I. Introduction
The Boyer-Moore string search algorithm was developed in 1977 by Robert S. Boyer and J Strother Moore. It is considered one of the most efficient algorithms for usual applications. For example, a simplified version of BM is often implemented in text editors for search and substitute commands. The algorithm preprocesses the pattern, making it ideal for situations where the pattern is much shorter than the text. It then scans characters in the pattern from right to left. If the character being observed does not match with the corresponding character in the text, it uses two precomputed tables to skip ahead in the text. These tables are computed using the bad character rule and good suffix rule.

II. String Matching
The problem we wish to solve is as follows: given a text T of length n and pattern P of length m, with \( m \leq n \), we want to find all occurrences of P in T. It can also be modified to find the first occurrence of P in T.

III. Shift Rules
A. The Bad Character Rule
If character \( T[i + j] \) mismatches character \( P[j] \), we call \( T[i] \) the “bad” character. The bad character rule finds the next occurrence of the bad character left of the current position, \( j \). The pattern can be shifted so the “bad character” in the pattern can be aligned with the “bad character” in the text. If there is no occurrence of the bad character, then it calculates a shift to bring the pattern past the bad character in the text.

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The comparison d-c causes a mismatch. The symbol ‘d’ occurs a position 0 in the pattern. Therefore, the pattern gets shifted to align the occurrence of ‘d’ in the text and in the pattern. Index \( i \) is the index where the pattern is being aligned in the text. A shift means \( i \) get increased or decreased by a certain amount. In this
example, i is incremented by 2, so the pattern is shifted by 2 places, as shown in the third row.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A function called \( \text{last}() \) is required. As input it takes a character and returns the index of the rightmost occurrence of that character in the pattern. If the character does not appear in the pattern, the function returns -1.

We preprocess the pattern by initializing a table of length \( |\Sigma| \) (\( \Sigma \) is the alphabet) to -1. We scan through the pattern and set table\[P[i]\] = i. This way the table stores the rightmost value of all characters in the pattern.

B. The Good Suffix Rule

The good suffix rule uses an array \( s \) where each entry \( s[i] \) contains the shift distance of the pattern if a mismatch at position \( i - 1 \) occurs. In order to determine shift distances, there are two cases to consider.

Case 1: The matching suffix occurs somewhere else in the pattern.

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>P</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 2: Part of the matching suffix occurs at the beginning of the pattern.

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 1:

We define the “border” of \( x \) as a substring \( r \), such that \( r = x_{b} \cdots x_{b+1} \) and \( r = x_{b-k} \cdots x_{k-1} \) for \( b \) between 0 and \( k - 1 \). The border of \( x \) is a substring that is both a proper prefix and proper suffix of \( x \). The width of the border is the length \( b \).

Example:
Consider the string “abcabab”.
Proper suffixes are: “a”, “ab”, “abc”, “abca”, “abcab”, “abcaba”.
Proper prefixes are: “b”, “ab”, “bab”, “abab”, “cabab”, “bcabab”.
Borders are: \(\varepsilon\), “ab”.
Note that \(\varepsilon\) is a border with width 0.

The preprocessing is similar to that of the Knuth-Morris Pratt algorithm.
Therefore, we need to determine the borders of the suffices of the pattern.
Moreover, we need to make sure that the border cannot be extended farther to the left by the same character because this would result in another mismatch after shifting.
The first part of the preprocessing algorithm computes an array \(f\). Each entry \(f[i]\) contains the starting position of the widest border of the suffix of the pattern beginning at index \(i\). The suffix \(\varepsilon\) beginning at position \(m\) has no border, so \(f[m] = m + 1\).
Each border is computed by checking if there exists a shorter known border that can be extended to the left by the same character.
However, if the border cannot be extended to the left, we still know that there is some shift. The corresponding shift distance is saved in another array, \(s\), if that entry is not already occupied. This shift difference is equal to \(i - j\), where \(i\) is the index of the beginning of the suffix and \(j\) is index of where the border of the suffix begins.
Example:

<table>
<thead>
<tr>
<th>(i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>(f)</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Consider the suffix “babab”, which starts at \(i = 2\). The widest border of “babab” is “bab”, which starts at \(i = 4\). Therefore, \(f[2] = 4\). If we look at the suffix “ab”, starting at \(i = 5\), we see that the widest border is \(\varepsilon\), so \(f[5] = i + 1 = 6\).
Looking at suffix “babab” again, starting at \(i = 2\), it has border “bab”, starting at \(i = 4\). It is impossible to extend the border to the left because \(P[1] \neq P[3]\). Therefore, we store the shift difference \(4 - 2 = 2\) in \(s[4]\).

Case 2:
If only a part of the matching suffix occurs at the beginning of the pattern, then it is a part of the border of the pattern. For each suffix, compute the the widest
border that occurs in that suffix. Store the starting position of the widest border at \( f[0] \). The value at \( f[0] \) is initially stored in all free positions of array \( s \). As the suffix becomes shorter than \( f[0] \), the algorithm continues with the next widest border, \( f[j] \).

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>s</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

IV. Examples of Simplified Boyer-Moore Algorithms
A. Version i

\[
i = 0, \ j = m - 1 \\
\text{while } (i \leq n - m) \\
\{
\quad j = m - 1 \\
\quad \text{while } (j \geq 0 \text{ and } p[j] = t[i + j]) \\
\quad \quad j-- \\
\quad \quad \text{if } (j < 0) \\
\quad \quad \quad \text{return } i \\
\quad \quad i = i + j - \text{last}(T[i + j]) \\
\}
\]

Return “no match”

Example:

\( P = ccba \quad T = abcacbca \)

<table>
<thead>
<tr>
<th>x</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>last(x)</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>-1</td>
<td>...</td>
</tr>
</tbody>
</table>

We compute our table using the bad character rule.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>P</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( i = 0, \ j = m - 1 = 3 \)

We see that \( T[i + j] = P[j] \) so we decrement \( j \).
i = 0, j = 2.

However, T[2] ≠ P[2], so we recalculate i using the formula: \(i + j - \text{last}(T[i + j])\)
and j gets resut to m - 1.

\(i = 0 + 2 - \text{last}(T[2]) = 2 - 1 = 1\)

\(j = 3\)

\(i = 1, j = 3\)

However, despite this shift, T[4] ≠ P[3], so we recalculate i again and set j = 3.

\(i = 1 + 3 - \text{last}(T[4]) = 1 + 3 - 1 = 3\)

\(j = 3\)

\(i = 3, j = 3\)

Again, there is no match at T[6] ≠ P[3].

\(i = 3 + 3 - \text{last}(T[6]) = 3 + 3 - 2 = 4\)

\(j = 3\)
Finally, $T[7] = P[3]$, so we decrement $j$. Everytime we decrement $j$, we see that $T[i + j] = P[j]$. Once $j < 0$, we know we have compared every element from the pattern to something in the text, so we have found a match.

B. Version ii

The main issue with the simplified BM shown above is that it can result in “bad” shifts. We call a shift “bad” if it shifts the pattern backwards (to the left). In fact, only using the Bad Character Rule can result in shifts that move the pattern too far left.

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>P</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Create table using last() function.

<table>
<thead>
<tr>
<th>x</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>last(x)</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>-1</td>
<td>...</td>
</tr>
</tbody>
</table>

$i = 0$

$j = 2$

$i = i + j - \text{last}(T[0 + 2]) = 0 + 2 - 5 = -3$

We definitely do not want to shift backwards, and we certainly do not want to shift off the array.

An easy fix is to replace the Good Suffix Rule with a shift of +1. This way we avoid shifting backwards because we always choose the maximum shift distance.
V. Full BM
   A. The full BM algorithm precomputes two tables using the Good Suffix and Bad Character Rules. It then scans through the pattern and text, comparing from the back to the front of the pattern. When a mismatch occurs, take the maximum shift value.

VI. Time Complexity
   A. To recap, we have previously defined m as the length of the pattern P, and n as the length of the text T that we are using in our algorithm.
   B. Preprocessing for the Bad Character Rule is $O(m + |\Sigma|)$ because we have to scan through the entire pattern as well as initialize a value in the table for every possible character in the input alphabet.
   C. Preprocessing for the Good Suffix Rule is $O(n)$ because we only need to compare the text with other parts of the text.
   D. Running Boyer-Moore is $O(nm + |\Sigma|)$. If m, length of the pattern, is proportional ($cn$, for some constant $c$) to n, the length of the text, then the time complexity is $O(n^2)$. The time complexity upper bound is not any better than brute force, however, we are ideally skipping over sections of the text, so on average, the time complexity should be less than $O(nm)$. An example worst case scenario for Boyer-Moore would be checking for $P = xaaaaa$ in text $T = aaaa...aaaa$. This would force the algorithm to look at all but one of the characters in the pattern and would only allow shifts of size one.
   The worst case for brute force is similar, except we would have the mismatched character at the end of the pattern instead of the beginning: $P = aaaaaa$ and $T = aaaaa...aaaaa$.
   E. The Galil Rule is a Boyer-Moore optimization which speeds up the process of matching each alignment. It allows us to achieve a worst-case $O(n + m)$ runtime by keeping track of previously matched substrings for reference in further comparisons, which allows us to increase our efficiency for comparison significantly.

VII. Comparison with other String Matching Algorithms
   A. Note that Boyer-Moore is considered the most efficient and reliable string matching algorithm for natural language searches, due to the high probability of finding mismatches given the relatively large alphabet. This chance of mismatching lets us skip over larger bodies of text more often during the search.
   B. Brute Force
O(\(nm\)) expected runtime, but probably the best option for very small patterns, since our runtime will essentially be linear for small values of \(m\). Unlike Boyer-Moore, it gets less efficient as the pattern gets longer. However, it is the easiest to implement.

C. Knuth-Morris-Pratt Algorithm (KMP)

1. Because KMP remembers prior matches of the pattern for use in later comparisons when searching, it works best when there is a small alphabet to work with - thus, KMP is superior for matching alphabets such as binary strings (where the alphabet consists of 0 and 1) and DNA sequences (which consist of A, T, G, and C). This is because smaller alphabets have a higher probability of finding sequence matches.

2. KMP also has a lower worst-case runtime than the simpler (non-Galil Rule) Boyer-Moore algorithms, being \(O(n + m)\).

D. Z Algorithm

Like KMP, the Z algorithm has a lower worst-case time complexity than the most basic variations of Boyer-Moore, being \(O(n + m)\), while being more easily understandable than KMP. However, Boyer-Moore still outperforms it in practice for natural language searching, especially when using the Galil Rule.

VIII. Sources


D. [http://dl.acm.org/citation.cfm?id=359146.359148](http://dl.acm.org/citation.cfm?id=359146.359148)