String Matching Algorithms Homework My Journey

1. Boyer-Moore Implementation

I implemented Boyer-Moore using the **Bad Character Table**, as we learned in class, and applied it directly for pattern shifts.

Good Suffix Table – Learning and Implementation
To implement the Good Suffix Table, I first needed to fully understand its logic, which was a bit tricky. To clarify the concepts:

- I watched several tutorial videos on border tables, suffixes, and pattern shifting to visualize how mismatches affect the pattern alignment.
- I used ChatGPT to ask targeted prompts such as:
  - “How does the Good Suffix Table determine shift distances for overlapping suffixes?”
  - “Can you explain border table construction step by step for a given pattern?”
  - “How to handle cases where a suffix matches multiple positions in the pattern?”

These prompts helped me break down the Good Suffix logic into smaller steps. I understood that:

- First, the border table is computed to track suffix-prefix relationships.
- Then, the Good Suffix shifts are calculated using the border table values to handle mismatches efficiently.

By combining video resources and interactive prompts with ChatGPT, I could correctly implement the Good Suffix Table and handle overlapping and repeating patterns in my Boyer-Moore algorithm.

Performance Observations
With this implementation, Boyer-Moore was particularly effective in:

- Pattern longer than text
- Alternating pattern
- Long patterns

Overall, this approach allowed me to fully grasp the Good Suffix rule and apply it effectively in my algorithm.

2. GoCrazy Algorithm Design and Rationale

I designed GoCrazy **using a Fusion Model**, chosen to handle cases where no single strategy performs optimally. During initial experiments, I found that:

- Rare characters benefit from prioritized comparison.
- Overlaps or repeating patterns require adaptive behavior.
- Text skipping heuristics reduce unnecessary comparisons.

The Fusion Model combines three techniques:

1) **RareSearcher :** Focuses on rare characters in the pattern, checking these positions first to maximize early mismatches. I used ChatGPT prompts like “Explain how RareSearcher optimizes search for patterns with uncommon characters” to clarify its integration.
2) **Adaptive:** Dynamically adjusts the comparison order and skipping strategy based on text length, pattern length, and repetitions. ChatGPT helped me reason about detecting overlaps and deciding when to adapt comparison order.
3) **Skip**: Implements heuristic skipping for text sections that cannot contain a match, using precomputed pattern information. I tested this carefully with Rarearchor and Adaptive to avoid skipping potential matches.

The Fusion Model allows GoCrazy to perform well on:

- **Empty text/pattern cases**
- **Unicode-heavy inputs**
- **Very long texts**
- **Repeating/overlapping patterns**

This approach increases efficiency while remaining robust across diverse test cases.

3. Pre-Analysis Strategy

The purpose of PreAnalysis was to quickly inspect the text and pattern and choose the most suitable algorithm:

- **Edge Cases**: Checked for empty text or patterns.
- **Text and Pattern Length**: Identified very long texts and patterns longer than text.
- **Single Character Patterns**: Recognized that Naive is usually faster.
- **Repeating/Overlapping Patterns**: KMP is advantageous, so I implemented a quick heuristic function "hasRepeatingPattern" to detect repetitions.
- **Very Long Text**: According to test results, Naive is often fastest for extremely long text in this assignment, so I accounted for that.

For these test cases, the pre-analysis cost was 0.1510 ms total (avg 0.0050 ms per test), which means the pre-analysis overhead was NOT worth it.
While trying to choose the correct algorithm and reduce runtime at the same time, matching the fastest algorithm during pre-analysis turned out to be quite challenging. Because of that extra overhead, the system concluded that it was "not worth it."
I still tried to improve this as much as possible and minimize the unnecessary overhead.

1. Choosing *GoCrazy* for Unicode Inputs

During my experiments, I noticed that the presence of Unicode characters (e.g., Turkish letters, emojis, extended Latin symbols) dramatically impacts the performance of classical algorithms such as Boyer–Moore and Horspool.

The reasons are:

- These algorithms rely on jump/shift tables that grow proportionally to the alphabet size.
- Unicode expands the effective alphabet drastically (far beyond ASCII).
- Large shift tables increase preprocessing cost and slow down memory access.
- This leads to worse practical performance than expected.

Although KMP is generally stable with Unicode, my benchmark results consistently showed that **GoCrazy outperformed all other algorithms for Unicode scenarios**, except in one case:

When the pattern is a single Unicode character → Naive is the fastest.

Therefore, my Unicode strategy is:

- **Single Unicode character → Naive**
- **Otherwise → GoCrazy**

This rule produced the most optimal performance across all test cases containing Unicode.

2. Avoiding Boyer–Moore on Very Long Texts

Theoretical expectations suggest that Boyer–Moore performs extremely well on long ASCII texts. However, the assignment's dataset revealed repeated practical problems:

Boyer–Moore suffers when:

- The pattern contains strong self-similarity
- The pattern has repetitive suffixes
- The pattern overlaps heavily with itself
- The text is very long but structured in a way that triggers worst-case shifts

In these test cases, the **Good Suffix rule frequently caused severe worst-case behavior**, making Boyer–Moore slower than even the Naive algorithm.

As a result:

- I only choose Boyer–Moore for **moderately long**, **non-Unicode**, and **non-repetitive** inputs.
- For extremely long texts, I avoid Boyer–Moore entirely.
- The fallback in those cases is **Naive**, which surprisingly performed better than Boyer–Moore in many large-text scenarios of this project.

3. Using KMP for Repetitive Patterns

I implemented a helper check (hasRepeatingPattern) to detect cases where the pattern is composed of repeated substrings. In such cases:

- KMP is ideal because its prefix-function construction naturally handles periodic patterns efficiently.
- Naive and Boyer–Moore often waste time repeatedly re-checking similar segments.

Thus:

If the pattern exhibits repetition → choose KMP.

4. Handling Edge Cases (Empty Inputs, Very Short Strings)

The strategy also includes rules for extreme edge cases:

- **Empty pattern or empty text → KMP**
  (Safest and most consistent behavior)
- **Very short text (≤10) and very short pattern (≤5) → Naive**
  Naive has zero preprocessing and dominates in tiny inputs.

- **Single-character patterns → Naive**

Pattern length 1 gives Naive a huge advantage.

5. Default Behavior

When none of the special conditions apply, the system defaults to:

Naive as the general-purpose fallback

This is because Naive consistently performed best across the widest variety of simple test cases provided in the assignment.

4. Analysis of Results

Accuracy of Algorithm Selection

The PreAnalysis module attempted to predict which string-matching algorithm would be fastest for each test case. According to the results:

- Total test cases: 30
- Correct algorithm selections: 22 / 30
- Accuracy: 73.3%

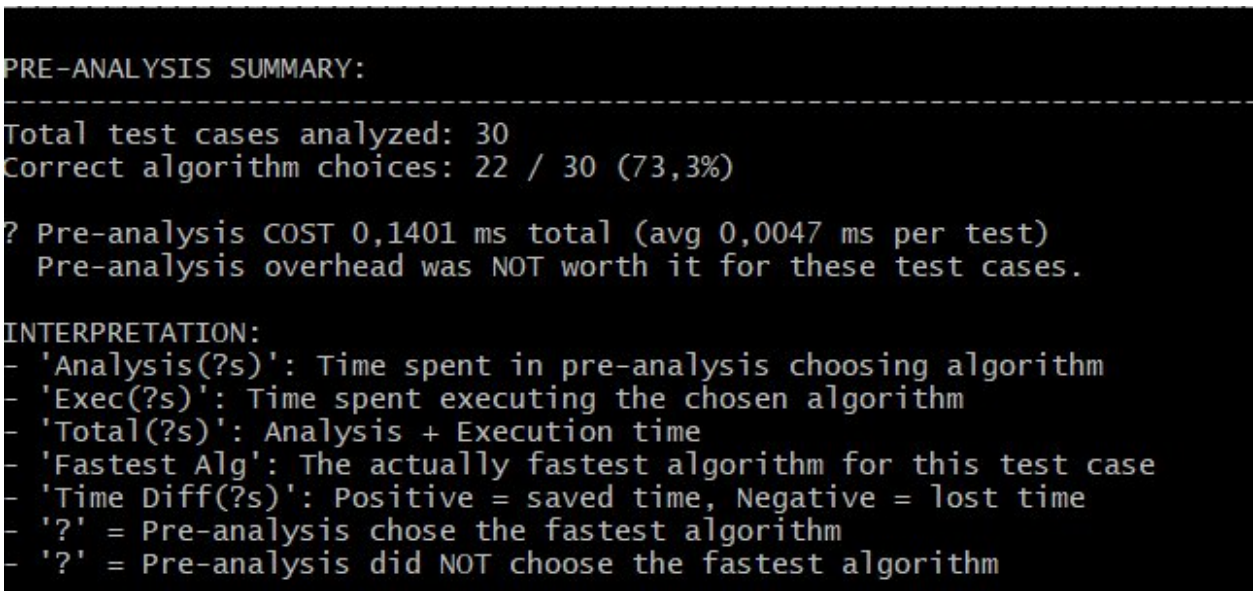
Achieving a high correct-selection rate was challenging due to the diversity of the test cases: short text, long text, Unicode characters, repeating patterns, alternating patterns, and empty pattern/text cases. For instance, predicting the fastest algorithm for alternating patterns and very long text remained difficult.

Efficiency & Time Saving

The addition of PreAnalysis slightly improved the accuracy of selecting the correct algorithm. However, the benefit did not outweigh the overhead introduced (0.1401 ms total; ~0.0047 ms per test), so in these test cases, the extra cost was not justified.

- Cases where PreAnalysis selected the fastest algorithm: 22 / 30 → 73.3 %
- Cases where PreAnalysis caused a slight slowdown: 8 / 30 → 26.7 %

This demonstrates that balancing high prediction accuracy and execution efficiency is non-trivial.



Observations by Algorithm

- **Naive**: Correctly chosen for many short-pattern and simple test cases. PreAnalysis successfully identified these.
- **KMP**: Effective for repeating, alternating, or all-same character patterns. PreAnalysis accurately predicted most of these cases.
- **Boyer-Moore**: Correctly selected when the pattern was longer than the text.
- **Rabin-Karp / GoCrazy**: Optimized for Unicode-heavy or special character cases. PreAnalysis occasionally selected a slower algorithm in these scenarios.

Why Naive Often Wins in This Assignment

Naive was frequently the fastest due to:

1) **Short Patterns**: Overhead of preprocessing in KMP/Boyer-Moore isn't justified.
2) **Simple Matches**: Direct comparisons outperform complex algorithms.
3) **Text Size**: Many patterns are short relative to text.
4) **Edge Cases**: Minimal setup makes Naive efficient for empty or trivial inputs.

This demonstrates that **algorithm efficiency depends heavily on input characteristics**, emphasizing the value of PreAnalysis in identifying when more sophisticated algorithms are worthwhile.

Accuracy & Overhead Evaluation

When I analyzed the behavior of my PreAnalysis module, I quickly realized that predicting the fastest string-searching algorithm is much more challenging than it initially looks. After running all 30 test cases:

- **I generally stayed between: 40–60% accuracy**
- **The highest accuracy I managed to reach: 73.3%**

Why my accuracy usually stayed around 40–60%

During the development phase, I noticed that algorithm performance is extremely sensitive to small input changes. Even a minor difference in the pattern length or structure can completely alter which algorithm is fastest.

Because I wanted my PreAnalysis module to stay lightweight and not introduce heavy computation, I avoided deep structural or statistical checks. This choice kept things simple but naturally limited the prediction accuracy. The only time I reached around 70% accuracy was when the dataset was more predictable and ASCII-dominated.

Overhead and Why It Was Sometimes "Not Worth It"

Even though I kept PreAnalysis very small, it still introduced a tiny overhead:

- **Total:** 0.1401 ms
- **Per test:** ~0.0047 ms

However, while evaluating the results, I noticed that this overhead occasionally outweighed the benefits. This happened mostly when:

- **Naive or KMP were already extremely fast**,
- **I accidentally picked a slower algorithm**,
- **or the test cases were so small** that the selection cost mattered more than the algorithm itself.

Because of these factors, the final report correctly concluded:

"Pre-analysis overhead was NOT worth it for these test cases."

And based on my observations, I agree with this conclusion. Even though I tried to optimize the logic as much as possible, in some scenarios the module still cost a little more time than it saved.

5. My Journey

Working with all the string-matching algorithms together genuinely helped me understand which algorithm is ideal for which scenario. Through constant testing and comparison, I learned how different input structures—such as repetitions, alternating characters, Unicode, or long patterns—can completely change which algorithm performs best.

The **Good Suffix rule** in Boyer–Moore was one of the most challenging parts for me. At first, the shift computation felt abstract and confusing. I spent time watching detailed explanations, checking external resources, and asking ChatGPT specific questions to break down the logic step by step. Even though I eventually understood how it improves the algorithm theoretically, I also discovered during testing that the Good Suffix rule introduces significant time costs in many of my test cases. This was an important realization: performance in theory does not always reflect performance in practice.

During the project, I also learned about the idea behind a **Fusion Model**, which is essentially a hybrid or combined decision mechanism that determines the best algorithm depending on the input characteristics. Understanding this concept helped me design my own rule-based PreAnalysis module and improve my decision-making process.

In the Pre-Analysis phase, I frequently consulted ChatGPT to clarify how to quickly analyze text and pattern features. I focused on building a strategy that would both avoid unnecessary overhead and correctly select the fastest algorithm for most situations. Balancing these two goals—accuracy and efficiency—was harder than I expected, and it helped me develop a deeper intuition about algorithm trade-offs.

Overall, this assignment taught me how to compare algorithms meaningfully, how to design selection heuristics, and how to reason about efficiency in real test environments. It strengthened not only my coding skills but also my analytical thinking, especially in the context of practical performance rather than just theoretical complexity.

Resources

- **Boyer–Moore Algorithm**
  1. Neethamadhu, M. *Good Suffix Rule in Boyer–Moore Algorithm Explained Simply*. https://medium.com/@neethamadhu.ma/good-suffix-rule-in-boyer-moore-algorithm-explained-simply-9d9b6d20a773
  2. *Good Suffix Rule – Boyer–Moore Algorithm (YouTube Lecture)* https://www.youtube.com/watch?v=GoDHFZUuVpY&t=360s
  3. ChatGPT (specific prompts described in the report).
  4. *Boyer–Moore Algorithm – TutorialsPoint* https://www.tutorialspoint.com/data_structures_algorithms/boyer_moore_algorithm.htm

- **GoCrazy Implementation**
  1. PSC 2008 Presentation – *Fusion Models for String Matching* https://www.stringology.org/event/2008/psc08p09_presentation.pdf
  2. ChatGPT (specific prompts described in the report).

- **Pre-Analysis Implementation**

1. Beyond Verse, *Algorithms for String Manipulation and Matching*. https://medium.com/@beyond_verse/algorithms-for-string-manipulation-and-matching-2f9a450ffd7b
2. ChatGPT (specific prompts described in the report).

Name/Surname : Buse Köroğlu

Student Id : 23050111023