String Matching Algorithms Report

Boyer–Moore, GoCrazy, and Pre-Analysis Strategy

Student Name

1 Boyer–Moore Implementation Approach

For the Boyer–Moore algorithm, my goal was to implement both of its major heuristics: the *bad character rule* and the *good suffix rule*. These two components allow Boyer–Moore to skip ahead in the text more aggressively than simpler algorithms like the Naive method.

I started with a preprocessing phase. First, I constructed the bad character table, which records the last position of each character in the pattern. This lets the search phase shift the pattern based on where the mismatching character occurs in the pattern.

The second and more complex part was building the good suffix table. To do this, I computed a suffix array that identifies how much of the pattern's suffix matches a prefix or some internal substring of the pattern. Using this information, I generated the actual shift values for cases where the mismatch happens near the end of the pattern.

In the search stage, the algorithm compares characters from right to left and uses both heuristics to determine how far to shift the pattern after each mismatch or match. When a full match occurs, the starting index is recorded and the good suffix table decides the next shift amount.

Overall, the implementation closely follows the standard description of Boyer–Moore. The results show that it behaves as expected and performs well on large texts or texts with a diverse alphabet.

2 GoCrazy Algorithm Design and Rationale

In this part, I designed a hybrid algorithm that combines two ideas:

- A bit-parallel Shift–Or style matcher for short patterns (up to 64 characters)
- A simplified Boyer–Moore–Horspool style skip method for longer patterns

The bit-parallel part uses bit masks to track which pattern positions correspond to each character. With each new character in the text, a bitwise update is applied to a

state variable that represents how much of the pattern matches so far. When the highest bit becomes set, it indicates a match at that position.

For longer patterns, the algorithm switches to a Horspool-like approach using a simple bad-character shift based on the last character of the pattern. This keeps preprocessing cost relatively low while still allowing the search to skip multiple characters on mismatches.

In practice, `GoCrazy` does not outperform the classical algorithms in most cases, but it still works correctly and is particularly fast in the case where the pattern is longer than the text, because it can terminate early.

3 Pre-Analysis Strategy and Motivation

To build this strategy, I examined the timing results for each algorithm and looked for patterns in when each one tends to be the fastest.

The decision logic is based on several simple characteristics:

- **Pattern length:** very short patterns are handled efficiently by the Naive algorithm.
- **Repetition in the pattern:** if the pattern has a high LPS (longest prefix-suffix) value, KMP usually performs well.
- **Alphabet size:** Boyer–Moore benefits from larger alphabets, especially on longer texts.
- **Text length:** for very long texts, Boyer–Moore or Rabin–Karp are often good options.
- **Special cases:** when the pattern is longer than the text, GoCrazy returns quickly and becomes the best choice.

Using these observations, I implemented a heuristic in `StudentPreAnalysis` that:

- chooses Naive for very short patterns,
- chooses KMP when the pattern shows strong repetition,
- chooses Boyer–Moore for large texts with relatively large alphabets,
- chooses Rabin–Karp in some long-text, long-pattern situations,
- chooses GoCrazy when the pattern length is greater than the text length,
- and falls back to Naive in other small or medium-sized cases, since Naive was often fastest there.

4 Analysis of Results

All algorithms passed every test case, which shows that the implementations are correct. The performance results also follow the expected behaviour of the algorithms:

- **Naive** was often the fastest on smaller test cases and very short patterns, because its overhead is small.
- **KMP** performed best when the pattern had strong internal repetition and when there were many overlapping matches.
- **Boyer–Moore** was one of the fastest algorithms on longer texts and on inputs with a large or diverse alphabet.
- **Rabin–Karp** was occasionally the fastest on some longer cases with suitable pattern and text lengths.
- **GoCrazy** was competitive mainly when the pattern was longer than the text, where it returns early.

The pre-analysis strategy was able to select the fastest or near-fastest algorithm in many cases by using only simple characteristics such as pattern length, text length, alphabet size, and pattern repetition.