Report

1. Boyer-Moore Implementation Approach

1.1 Overview

The Boyer-Moore algorithm is implemented in the BoyerMoore class within **Analysis.java.** This implementation uses two main heuristics:

1. **Bad Character Rule**
2. **Good Suffix Rule**

1.2 Bad Character Rule

The bad character rule stores the last occurrence position of each character in the pattern using a HashMap. When a mismatch occurs, we use this table to determine how many positions we can skip.

How it works:

- Create a HashMap mapping each character to its rightmost position in the pattern
- On mismatch, calculate shift based on the mismatched character's position
- Time Complexity: O(m) preprocessing, O(1) lookup

Example: For pattern "ABCAB", the table stores: A→3, B→4, C→2

1.3 Good Suffix Rule

The good suffix rule is more complex and handles two cases:

- **Case 1:** Find another occurrence of the matched suffix preceded by a different character
- **Case 2:** Find the longest suffix that matches a prefix of the pattern

Implementation approach:

- Calculate border positions array from right to left
- Compute shift values for each position
- Handle both strong and weak good suffix cases
- Time Complexity: O(m) preprocessing

The border position calculation is similar to KMP's failure function but applied to suffixes.

1.4 Main Search Algorithm

The search algorithm:

- Compares pattern with text from right to left
- On mismatch, calculates both bad character shift and good suffix shift
- Takes the maximum of both shifts to skip positions
- Continues until the end of text

Edge cases handled:

- Empty pattern (returns all positions)
- Pattern longer than text (returns empty result)
- Single character patterns
- Complete text match

1.5 Complexity Analysis

- **Preprocessing:** $O(m + \sigma)$ where m is pattern length, σ is alphabet size
- **Search:**
  - Best case: $O(n/m)$ - can skip multiple characters
  - Average case: $O(n+m)$
  - Worst case: $O(n \cdot m)$
- **Space:** $O(m + \sigma)$

---

2. Pre-Analysis Strategy (StudentPreAnalysis Class)

The StudentPreAnalysis class implements an intelligent algorithm selection system that analyzes text and pattern characteristics to choose the most suitable algorithm, maximizing the frequency of selecting the fastest algorithm for each test case.

2.1 Algorithm Selection Criteria

NAIVE - For Simple Cases:

- Selected when text is very short ($n < 150$)
- Selected when pattern is very short ($m \leq 3$)
- Selected when text is barely longer than pattern ($n \leq m + 5$)

- **Rationale:** For small inputs, preprocessing overhead outweighs the benefits of advanced algorithms

RABIN-KARP - For Small Alphabet with Repetition:

- Selected for small alphabet (≤ 4 unique characters) with repetitive patterns
- Selected for long text ($n > 400$) with short pattern ($m \leq 3$)
- Selected for medium-long text ($n > 250$) with medium-short pattern ($m < 25$)
- **Rationale:** Rolling hash provides O(n+m) average performance, handles small alphabets well with minimal collision

BOYER-MOORE - For Large Alphabet or Long Patterns:

- Selected for small alphabet without repetition
- Selected for long text ($n > 250$) with long pattern ($m \geq 25$)
- **Rationale:** Bad character rule works well with large alphabets, long patterns provide more skip opportunities

KMP - For Repetitive Structures:

- Selected for medium-length text ($150 \leq n \leq 250$) with repetitive patterns
- **Rationale:** LPS (Longest Proper Prefix Suffix) table exploits pattern structure, guaranteed O(n+m) with no worst-case degradation

2.2 Helper Functions

Small Alphabet Detection:

- Examines first 10 characters of pattern
- Counts unique characters
- Returns true if unique count ≤ 4
- Time Complexity: O(min(m, 10)) - minimal overhead

Repetition Detection:

- Checks for dominant character (e.g., "AAAAB")
- Checks for period-2 repetition (e.g., "ABAB")
- Examines first 8 characters only
- Time Complexity: O(min(m, 8)) - fast heuristic

2.3 Strategy Logic

The pre-analysis runs in O(min(m, 10)) time with minimal overhead because:

- Only examines the pattern (not the text)
- Uses simple counting operations
- No complex data structures
- Early exits when conditions are met

Key Thresholds:

- n < 150: Preprocessing not beneficial
- m ≤ 3: Naive is competitive
- m ≥ 25: Boyer-Moore skip advantage significant
- Alphabet ≤ 4: Use specialized algorithms

2.4 Design Decisions

- **Pattern-only analysis:** Pattern characteristics are usually representative of the search domain
- **Limited sampling:** First 8-10 characters provide sufficient hints without overhead
- **Simple heuristics:** Fast approximate checks rather than comprehensive analysis
- **Trade-off balance:** Analysis time vs. potential speed gain

3. Research Documentation

3.1 Resources Used

1. Boyer-Moore Algorithm - Wikipedia

- URL: https://en.wikipedia.org/wiki/Boyer–Moore_string-search_algorithm
- Used for: Understanding theoretical foundation of bad character and good suffix rules

2. GeeksforGeeks - Pattern Searching Algorithms

- URL: https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/
- Used for: Comparative analysis of different string matching algorithms

3. Algorithm Design Manual by Steven Skiena

- Used for: Understanding time complexity trade-offs
- Key takeaway: Preprocessing overhead vs. search efficiency balance

Online Resources

4. YouTube - Abdul Bari Algorithm Lectures

- Used for: Visual explanation of Good Suffix Rule

5. Stack Overflow - Boyer-Moore Implementation Questions

- Used for: Edge cases and implementation details
- Key takeaway: Handling empty pattern, pattern > text scenarios

Step 3: Sources Used

Below are some of the resources we use:

- https://www.tutorialspoint.com/data_structures_algorithms/boyer_moore_algorithm.htm

https://www.geeksforgeeks.org/dsa/pattern-searching/

https://stackoverflow.com/questions/5423886/boyer-moore-algorithm-implementation

https://medium.com/tech-in-depth/string-matching-algorithms-271d50a2a265

https://medium.com/@AlexanderObregon/finding-patterns-with-boyer-moore-in-java-75c8a22d741a

We were careful to understand the logic behind these implementations rather than copy any particular version directly. The final implementation in our homework reflects what we learned and synthesized from these sources.

2. Our Journey

Understanding string matching algorithms turned out to be much more difficult than we initially expected. Although the ideas behind Boyer–Moore are elegant, the details—especially the good-suffix preprocessing—are surprisingly complicated. Even after we thought we finally understood the concept, examining the code again often made everything feel confusing.

The good-suffix rule in particular felt like a conceptual "black box" at first. While the idea is brilliant, tracing the exact transitions and array updates during preprocessing was a genuine challenge for both of us.

The “GoCrazy” algorithm part was also psychologically difficult—not because of the coding, but because designing something “new” felt overwhelming while we were still struggling to understand the existing algorithms properly. Since innovation in this context requires a strong foundation, we felt unprepared to propose a new strategy confidently. For this reason, we decided not to implement a new algorithm and instead focused our effort on fully understanding and correctly implementing Boyer–Moore.

Despite the challenges, we genuinely enjoyed the rest of the assignment. Working through the test cases, comparing algorithms, and analyzing their behaviors helped us understand Boyer–Moore and other string matching techniques far more deeply than before.

Although the learning curve was steep, we ultimately feel more confident about how pattern-matching algorithms operate internally.

This homework pushed us to learn beyond class material, guided us to explore documentation and external resources, and taught us how to validate, compare, and refine different implementations. In that sense, it was a valuable learning experience, even if at times it was frustrating.

Murat Emir Öncül - 22050111057
Reyyan Naz Doğan - 22050111012