Feb 2025, India

Boyer – Moore Algorithm

Tutorial

Tutorial By

Vikas Awadhiya

Invented by,

Robert S. Boyer

&

J Strother Moore

Tutorial By

Vikas Awadhiya

LinkedIn profile: https://in.linkedin.com/in/awadhiya-vikas

Introduction

Boyer-Moore algorithm is an efficient string searching algorithm. It searches the pattern in the string in linear time complexity in average case scenario. It has quadratic time complexity in worst case scenario but this is rare in practice. In best case scenario Boyer-Moore algorithm performs better than linear time complexity. As compared to Boyer-Moore algorithm, naïve (brute-force) algorithm has quadratic time complexity.

Boyer-Moore algorithm obtains the efficiency by changing the conventional approach of matching the pattern from left to right. The Boyer-Moore algorithm matches the pattern from right to left which means it begins matching from the last character of the pattern and moves backward to the first character.

Algorithm pre-processes the pattern to obtain information about it. This information is gathered in terms of two tables. These tables help to decide how many positions to the right the pattern must slides in case of a character mismatch. In naïve algorithm when a mismatch happens pattern slides to the right by one position and comparisons restart from first character of the pattern but at a mismatch, Boyer-Moore algorithm tries to slide the pattern as far as possible to the right, which can be less than or equals to the length of pattern and after pattern sliding to the right, character matching begins again from the last character of the pattern.

The space complexity is close to linear. It requires two tables, the first table have size equals to the number of letters in the alphabet of a language and the second table has size equals to the length of pattern.

Pattern Searching

The Boyer-Moore algorithm begins searching the pattern in the string by first aligning the left ends of both the string and the pattern and then the character matching begins from the last character of the pattern and moves backward to the first character as the matching succeeds. When a character mismatch happens, the naïve algorithm slides the pattern to the right by one position and restarts the character matching from the first character of the pattern. A character mismatch is just a mismatch for the naïve algorithm but in the same situation the Boyer-Moore algorithm outperforms the naïve algorithm.

The Boyer-Moore algorithm defines three categories or scenarios of a character mismatch. These scenarios help algorithm to slide the pattern more efficiently when a mismatch happens. These scenarios are as follows,

When A Mismatched Character Is Not Present In The Pattern

If the string is **S** = “**abcabdcbabcbcaba**” and the pattern is **Patt** = “**abcbca**” then,

S =	a	b	c	a	b	d	c	b	a	b	c	b	c	a	b	a
Patt =	a	b	c	b	c	a										

Fig 1.0

As highlighted by gray color filled cells above in fig 1.0, the last character of the pattern doesn’t match with respective string character and the mismatched character “d” is not present in the pattern. It means pattern cannot slide to the right by distance less than pattern’s length because no character in pattern can match with character “d”. So, the pattern will slide by six positions beyond the character “d” and character matching begins again from the last character of the pattern as shown below in fig 2.0.

S =	a	b	c	a	b	d	c	b	a	b	c	b	c	a	b	a
Patt =							a	b	c	b	c	a				

Fig 2.0

The scenario presented by last character mismatch but it can occur with a character mismatch at any position in the pattern.

When A Mismatched Character Is Present In The Pattern

This is a specific scenario of mismatch of the last character of the pattern. If we reconsider the previous string and pattern as shown above in fig 2.0. The last character of pattern doesn’t match with respective character of the string but this time the mismatched character “b” is present in the pattern. To avoid the further mismatch, the pattern must be slide to the right in such way that the character “b” of the pattern aligns to the character “b” of the string but here the character “b” occurs twice in the pattern. So, if the pattern has multiple occurrences of a character, then rightmost occurrence of the character must be aligned. In this case the rightmost occurrence of the character “b” is at two positions away from the last character of the pattern. So, the pattern slides to the right by two positions and the character matching begins again from the last character of pattern as shown below in fig 3.0,

S =	a	b	c	a	b	d	c	b	a	b	c	b	c	a	b	a
Patt =									a	b	c	b	c	a		

Fig 3.0

When A Mismatched Character Is Present In The Pattern And The Mismatch Is Not At The Last Character

In this scenario the mismatch character is present in the pattern and mismatch occurs after some successful matches and not at the last character of the pattern.

If the string is **S** = "**abacbbadabcabdcab**" and patter is **Patt** = "**dabcabdcab**" then,

S =	a	b	a	c	b	b	a	d	a	b	c	a	b	d	c	a	b
Patt =	d	a	b	c	a	b	d	c	a	b							

Fig 4.0

As highlighted by gray color filled cells above in fig 4.0, character mismatched after matching of two characters and it is not at the last character of the pattern.

The character "d" is present in the pattern and the distance between the rightmost occurrence of the character "d" and the mismatch position is one. So, the pattern can slide to the right by one position but this scenario may offer more efficient approach than the character alignment.

As show above in the fig 4.0, the remaining pattern after the mismatched character "c" in the pattern can be refer as a sub-pattern and if this sub-pattern = "ab" reoccurs in its left side of the pattern and not preceding by the character "c" then pattern can slide to the right to align the reoccurrence of the sub-pattern.

Patt =	d	a	b	c	a	b	d	c	a	b

Fig 5.0

As show above in the fig 5.0, all the reoccurrences of sub-pattern are highlighted by gray color filled cells. There are two reoccurrences of sub-pattern but one of them is preceded by a character "c" and it cannot be used but the other reoccurrence close to the left end of the patter is preceded by a different character "d". So, the pattern slides to the right by 7 positions which is equal to the distance between reoccurrence of sub-pattern and sub-pattern. Character matching begins again from the last character of the pattern as show below in the fig 6.0.

S =	a	b	a	c	b	b	a	d	a	b	c	a	b	d	c	a	b
Patt =								d	a	b	c	a	b	d	c	a	b

Fig 6.0

Tables Creation

The previous section is a theoretical explanation of pattern pre-processing but the algorithm creates two tables to store the information gathers in pattern pre-processing. The delta$_1$ table provides information of first and second scenarios and delta$_2$ table provides information of third scenario discussed in previous section.

The Boyer-Moore algorithm is available as standard library from C++17 onwards (std::boyer_moore_searcher) and it is not required to implement and that's why the document explains the algorithm with 1-based index.

Delta$_1$ Table

This table contains one entry for each letter of the alphabet and has size equals to the number of letters in the alphabet of a language.

If a letter is not present in the pattern, then its entry has pattern length as a value in detal$_1$ table and if a letter is present, then its entry has a value equal to the difference between pattern length and index of its rightmost occurrence in the pattern. A table with one entry for each letter of the English alphabet cannot be shown here and due to this, entries of the absent letters are not shown explicitly but they have values equal to the pattern length. If the pattern **Patt** = "**abcdbabcab**" and pattern length is 10 then,

Patt =	a	b	c	d	b	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10

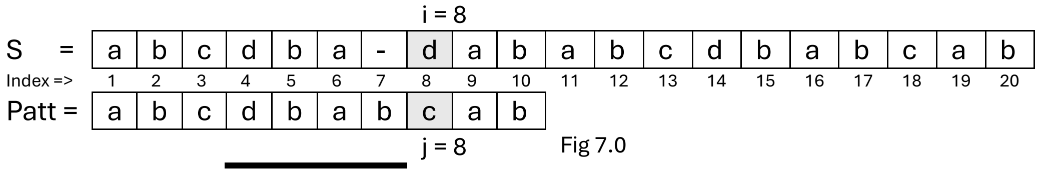
	a	b	c	d
delta$_1$ =	1	0	2	6

The value of each entry of the deta$_1$ table is evaluated as follows,

delta$_1$[character] = (mismatch index – rightmost index of character) + (pattern length – mismatch index)

= pattern length – rightmost character index

A positive and a negative mismatch index variables cancel out each other and as a result formula becomes difference between pattern length and rightmost character index. The first sub-part of the original formula is to align the mismatched character by sliding the pattern to the right by the distance between the mismatch index and rightmost index of character. If we search this pattern in a string and a mismatch happens as shown below in fig 7.0 the first sub-part of the formula as a distance highlighted by bold black underline,



S =	a	b	c	d	b	a	-	d	a	b	a	b	c	d	b	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Patt =	a	b	c	d	b	a	b	c	a	b										

Fig 7.0

After pattern sliding the character matching restart form last character of pattern and as shown above in fig 7.0 characters mismatch happened after two-character matches and to restart the character matching, index i must additionally moves two positions to the right to align with last character of pattern and that is the second sub-part of the formula.

Here the sub-pattern “ab” also reoccurs at index 6 and not preceded by a character “c” as show above in fig 7.0. This is delta$_2$ table’s information and offers 5 positions slide but this is less the than the 6 positions slide offered by delt$_1$ table and the algorithm selects the max value among the values from delta$_1$ and delta$_2$ tables.

														i = 14						
S =	a	b	c	d	b	a	-	d	a	b	a	b	c	d	b	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Patt =					a	b	c	d	b	a	b	c	a	b						
Index =>					1	2	3	4	5	6	7	8	9	10						
														j = 10						

Fig 8.0

The algorithm consider number of string’s characters equals to the pattern length and initially algorithm focused on string [1, 10] but after 4 positions sliding to the right on character mismatch focus shifts to string [5, 14] and as shown above in fig 8.0, index i is updated correctly from i = 8 to i = 8 + 6 = 14.

It is possible on character mismatch that a rightmost occurrence of the character is at the right of the mismatched index in pattern. In this case delta$_2$ table offers relevant and bigger value.

Delta$_2$ Table

The delta$_2$ table provides the information of sub-pattern reoccurrence, the third scenario discussed in pattern searching section. The number of entries in delta$_2$ table is equal to the length of the pattern. The delta$_2$[j] provides the reoccurrence information of sub-pattern begins at the index j + 1.

As discussed the rightmost reoccurrence of sub-pattern Patt[j +1, pattern length] must not preceded by the same character as at pattern[j]. But the algorithm also allows the special / partial reoccurrence of the sub-pattern, if the non-reoccurred part falls outside of the left-end of pattern. To understand this let’s consider the pattern **Patt** = “**abcab**”

Index =>	1	2	3	4	5
Patt =	a	b	c	a	b

The sub-pattern, Patt[4, 5] = “ab” preceded by character “c” at Patt[3] but it doesn’t reoccur in pattern preceded by a different character. There is a reoccurrence of “ab” at index 1 but it is left-end of the pattern and there is no character to precede it. To over such problems algorithm assumes a special character denoted by “$” which doesn’t occur in the pattern. This character “$” at a same time can match or mismatch with any character. The algorithm also assumes that the character “$” present before index 1 as many times as requires in a particular case. So for the sub-pattern Patt[4,5] following would be the pattern,

Index =>	0	1	2	3	4	5
Patt =	$	a	b	c	a	b
Sub-Patt =	c	a	b			

The sup-pattern Patt[4, 5] reoccurs at index 1 and it is not preceded by a character “c”.

The sub-pattern Patt[3, 5] = “cab” is preceded by a character “b” at Patt[2], in this case the pattern would be as follows,

Index =>	-1	0	1	2	3	4	5
Patt =	$	$	a	b	c	a	b
Sub-Patt =	b	c	a	b			

Sub-Pattern Patt[3, 5] reoccurs at index 0 where the character “$” at Patt[0] matches with character “c” and the character “$” at patt[-1] doesn’t matches with character “b”, so the sub-pattern reoccurrence is not preceded by a character “b” as required.

Finally the sub-pattern Patt[2,5] = “bcab” is preceded by a character “a” at Patt[1], in this case pattern would be as follows,

Index =>	-2	-1	0	1	2	3	4	5
Patt =	$	$	$	a	b	c	a	b
Sub-Patt =	a	b	c	a	b			

The sub-pattern Patt[2,5] reoccurs at index -1 (yes the negative index is allowed) where character “$” at Patt[0] and Patt[-1] matches to the character “c” and the character “b” respectively and the character “$” at Patt[-2] doesn’t match with respective a character “a”, so the reoccurrence is not preceded by a character “a” as required.

This technique allows the sub-pattern reoccurrence in the remaining part of the pattern even when the remaining part is smaller than the size of sub-pattern. Now let’s consider the pattern **Patt** = “**abcdabcab**” and it’s $delta_2$ table. The value of each entry of $delta_2$ table is evaluated as follows,

$delta_2$[j] = (j + 1) – sub-pattern reoccurrence index + (pattern length – j)

= pattern length + 1 – sub-pattern reoccurrence index

This formula also has two sub-parts and these sub parts have the same reasoning as the formula of $delta_1$ table has.

Let’s see the values of $delta_2$[j] where j iterates from j = pattern length, to j = 1. The current j is highlighted by gray color filled cell.

Index =>	1	2	3	4	5	6	7	8	9
Patt =	a	b	c	d	a	b	c	a	b
$delta_2$ =									1

At j = 9 as highlighted by gray color filled cell, the last character of pattern, its delta$_2$ value will always be 1 because, at j = 9 there is no sub-pattern starts at j + 1 = 10 rather it is considered as an empty sub-pattern and due to this sub-pattern reoccurrence index is also index 9 and it's delta$_2$ value is,

delta$_2$[9] = (9 + 1) – 9 + (9 – 9) = 1

At j = 8 the sub pattern begins at j + 1 = 9 is "b" but there is no reoccurrence of it where sub-pattern is not preceded by a character other than the character "a". So the sub-pattern reoccurs at index 0 and the preceding character "$" at Patt[-1] doesn't match with character "a" means reoccurrence is not preceded by a character "a" as show below,

delta$_2$[8] = (8 + 1) – 0 + (9 – 8) = 10, as shown below,

Index =>	-1	0	1	2	3	4	5	6	7	8	9
Patt =	$	$	a	b	c	d	a	b	c	a	b
Sub-Patt =	a	b									
delta$_2$ =										10	1

Similarly the value of remaining delta$_2$[j] can be evaluated as follows,

Index =>	1	2	3	4	5	6	7	8	9
Patt =	a	b	c	d	a	b	c	a	b
Sub-Patt =				c	a	b			

delta$_2$ [7] = (7 + 1) – 5 + (9 – 7) = 5

Index =>	-1	0	1	2	3	4	5	6	7	8	9
Patt =	$	$	a	b	c	d	a	b	c	a	b
Sub-Patt =	b	c	a	b							

delta$_2$ [6] = (6 + 1) – 0 + (9 – 6) = 10

Index =>	-2	-1	0	1	2	3	4	5	6	7	8	9
Patt =	$	$	$	a	b	c	d	a	b	c	a	b
Sub-Patt =	a	b	c	a	b							

delta$_2$ [5] = (5 + 1) – (–1) + (9 – 5) = 11

Index =>	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
Patt =	$	$	$	$	a	b	c	d	a	b	c	a	b
Sub-Patt =	d	a	b	c	a	b							

delta$_2$ [4] = (4 + 1) – (–2) + (9 – 4) = 12

Similarly values of all the entries of the delta$_2$ table can be evaluated and the fully constructed delta$_2$ table is shown below in fig 9.0,

delta$_2$ =	15	14	13	12	11	10	5	10	1
Index =>	1	2	3	4	5	6	7	8	9

Fig 9.0

Tables Usage With A Complete Example

The conceptual meaning and creation of delta$_1$ and delta$_2$ tables is explained and now it time to see the usage of these table in algorithm. Let's consider the patter used in sub-section "delta$_2$ table" of previous section that is **Patt** = "**abcdabcab**", so the delta$_2$ table of the pattern is same as discussed in previous section and it's delta$_1$ table is shown below in fig 10.0,

	a	b	c	d
delta$_1$ =	1	0	2	5

delta$_2$ =	15	14	13	12	11	10	5	10	1
Index =>	1	2	3	4	5	6	7	8	9

Fig 10.0

Let consider the string **S** = "**abcda-babeab-a-acabcdabcab**" and see the algorithm step by step as follows,

i = 9

S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=	a	b	c	d	a	b	c	a	b																	
Index =>	1	2	3	4	5	6	7	8	9																	

j = 9

As highlighted by gray color filled cells pattern matching begins at string index i = 9 and pattern index j = 9. The last character of the pattern matches with the respective string character and the matching continues until index j = 7, where a mismatch happens as show below,

i = 7

S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=	a	b	c	d	a	b	c	a	b																	
Index =>	1	2	3	4	5	6	7	8	9																	

j = 7

At i = 7 and j = 7 the mismatch happens, now the algorithm wants to slide the patter to the right and for doing this it reads the values from the delta$_1$ and delta$_2$ tables. The mismatch character "b" has value delta$_1$["b"] zero and detla$_2$[7] has value 5 as show above in fig 10.0. So, the algorithm considers the max value and as a result index i will be updated from i = 7, to i = 7 + 5 = 12 and index j will be assigned to a value equals to the pattern length j = 9, as highlighted by gray color filled cells shown below,

i = 12

S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=				a	b	c	d	a	b	c	a	b														
Index =>				1	2	3	4	5	6	7	8	9														

j = 9

At i = 12 and j = 9 matching continues and two-character matches but at j = 7 a mismatch happens, as highlighted by gray color filled cells show below,

										i = 10																
S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=				a	b	c	d	a	b	c	a	b														
Index =>				1	2	3	4	5	6	7	8	9														
										j = 7																

The mismatch character “e” is not present in the patter, so the delta$_1$[“e”] has value 9 which is equals to the pattern length and delta$_2$[7] has value 5.

	a	b	c	d					
delta$_1$ =	1	0	2	5					
delta$_2$ =	15	14	13	12	11	10	5	10	1
Index =>	1	2	3	4	5	6	7	8	9

Fig 11.0

Then index i = i + max(delta1[“e”] , delta2[7]) = 10 + max(9, 5) = 10 + 9 = 19 and j = 9 as highlighted by gray color filled cells shown below.

																			i = 19							
S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=											a	b	c	d	a	b	c	a	b							
Index =>											1	2	3	4	5	6	7	8	9							
																			j = 9							

At i =19 and j = 9 matching continue and a mismatch happens at j = 6 as shown below.

																i = 16										
S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=											a	b	c	d	a	b	c	a	b							
Index =>											1	2	3	4	5	6	7	8	9							
																j = 6										

Then index i = i + max(delta1[“a”] + delta2[6]) = 16 + max(1, 10) = 16 + 10 = 26 and j = 9 as highlighted by gray color filled cells shown below.

																										i = 26
S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=																		a	b	c	d	a	b	c	a	b
Index =>																		1	2	3	4	5	6	7	8	9
																										j = 9

At i = 26 and j = 9 matching continues but this time no mismatch happens and algorithm finds the pattern in the string. Index j becomes zero as shown below.

																	i = 17									
S =	a	b	c	d	a	-	b	a	b	e	a	b	-	a	-	a	c	a	b	c	d	a	b	c	a	b
Index =>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Patt=																		a	b	c	d	a	b	c	a	b
Index =>																		1	2	3	4	5	6	7	8	9
																	j = 0									

When index j become j = 0, it means all the characters matched and the algorithms stops searching and returns the value i + 1, the first index in the string where the pattern is found. In this case the pattern found at index 18 in the string.

Implementation

The Boyer-Moore algorithm and its variable the Boyer-Moore-Horspool algorithm are the part of standard library since C++17. These are respectively **std::boyer_moore_searcher** and **std::boyer_moore_horspool_searcher**.

February 2025, India