

An entire column  $M^l(j)$  can now be computed, using bit operations, as follows:

$$M^l(j) \quad := \quad M^{l-1}(j) \quad (1)$$

$$\text{OR} \quad \text{BitShift}(M^{l-1}(j-1)) \text{ AND } U(T[j]) \quad (2)$$

$$\text{OR} \quad \text{BitShift}(M^{l-1}(j-1)) \quad (3)$$

(1)–(3) correspond to the cases of the previous slide

# Complexity

Only two columns ( $j$  and  $j - 1$ ) of each table need to be maintained  $\rightsquigarrow$  **space** complexity is  $\Theta(nk) = \Theta(k \times |P|)$

$k + 1$  tables each of size  $n \times m$  are computed, each table entry in constant time  $\rightsquigarrow$  total **time** is  $\Theta(knm)$

If  $n \leq w$ , each column is computed with a few machine instructions, and time is  $\Theta(km)$  (and fast in practice)