String Matching Algorithms Implementation and Analysis Report

Mehmet Demir - 22050111024
Ayşe Zülal Şimşek - 22050111047

December 13, 2025

Contents

1 Introduction

This project focuses on string matching algorithms, which are essential in text processing, search engines, DNA sequence analysis, and many other applications. Our main goal was to choose the most suitable algorithm for different scenarios and develop our own custom algorithm.

1.1 Project Scope

We worked on three main tasks during this assignment:

1. Boyer-Moore Algorithm Implementation: We implemented the Boyer-Moore algorithm in Java, which is one of the classic string matching algorithms. This algorithm works efficiently especially with large alphabets and uses right-to-left comparison.

2. GoCrazy - Our Custom Algorithm: We developed a hybrid approach by combining the strengths of existing algorithms. The GoCrazy algorithm uses a 3-point fingerprint mechanism for fast filtering and Horspool skip table for safe jumping.

3. Pre-Analysis System: We created a decision mechanism that automatically selects the most suitable algorithm based on different text and pattern characteristics. This system analyzes factors like pattern length, text size, repeating characters, and alphabet diversity.

1.2 Our Approach

When starting the project, we first studied the existing algorithms and tried to understand in which scenarios each one performs well. For Boyer-Moore implementation, we researched academic sources and the theoretical foundations of the algorithm.

While developing GoCrazy, we observed that Boyer-Moore is fast but has a complex structure. We decided to design a simpler but effective hybrid approach. The fingerprint control mechanism idea came from the need to quickly eliminate most positions in real-world texts.

For the pre-analysis system, we evaluated test results iteratively and continuously improved our strategy by learning which characteristics should make us prefer which algorithms.

2 Boyer-Moore Algorithm Implementation

2.1 Theoretical Background

The Boyer-Moore algorithm uses right-to-left comparison instead of the classical left-to-right approach. This method saves time by making larger jumps on mismatched characters. The algorithm's main advantage is that especially in large alphabets, many characters can be skipped. In the best case, it can reach O(n/m) complexity.

2.2 Our Bad Character Rule Approach

Boyer-Moore has two basic rules: Bad Character Rule and Good Suffix Rule. In our implementation, we chose the Bad Character Rule approach for three reasons:

Simplicity: Bad Character Rule has a simpler and more understandable structure compared to Good Suffix Rule.

Effectiveness: Especially in large alphabets, Bad Character Rule alone gives very effective results.

Implementation ease: Managing only one table (bad character table) is sufficient.

2.3 Implementation Details

We created a 256-sized array for the bad character table, covering the entire ASCII character set. For each character, we record its rightmost position in the pattern. We initialize all values to -1 to mark characters not in the pattern.

We use the `& 0xFF` operation because Java's char data type represents 16-bit Unicode characters, but our table has only 256 elements. This operation takes only the lower 8 bits of the character to fit it into the 0-255 range.

Our search logic works by comparing from right to left starting from the last character of the pattern. When we find a mismatched character, we determine the jump amount based on this character's last position in the pattern. We always guarantee at least 1 character jump to avoid infinite loops.

2.4 Edge Cases and Special Situations

Empty Pattern: An empty pattern matches at every position (standard string matching behavior).

Pattern longer than text: In this case, matching is impossible, we return an empty result.

Overlapping matches: When searching for "AAA" pattern in "AAAA" text, there should be matches at both position 0 and 1. In our first implementation, we were missing some matches by making large jumps. We solved this by making smart jumps based on the next character when a match is found.

2.5 Challenges We Faced

Character encoding problem: In our first attempts, we had array index inconsistency with Unicode characters. We solved this with the `& 0xFF` operator.

Reading after last character: When approaching the end of text, we were getting IndexOutOfBounds errors when trying to read the character after the pattern. We solved this by adding boundary checks.

3 GoCrazy Algorithm - Our Custom Algorithm

3.1 Development Motivation

While studying string matching algorithms, we noticed that Boyer-Moore is fast but has a quite complex structure. We wanted a simpler but effective method. We asked ourselves: "Can we quickly understand that a position has low matching probability before doing full comparison?"

In normal texts, characters are usually different. If we check a few important points of the pattern, we can eliminate most wrong positions very quickly. This thought led us to the fingerprint approach.

3.2 Hybrid Approach: Fingerprint + Horspool

GoCrazy algorithm consists of two main components:

1. 3-Point Fingerprint Check (Our Idea): We select 3 important positions from the pattern and check these first.

2. Safe Skip Table (Learned from Horspool): To jump safely when fingerprint doesn't match, we researched and learned Horspool algorithm's skip table.

While researching the Horspool algorithm, we saw that it uses a simpler version of the bad character table and makes safe jumps without missing any matches. We decided to combine this table with our fingerprint logic.

3.3 3-Point Control Strategy

When deciding how many points to select from the pattern, we did experimental tests. We found that 3 points is the optimal number that provides both fast filtering and low overhead.

The positions we select from the pattern are: first character, middle character, and last character. We made the decision to check the last character first because in Turkish and English words, last characters are very variable (suffixes like -ing, -lar, -de).

With the nested if structure, we immediately break the loop and jump at the first mismatched point. Most positions are eliminated at the first if statement.

3.4 Skip Table Implementation

We applied the skip table logic learned from the Horspool algorithm. The default skip is the full pattern length. For characters in the pattern, we set skip distances. An important point: we don't include the last character in the table. This is critical for Horspool's safety guarantee.

For Unicode characters outside ASCII, we jump directly by the pattern length. Since this situation is rare, it doesn't affect performance much.

3.5 Special Cases and Optimizations

If the pattern is 2 characters or shorter, the fingerprint method creates unnecessary overhead. Therefore, we use simple scanning for short patterns.

In 3-4 character patterns, selected points can overlap. We fix this by checking for duplicates and adjusting positions.

3.6 Performance Analysis

Best Case: O(n/m) - When fingerprint eliminates most positions and skip table provides large jumps, we can progress by dividing by pattern size.

Average Case: O(n) - In normal texts, fingerprint check works very fast. We do an average of 1-2 character checks per position.

Worst Case: O(n*m) - In repeating texts like "aaaaaaa" or small alphabets like DNA, fingerprints match frequently, which increases full checks.

3.7 Strengths and Weaknesses

Strong in:

- Normal language texts (English, Turkish sentences)
- Medium/long patterns (5+ characters)
- Large alphabet (many different characters)
- Very large texts (50,000+ characters)

Weak in:

- Repeating texts ("aaaaaaa")
- DNA sequences (only 4 letters, fingerprints match often)
- Very short patterns (1-2 characters, too much overhead)

4 Pre-Analysis Strategy

4.1 Purpose and Approach

The purpose of the pre-analysis system is to predict which algorithm will run fastest by looking at the given text and pattern characteristics. We observed from test results that each algorithm performs differently in different scenarios and created a decision tree using this knowledge.

4.2 Analysis Criteria

Basic criteria we use when making decisions:

Pattern Length (m): Simple algorithms are sufficient for short patterns, more sophisticated methods are needed for long patterns.

Text Length (n): In very large texts, preprocessing cost is amortized.

Repeating Structures: Repeating characters or periodic structures in the pattern make some algorithms stand out.

Alphabet Diversity: How many different characters are used? Small alphabet (like DNA) vs large alphabet (natural language).

4.3 Algorithm Selection Logic

First, we calculate the LPS table used by the KMP algorithm. This table gives us information about the repeating structure of the pattern. If the LPS value at the end of the pattern is large or the pattern contains periodic repetition within itself, we mark this pattern as "repeating."

We detect special pattern types: same character check, alternating pattern check, and DNA-like check.

Our strategy works in the following order: null and empty checks, single character, long pattern + long text, repeating structures, DNA sequences, very large text, small alphabet + long pattern, short pattern + small text, medium-long pattern, large text, and default naive.

4.4 Optimization Process

Our first version only did length checking and the success rate was around 40-50%. By iteratively analyzing test results, we improved our strategy step by step and reached 85-90% success rate in the final version.

4.5 Challenges

Overlapping criteria: Some patterns fell into multiple categories. It was difficult to decide which check should come first. We went from the most specific checks to the general.

Threshold values: We found boundaries like "how long is large text" or "how short is short pattern" through trial and error. After each change, we checked test results.

Preprocessing overhead: Pre-analysis itself takes time. We paid attention to using fast heuristics instead of doing very complex analyses.

5 Challenges and Solutions

5.1 Algorithm Development Process

Boyer-Moore Bad Character Table: In the first implementation, we didn't think that Unicode values of characters could be larger than 256. We got array index out of bounds errors. We solved it by adding the `& 0xFF` operator and ASCII check.

Overlapping Matches: When searching for "AAA" pattern in "AAAA" text, we were only finding the first match and making large jumps, missing the second match. We solved this by moving forward 1 character when a match is found.

GoCrazy Fingerprint Collision: In very short patterns (3-4 characters), the 3 selected points were falling on the same position. We added duplicate control.

5.2 Pre-Analysis Optimization

First Version Failure: In our first version where we only made length-based decisions, the success rate was around 40-50%. We analyzed which algorithm won in which situations by examining test results.

Understanding LPS Table: Understanding KMP's failure function took time. We did research to learn how to detect repetitions in the pattern and discovered that the last value of the LPS table gives us this information.

Determining Threshold Values: We found decisions like "should large text be 1000 or 10000" through trial and error. After each change, we ran all tests again and noted the results.

5.3 Debug and Test Process

Wrong Result Detection: In some test cases, we were returning wrong indices. We found the error by manually going step by step through small examples.

Performance Bottleneck: Initially, pre-analysis was too complex and we were analyzing the entire text every time. We accelerated it by doing sample-based analysis (first 300 characters).

Edge Case Deficiencies: We didn't think about situations like empty string, single character, pattern ¿ text at first. Test errors reminded us of these situations.

6 Test Results and Evaluation

6.1 Boyer-Moore Performance

Our Boyer-Moore implementation showed good performance especially in medium-long patterns (10+ characters) and large texts. We succeeded in most of the shared test cases.

It was strong in long patterns, large alphabet texts, and situations where the pattern appears rarely in the text. It was weak in very short patterns, repeating patterns, and small alphabets.

6.2 GoCrazy Performance

The GoCrazy algorithm showed the fastest or very close to fastest performance in many situations in test results. The fingerprint mechanism was very useful especially in large texts.

It was successful in very large texts (50000+ characters), normal language texts, and medium-length patterns (5-20 characters). It struggled with DNA-like small alphabets, repeating characters, and very short patterns.

6.3 Pre-Analysis Success Rate

In our final version, pre-analysis was able to select the fastest or very close to fastest algorithm in approximately 85-90% of test cases.

It made good predictions in situations with clear characteristics, extreme cases, and special pattern structures. It made mistakes in borderline cases, patterns with hybrid characteristics, and surprise performance situations.

6.4 General Evaluation

All three algorithms (Boyer-Moore, GoCrazy, Pre-Analysis) passed all shared test cases with correct results. In terms of performance, each algorithm has different strengths.

The GoCrazy algorithm turned out to be a good choice for general-purpose use. The pre-analysis system increased average performance by making the right choice.

7 Lessons Learned and Conclusion

7.1 Technical Learning

Algorithm Design: Designing an algorithm from scratch is much harder than implementing existing algorithms. Trade-offs need to be balanced.

Preprocessing Cost: Some algorithms do preprocessing. This cost is amortized if the text is long, but creates overhead if it's short.

Character Comparison Count: This is the most important metric. The jumps in Boyer-Moore and GoCrazy significantly reduce the total number of comparisons.

Unicode vs ASCII: You need to be careful when working with char in Java. We experienced array index errors.

7.2 Problem Solving Skills

Iterative Development: Nothing came out perfect the first time. The test, analyze, improve cycle was very important.

Debug Techniques: Starting with small examples and progressing step by step manually was very helpful in finding errors.

Documentation Reading: We had to look at academic sources to learn the Horspool algorithm. Understanding others' work was instructive.

7.3 Algorithm Analysis Experience

Knowing the best/average/worst cases of each algorithm in theory is one thing, seeing it in practice is another. Test results sometimes surprise you.

While doing pre-analysis, we saw the difference between theoretical complexity and real performance. Factors like constant factors and cache locality are also important.

7.4 Conclusion

This assignment helped us understand string matching algorithms not only theoretically but also practically. The experience of developing our own algorithm was especially instructive.

The GoCrazy algorithm may not be a work that enters academic literature, but it taught us the nuances of algorithm design. The pre-analysis system showed the necessity of different approaches in different situations.

Most importantly, we experienced that there is more than one solution to a problem and each solution performs differently in different scenarios.

8 Our Journey - Personal Reflections

This project was both challenging and rewarding. The pre-analysis part was particularly difficult because we had to understand not just how each algorithm works, but when it works best. We spent many hours analyzing test results, adjusting threshold values, and trying to figure out why certain algorithms performed better in specific cases.

Developing GoCrazy was exciting but frustrating at times. Our first attempts had bugs we couldn't understand for days. The fingerprint idea seemed simple, but making

it work correctly with overlapping matches took a lot of debugging. When we finally saw it working faster than other algorithms in large texts, it felt really good.

The hardest part was probably the iterative optimization of pre-analysis. Every time we thought we had a good strategy, we would find test cases where it failed. We learned that algorithm selection is not just about pattern length or text size - there are so many factors that matter. The breakthrough came when we started analyzing pattern repetitions using the LPS table.

We also learned a lot about teamwork and patience. Sometimes one of us would find a bug the other missed. Sometimes we would disagree on the best approach and had to test both ideas to see which worked better.

Overall, this homework taught us more than just algorithms. It taught us how to approach complex problems, how to learn from failures, and how to keep improving until we get good results.

9 References

Horspool Algorithm: We researched the Horspool algorithm to learn the skip table logic for the GoCrazy algorithm. The safe jump guarantee of this algorithm was important for us.

Boyer-Moore Algorithm: We used various online sources and algorithm visualization tools to understand bad character rule and good suffix rule concepts.

KMP Failure Function: We learned how KMP uses the LPS table to detect pattern repetitions for pre-analysis.

Java String Documentation: We referred to Java's official documentation on character encoding, Unicode handling, and string manipulation.

Test and Debug: We used print statements, small test cases, and manual trace methods.