String Matching Algorithms: Implementation Report

Muhammed Enes Uğraş 23050151030
Nurten Çiftçioğlu 21050111027

1. Boyer-Moore Implementation Approach

Our Boyer-Moore implementation utilizes both the bad character rule and the good suffix rule to achieve efficient pattern matching. The bad character rule was straightforward to implement. We created a hash table storing the rightmost occurrence of each character in the pattern. The good suffix table implementation was significantly more challenging and took considerable time to understand. We initially attempted to learn from GeeksforGeeks but struggled with their approach. A Medium article provided clearer explanations that helped us grasp the underlying logic. Once we understood the concept, we returned to the GeeksforGeeks implementation to complete our work.

Our good suffix implementation handles two distinct cases. The first case occurs when the matched suffix appears elsewhere in the pattern with a different preceding character. The second case handles situations where a prefix of the pattern matches a suffix of the matched portion. We computed border positions and filled the shift table for both cases. The algorithm uses these precomputed tables to determine the maximum shift at each mismatch position. The overall algorithm flow starts at end of the pattern and compares the pattern from right to left. When a mismatch occurs, we calculate shifts from both heuristics and move by the maximum of the two. When we find a complete match, we use the good suffix table to determine the next search position.

2. GoCrazy Algorithm Design and Rationale

The GoCrazy algorithm was not implemented.

3. Pre-Analysis Strategy

Our pre-analysis strategy attempted to select algorithms based on computable features without actually running the patterns. We wanted preprocessing to be lightweight, calculating simple statistics and making decisions through a decision tree structure. We reasoned that different algorithms perform better under different conditions. Boyer-Moore excels with large alphabets and longer patterns, while KMP works well with repetitive patterns. Rabin-Karp handles multiple pattern searches efficiently, and naive can be faster for very short patterns due to low overhead. We extracted features including text length, pattern length, unique character counts in both text and pattern, pattern repetitiveness through character frequency, alphabet diversity ratio.

We initially had ChatGPT generate a decision tree structure based on our parameter thresholds. The thresholds were tweaked based on intuitions such as favoring Boyer-Moore for higher unique character counts, favoring KMP for more repetitive patterns, favoring naive for very short patterns, and considering Rabin-Karp for very large texts that not favored by KMP or Boyer-Moore. This resulted in a large if-else structure that's hard to read but theoretically avoids runtime overhead.

We chose this approach because it's deterministic and fast without requiring model inference, it uses intuitive heuristics from algorithm theory, and it avoids the overhead of running multiple algorithms.

We briefly explored training a machine learning model to select algorithms, which led us to find recent research papers on this exact problem. The paper "Entropy-Based Approach in Selection Exact String-Matching Algorithms" discusses preprocessing patterns and calculating entropy for algorithm selection. Another paper titled "Boosting exact pattern matching with extreme gradient boosting" uses numerous features from distinct string matching algorithms with tree-based models and achieved 11% speed improvements. However, we rejected the machine learning approach because no suitable datasets exist with text-pattern-expected triplets, creating such a dataset would be time-consuming and biased, we'd need massive amounts of data covering diverse use cases, and the preprocessing time might negate any performance gains.

4. Analysis of Results

Our preprocessing strategy achieved approximately 70-80% success rate in selecting the optimal or near-optimal algorithm. However, this success rate is heavily dependent on the extremely close runtimes between different algorithms for our test cases, which fluctuates the accuracy within that range. The primary challenge we encountered was that most test cases favored the naive algorithm due to pattern sizes. The overhead of preprocessing and complex heuristics often approached or exceeded the marginal time differences between algorithms. When algorithm runtimes differ by only microseconds, even small variations in execution conditions can change which algorithm appears fastest. When we re-ran tests, different algorithms would occasionally win across runs, suggesting that the performance differences were often within the noise margin. This inconsistency made it difficult to validate whether our selection strategy was genuinely effective or simply benefiting from the fact that most algorithms performed similarly on our test cases. We faced a fundamental evaluation dilemma. We couldn't evaluate patterns deeply during preprocessing without incurring high costs, but light preprocessing proved insufficient for making truly informed decisions. Without thorough analysis, we couldn't confidently predict the best algorithm, yet thorough analysis defeats the purpose of fast preprocessing. This trade-off meant our heuristics were based on educated guesses rather than comprehensive pattern analysis. There were notable edge cases that highlighted unexpected behaviors. With Unicode character handling, choosing Rabin-Karp over Boyer-Moore or vice versa resulted in identical 8μs differences in opposite directions. The decision tree structure we generated makes our code virtually unreadable, and we're uncertain if avoiding method call overhead actually provides meaningful benefits given modern JVM optimizations and the small absolute time differences we're trying to optimize.

While we successfully implemented Boyer-Moore with both heuristics and gained understanding of string matching algorithms, our preprocessing strategy showed that practical algorithm selection is challenging when dealing with short patterns and close runtimes. The 70-80% success rate shows that heuristic-based selection can work reasonably well. However, given the limited performance gains from choosing the optimal algorithm and the added code complexity, we believe the preprocessing approach may not be justified for our specific use cases.