Boyer-Moore String Matching: An Efficient Approach

Sumanth

June 18, 2025

Abstract

The Boyer-Moore algorithm is an efficient **string matching** technique that finds all occurrences of a pattern in a text by leveraging preprocessing to skip unnecessary comparisons. Using the **bad character heuristic**, it achieves better performance than naive string matching, especially for large alphabets. This document details the algorithm's formulation, pseudocode, complexity analysis, practical applications, and implementation considerations, focusing on the bad character heuristic as implemented in the provided code and its role as a foundation for advanced string matching techniques.

1 Introduction

The Boyer-Moore algorithm is a highly efficient **string matching** algorithm that locates all occurrences of a pattern within a text. Unlike naive string matching, it uses preprocessing heuristics—primarily the **bad character** and **good suffix** rules—to skip multiple characters when a mismatch occurs, reducing the number of comparisons. The provided implementation focuses on the bad character heuristic, making it suitable for educational purposes and practical applications with large alphabets. Its efficiency and conceptual clarity make it a cornerstone for understanding advanced string matching techniques.

2 Problem Formulation

Given a text T of length n and a pattern P of length m, find all starting indices i in T such that the substring $T[i \ldots i+m-1]$ matches P. The output is a list of indices where the pattern occurs in the text.

2.1 Key Idea

The algorithm aligns the pattern with the text and compares characters from right to left. Upon a mismatch, it uses the **bad character heuristic** to shift the pattern based on the mismatched character's last occurrence in the pattern, maximizing the skip distance. This approach minimizes comparisons, especially when the pattern contains distinct characters or the alphabet is large.

3 Boyer-Moore Algorithm

The algorithm preprocesses the pattern to build a bad character table and performs right-to-left comparisons. Below is the pseudocode, reflecting the provided code:

Algorithm 1 Boyer-Moore String Matching (Bad Character Heuristic)

Input: Text $T[1 \ldots n]$, Pattern $P[1 \ldots m]$
Output: List of starting indices where P occurs in T
Procedure BadCharHeuristic($P, m,$ badchar):
for $i = 0$ to 255 **do**
    badchar$[i] \leftarrow -1$
end for
for $i = 0$ to $m-1$ **do**
    badchar$[P[i]] \leftarrow i$
end for
Initialize indices $\leftarrow \emptyset$
Initialize badchar$[0 \ldots 255]$
Call BadCharHeuristic($P, m,$ badchar)
$s \leftarrow 0$ ▷ Shift of pattern
while $s \leq n - m$ **do**
    $j \leftarrow m-1$
    while $j \geq 0$ and $P[j] = T[s+j]$ **do**
        $j \leftarrow j-1$
    end while
    if $j < 0$ **then**
        Add s to indices
        $s \leftarrow s + (s+m < n ? m - \text{badchar}[T[s+m]] : 1)$
    else
        $s \leftarrow s + \max(1, j - \text{badchar}[T[s+j]])$
    end if
end while
Return indices

3.1 Bad Character Heuristic

The `BadCharHeuristic` function creates a table mapping each character to its rightmost position in the pattern (or -1 if absent). When a mismatch occurs at position j in the pattern with character $T[s+j]$, the pattern is shifted to align the last occurrence of $T[s+j]$ in the pattern with $T[s+j]$, or by at least 1 if no such occurrence exists.

3.2 Good Suffix Heuristic

The provided code omits the good suffix heuristic, which shifts the pattern based on matched suffixes to handle repetitive patterns. It can be added to further optimize performance but increases implementation complexity.

4 Complexity Analysis

- **Time Complexity**:
  - Preprocessing: $O(m+\sigma)$, where σ is the alphabet size (256 in the code).
  - Searching:
    - Best Case: $O(n/m)$, when large shifts occur due to frequent mismatches.
    - Average Case: $O(n/m)$, with sublinear behavior for large alphabets.
    - Worst Case: $O(n \cdot m)$, for repetitive patterns (e.g., $T = \text{AAAAA}, P = \text{AAAA}$).
- **Space Complexity**: $O(\sigma)$, for the bad character table (fixed at 256 in the code).

5 Practical Applications

Boyer-Moore is used in:

- **Text Processing**: Searching patterns in editors or search engines.
- **Bioinformatics**: Finding DNA subsequences in large genomic texts.
- **File Search**: Efficient pattern matching in large files or logs.
- **Education**: Teaching advanced string matching concepts.

6 Example

Given the text $T = \text{ABABCABAB}$ and pattern $P = \text{ABAB}$:

Text	A	B	A	B	C	A	B	A	B
Index	0	1	2	3	4	5	6	7	8

Bad character table for $P = \text{ABAB}$:

Char	A	B	Others
Position	2	3	-1

The algorithm finds matches at indices 0 and 5, shifting the pattern efficiently using the bad character rule.

7 Limitations and Extensions

- **Limitations**:
  - Worst-case $O(n \cdot m)$ complexity for repetitive patterns.
  - Bad character heuristic alone is less effective for small alphabets or repetitive patterns.
- **Extensions**:
  - **Good Suffix Heuristic**: Add to handle repetitive patterns, improving average-case performance.
  - **Galil Rule**: Optimize by skipping guaranteed matches after a full match.
  - **Horspool Variant**: Simplify to use only the bad character rule for the rightmost text character.

8 Implementation Considerations

- **Alphabet Size**: The code assumes a 256-character alphabet; adjust for Unicode or larger alphabets.
- **Input Validation**: Check if $m \leq n$ and handle empty text or pattern cases.
- **Bad Character Table**: Initialize efficiently for the actual character set to save memory.
- **Good Suffix Integration**: Consider adding the good suffix heuristic for better performance on repetitive texts.
- **Output Format**: Store indices in a list or print directly, ensuring clarity for multiple matches.

9 References

- GeeksForGeeks: Boyer-Moore Algorithm
- Wikipedia: Boyer-Moore String-Search Algorithm