Boyer-Moore Algorithm

Now we outline the Boyer-Moore algorithm itself. If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool's algorithm. Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed as explained earlier.

The two algorithms act differently, however, after some positive number k $(0 < k < m)$ of the pattern's characters are matched successfully before a mismatch is encountered:

s_0	$\ldots$		c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
			$\not\vert$	$\vert\vert$		$\vert\vert$			
	p_0	$\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern

In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities. The first one is guided by the text's character c that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad-symbol shift***. The reasoning behind this shift is the reasoning we used in Horspool's algorithm. 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 (see above) and k is the number of matched characters:

s_0	$\ldots$		c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
			$\not\vert$	$\vert\vert$		$\vert\vert$			
	p_0	$\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern
				p_0	$\ldots$		p_{m-1}		

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 $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:

s_0	$\ldots$			S	E	R			$\ldots$	s_{n-1}
				$\not\vert$	$\vert\vert$	$\vert\vert$				
		B	A	R	B	E	R			
					B	A	R	B	E	R

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\mathsf{A}) - 2 = 4 - 2 = 2$ positions:

s_0	$\ldots$			A	E	R			$\ldots$	s_{n-1}
				$\not\vert$	$\vert\vert$	$\vert\vert$				
		B	A	R	B	E	R			
				B	A	R	B	E	R	

If $t_1(c) - k \leq 0$, 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 d_1 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:

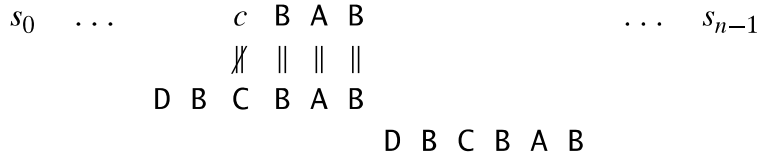
$$d_1 = \max\{t_1(c) - k, 1\}. \quad \textbf{(7.2)}$$

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***. We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, \ldots, m-1$ to fill in the good-suffix shift table.

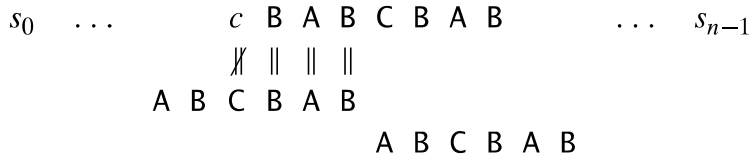
Let us first consider the case when there is another occurrence of $suff(k)$ in the pattern or, to be more accurate, there is another occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence. (It would be useless to shift the pattern to match another occurrence of $suff(k)$ preceded by the same character because this would simply repeat a failed trial.) In this case, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of $suff(k)$ and its rightmost occurrence. For example, for the pattern ABCBAB, these distances for $k = 1$ and 2 will be 2 and 4, respectively:

k	**pattern**	d_2
1	ABCBAB	2
2	ABCBAB	4

What is to be done if there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence? In most cases, we can shift the pattern by its entire length m. For example, for the pattern DBCBAB and $k = 3$, we can shift the pattern by its entire length of 6 characters:



Unfortunately, shifting the pattern by its entire length when there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence is not always correct. For example, for the pattern ABCBAB and $k = 3$, shifting by 6 could miss a matching substring that starts with the text's AB aligned with the last two characters of the pattern:



Note that the shift by 6 is correct for the pattern DBCBAB but not for ABCBAB, because the latter pattern has the same substring AB as its prefix (beginning part of the pattern) and as its suffix (ending part of the pattern). To avoid such an erroneous shift based on a suffix of size k, for which there is no other occurrence in the pattern not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length m. As an example, here is the complete list of the d_2 values—the good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB:

k	**pattern**	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Now we are prepared to summarize the Boyer-Moore algorithm in its entirety.

The Boyer-Moore algorithm

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

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases} \tag{7.3}$$

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

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 As a complete example, let us consider searching for the pattern BAOBAB in a text made of English letters and spaces. The bad-symbol table looks as follows:

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

The good-suffix table is filled as follows:

k	**pattern**	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{BA}}$OB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{BA}}$O$\underline{\text{BAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

The actual search for this pattern in the text given in Figure 7.3 proceeds as follows. After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves $t_1(\text{K}) = 6$ from the bad-symbol table and shifts the pattern by $d_1 = \max\{t_1(\text{K}) - 0, 1\} = 6$ positions to the right. The new try successfully matches two pairs of characters. After the failure of the third comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 5$ from the good-suffix table to shift the pattern by $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$. Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.

The next try successfully matches just one pair of B's. After the failure of the next comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 2$ from the good-suffix table to shift

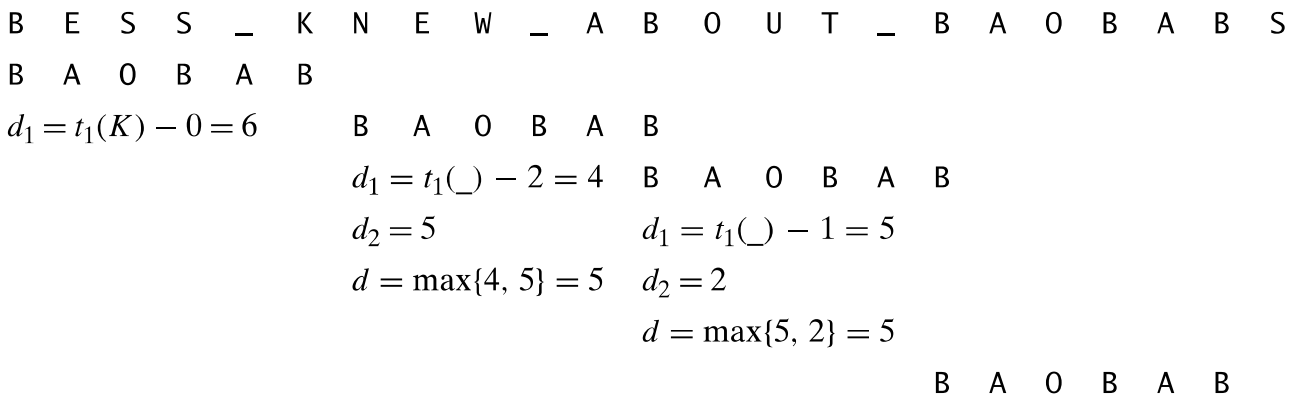


FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm.

the pattern by $\max\{d_1, d_2\} = \max\{6 - 1, 2\} = 5$. Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text. ■

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear. Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern), many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.

Exercises 7.2

1. Apply Horspool's algorithm to search for the pattern BAOBAB in the text

   BESS_KNEW_ABOUT_BAOBABS

2. Consider the problem of searching for genes in DNA sequences using Horspool's algorithm. A DNA sequence is represented by a text on the alphabet {A, C, G, T}, and the gene or gene segment is the pattern.
   a. Construct the shift table for the following gene segment of your chromosome 10:

      TCCTATTCTT

   b. Apply Horspool's algorithm to locate the above pattern in the following DNA sequence:

      TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT