Text Book: Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin 2 nd Edition

Unit-4

4. Input Enhancement in String Matching – Boyer Moore algorithm

String matching involves finding an occurrence of a given string of *m* characters called the ***pattern*** in a longer string of *n* characters called the ***text***.

Boyer Moore algorithm compares pattern characters to text from right to left precomputing shift sizes in **two** tables

1. ***bad-symbol table*** indicates how much to shift based on text's character causing a mismatch
2. ***good-suffix table*** indicates how much to shift based on matched part (suffix) of the pattern (taking advantage of the periodic structure of the pattern)

Bad-symbol shift Table:

1. If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
2. If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character *c* is encountered after *k* > 0 matches

$$\begin{array}{ccccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern}
\end{array}$$

If *c* is not in the pattern, we shift the pattern to just pass this *c* in the text. Conveniently, the size of this shift can be computed by the formula ***$t_1(c) - k$*** where ***$t_1(c)$*** is the entry in the precomputed table used by **Horspool's algorithm** and ***k*** is the **number of matched characters**:

$$\begin{array}{cccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \nparallel & \| & & \| & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \\
 & & & & p_0 & \dots & & p_{m-1} & &
\end{array}$$

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathrm{S}) - 2 = 6 - 2 = 4$ positions:

$$\begin{array}{cccccccccccccc}
s_0 & \dots & & & & \mathrm{S} & \mathrm{E} & \mathrm{R} & & & & & \dots & s_{n-1} \\
 & & & & & \nparallel & \| & \| & & & & & & \\
 & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & & & & & & \\
 & & & & & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & &
\end{array}$$

If **$t_1(c) - k \le 0$**, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.
To summarize, the bad-symbol shift d1 is computed by the Boyer-Moore algorithm either as **$t_1(c) - k$** if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

$d_1 = \max\{t_1(c) - k, 1\}$

Good-suffix shift:

The second type of shift is guided by a successful match of the last k > 0 characters of the pattern.We refer to the ending portion of the pattern as its suffix of size k and denote it suff (k). Accordingly, we call this type of shift the good-suffix shift.

$d_2(k)$ = Distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

k	**pattern**	d_2
1	$\text{ABC}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{AB}}\text{CB}\underline{\text{AB}}$	4
3	$\overline{\text{AB}}\text{C}\underline{\text{BAB}}$	4
4	$\overline{\text{AB}}\underline{\text{CBAB}}$	4
5	$\overline{\text{A}\underline{\text{B}}}\underline{\text{CBAB}}$	4

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

$$\text{where } d_1 = \max\{t_1(c) - k, 1\}$$

<u>Boyer-Moore algorithm</u> :

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

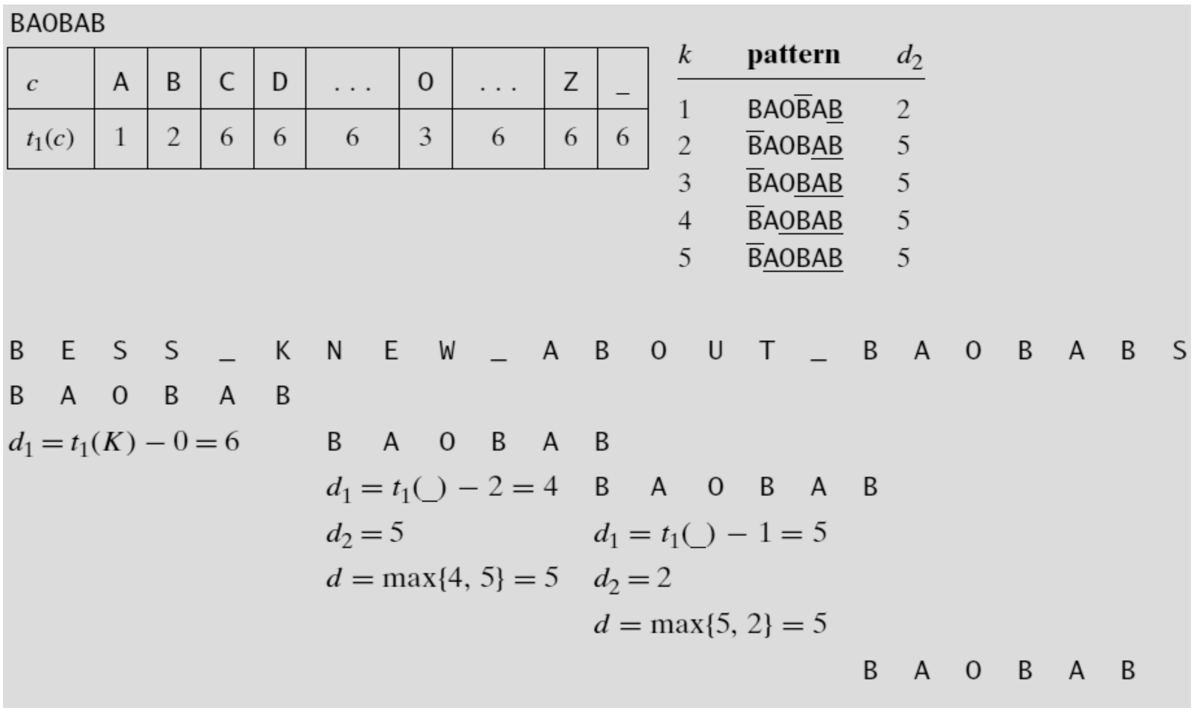
Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

Shifting by the maximum of the two available shifts when $k > 0$ is quite logical. The two shifts are based on the observations—the first one about a text's mismatched character, and the second one about a matched group of the pattern's rightmost characters—that imply that shifting by less than d_1 and d_2 characters, respectively, cannot lead to aligning the pattern with a matching substring in the text. Since we are interested in shifting the pattern as far as possible without missing a possible matching substring, we take the maximum of these two numbers.

Example


BAOBAB

c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

k	**pattern**	d_2
1	$\mathrm{BAO\overline{B}A\underline{B}}$	2
2	$\mathrm{\overline{B}AOB\underline{AB}}$	5
3	$\mathrm{\overline{B}AO\underline{BAB}}$	5
4	$\mathrm{\overline{B}A\underline{OBAB}}$	5
5	$\mathrm{\overline{B}\underline{AOBAB}}$	5

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) - 0 = 6$ B A O B A B

$d_1 = t_1(\_) - 2 = 4$ B A O B A B

$d_2 = 5$ $d_1 = t_1(\_) - 1 = 5$

$d = \max\{4, 5\} = 5$ $d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear.