String Matching Algorithms - Project Report

1. Boyer-Moore Implementation Approach

Overview

My implementation of the Boyer-Moore algorithm focuses on the **Bad Character Heuristic**. This heuristic allows the algorithm to skip sections of the text when a mismatch occurs, by aligning the mismatched character in the text with its last occurrence in the pattern.

Implementation Details

- **Preprocessing:** Instead of using a fixed-size array (which would need to be size 65,536 for Unicode support), I used a `HashMap<Character, Integer>`. This map stores the index of the last occurrence of each character in the pattern.
- **Why HashMap?** Initializing a 64k array takes significant time. For a pattern of length 5, we only need to store 5 entries. The `HashMap` approach makes the initialization time proportional to the pattern length ($O(m)$) rather than the alphabet size ($O(\Sigma)$).
- **Search Logic:** The algorithm scans the pattern from right to left. Upon a mismatch, it calculates the shift based on the bad character rule. It also handles the case where the pattern is found, shifting by 1 or by the bad character rule for the character immediately following the match.

2. GoCrazy Algorithm (Boyer-Moore-Horspool)

Design and Rationale

For the "GoCrazy" task, I implemented the **Boyer-Moore-Horspool** algorithm.

Why Horspool? Standard Boyer-Moore uses two rules: Bad Character and Good Suffix. The Good Suffix rule is complex to implement and computationally expensive to preprocess. Research suggests that for large alphabets (like ASCII/Unicode) and random text, the Good Suffix rule rarely provides a larger shift than the Bad Character rule.

Horspool simplifies Boyer-Moore by: 1. Dropping the Good Suffix rule entirely. 2. Modifying the Bad Character rule to always base the shift on the **last character of the current text window**, regardless of where the mismatch actually occurred.

Optimization

Like my Boyer-Moore implementation, I used a `HashMap` for the shift table to support Unicode efficiently without the memory/time penalty of large arrays. This makes the algorithm extremely lightweight and fast for long patterns.

3. Pre-Analysis Strategy

The Logic

My `StudentPreAnalysis` class uses a decision tree based on empirical testing results:

1. **Short/Medium Patterns ($m \leq 20$): Use Naive**
   - **Reasoning:** Java's `String.charAt` and simple loops are heavily optimized by the JIT compiler. For patterns shorter than ~20 characters, the overhead of allocating memory and building tables for BM/KMP/Horspool exceeds the time saved by skipping characters. Naive is consistently the winner here.
2. **Repeating Prefixes: Use KMP**
   - **Reasoning:** If a pattern has a repeating prefix (e.g., "AAAA"), Naive performs poorly ($O(n \cdot m)$). KMP handles this in $O(n)$ by using the failure function to avoid re-scanning text. I implemented a helper `hasRepeatingPrefix` to detect this specific case.
3. **Long Patterns ($m > 20$): Use GoCrazy (Horspool)**
   - **Reasoning:** As the pattern gets longer, the probability of a mismatch increases, and the potential shift distance grows. Horspool can skip large sections of text, making it significantly faster than Naive for long patterns.

Analysis of Results

- **Naive Dominance:** Surprisingly, the Naive algorithm won the majority of test cases. This highlights that for modern hardware and JVMs, simple code often beats "smarter" but more complex code for small datasets.
- **Optimization Impact:** Switching from arrays to HashMaps for the shift tables reduced the execution time of Boyer-Moore and GoCrazy by nearly 100x in some cases, making them competitive.
- **Pre-Analysis Accuracy:** The final strategy correctly identifies the "sweet spots" for each algorithm, choosing KMP for repetitive patterns and Horspool for long ones, while defaulting to the efficient Naive for the rest.

4. Performance Optimization Update

After the initial implementation, I noticed that `BoyerMoore` and `GoCrazy` were significantly slower than `Naive` for almost all test cases. - **Problem:** I was initializing a `new int[65536]` array for every call to handle Unicode. This initialization overhead was dominating the execution time. - **Solution:** I switched to using `HashMap<Character, Integer>`. - **Result:** - Boyer-Moore average time dropped from ~315,000 s to ~8,100 s (~**38x speedup**). - GoCrazy average time dropped from ~177,000 s to ~7,700 s (~**23x speedup**). - This optimization made the advanced algorithms competitive and allowed the Pre-Analysis

logic to actually be meaningful.

Name: Burhan Koçak

Student_ID: 21050111044