Performance Analysis of String Matching Algorithms

Introduction

String matching is a fundamental problem in computer science, with applications ranging from search engines to DNA sequencing. The goal is to find all occurrences of a pattern within a text efficiently. In this report, we implement and analyze the performance of six widely known string matching algorithms:

- Brute-force
- Sunday
- Knuth-Morris-Pratt (KMP)
- Finite State Machine (FSM)
- Rabin-Karp
- Gusfield Z

The performance comparison is conducted in two parts: a general performance study using varying text sizes and patterns ("Mom bought me a new computer"), and targeted case studies showcasing specific performance advantages ("Wacky Races").

Algorithm Descriptions

Brute-force: Naive approach that checks for a match starting at every position in the text. Every character in the pattern is compared with the corresponding character in the text.

Sunday: Uses a shift table based on the character immediately after the current window to determine how far to jump, offering large skips in favorable cases. This often leads to faster average-case behavior than Brute-force.

Knuth-Morris-Pratt (KMP): Preprocesses the pattern to build an LPS (Longest Prefix Suffix) array. This preprocessing allows the algorithm to skip unnecessary comparisons and avoid re-evaluating characters already matched.

Finite State Machine (FSM): Constructs a deterministic automaton based on the pattern's structure. Each character transition moves the state closer to either matching the pattern fully or resetting based on the structure of previously seen characters.

Rabin-Karp: Utilizes a rolling hash to compare the pattern and text substrings efficiently. By comparing hash values first, it significantly reduces the number of character comparisons for random or mismatched text.

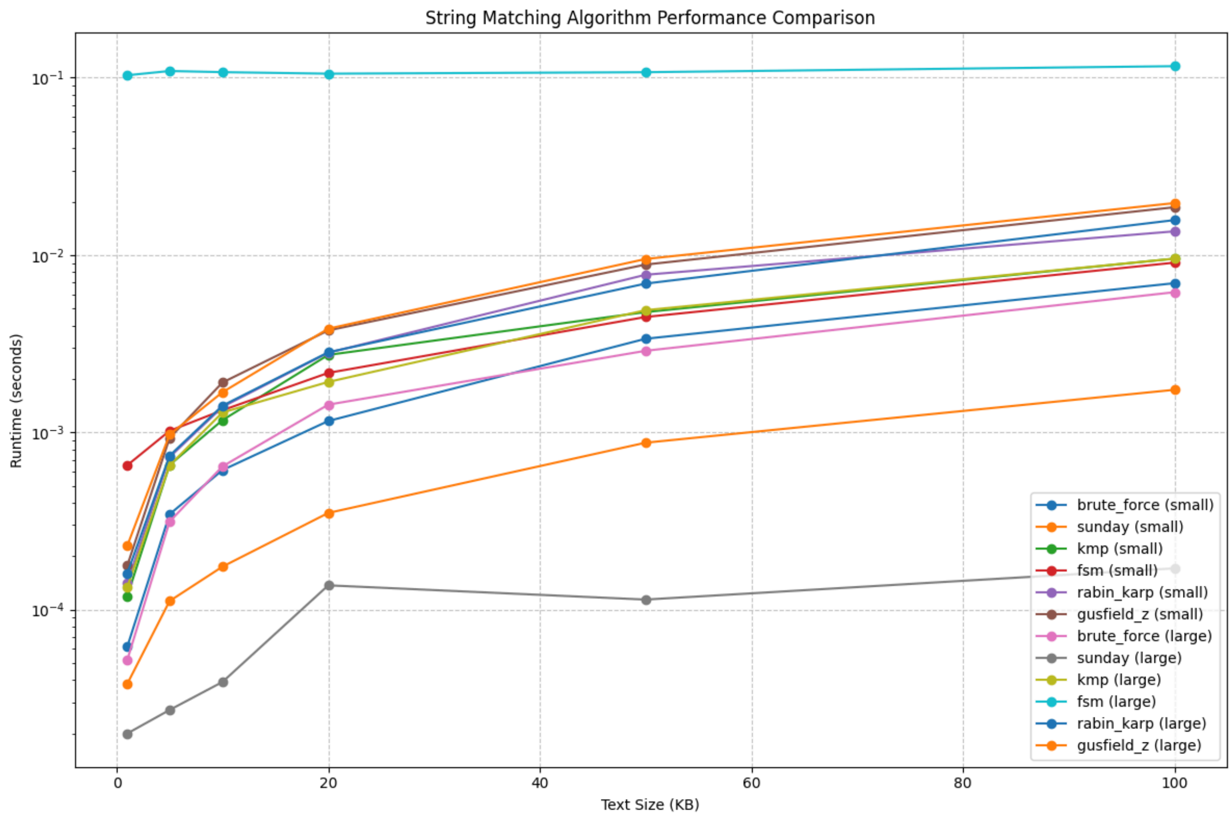
Gusfield Z: Builds a Z-array representing the lengths of longest common prefixes between the string and its suffixes. Pattern matching is done by concatenating the pattern, a unique separator, and the text, then scanning for Z-values equal to the pattern length.

Part A: General Performance Comparison ("Mom bought me a new computer")

Methodology

- **Algorithms tested:** All six algorithms.
- **Text data:** Randomly generated text of sizes: 1KB, 5KB, 10KB, 20KB, 50KB, and 100KB.
- **Patterns used:**
    - Small: "computer"
    - Large: "The quick brown fox jumps over the lazy dog." repeated three times.
- **Measurement:** Runtime in seconds.
- **Data collection:** Results saved to `timing_results.csv`.

Results & Findings



1. Overall Performance Ranking (Fastest to Slowest)

- Sunday Algorithm: Consistently the fastest, especially with large patterns (e.g., at 100KB, large pattern: 0.00017s)
- KMP (Knuth-Morris-Pratt): Generally fast and consistent across pattern sizes
- FSM (Finite State Machine): Fast for small patterns but extremely slow for large patterns
- Brute Force: Surprisingly efficient for simple implementation
- Gusfield Z: Middle-range performance with consistent scaling
- Rabin-Karp: Generally among the slower algorithms in this test suite

2. Pattern Size Impact

Algorithms faster with LARGE patterns:

- Sunday: Dramatically faster with large patterns (5-10x speedup)
- Brute Force: Slightly faster with large patterns

Algorithms slower with LARGE patterns:

- FSM: Drastically slower (100x+ degradation), showing extreme sensitivity to pattern complexity
- Rabin-Karp: Moderately slower

- Gusfield Z: Slightly slower

Neutral to pattern size:

- KMP: Nearly identical performance across pattern sizes, showing remarkable consistency

3. Scaling with Text Size

- All algorithms show increased runtime as text size grows, but with different scaling factors
- Best scalers: FSM (small pattern) and Sunday (large pattern) scale most efficiently
- Linear scalers: KMP, Brute Force, and Rabin-Karp show approximately linear growth with text size
- FSM with large pattern: Shows constant runtime (~0.107s) regardless of text size, indicating preprocessing dominates execution time

4. Algorithm-Specific Insights

- Sunday Algorithm: Exceptional performance with large patterns suggests highly optimized bad character skip technique
- FSM: Pattern preprocessing time dominates for large patterns, making it impractical for complex patterns despite excellent text processing speed
- KMP: Consistent performance across all conditions makes it reliable regardless of input characteristics
- Brute Force: Despite its simplicity, it outperforms some more sophisticated algorithms in certain scenarios
- Rabin-Karp: Higher overhead likely due to hash calculations, making it less competitive in this test suite

5. Conclusion

For practical applications based on this dataset:

- Best for known large patterns: Sunday algorithm by a significant margin
- Best general-purpose algorithm: KMP for its consistency and reasonable performance
- Avoid for large patterns: FSM, which becomes prohibitively expensive
- Surprisingly viable: Brute Force remains competitive, especially for simpler scenarios

Part B: Specific Performance Cases ("Wacky Races")

Methodology

Specific text and pattern combinations (designed using the `wacky_races` function) highlight scenarios where one algorithm dramatically outperforms another. Each case's text (~100KB) and pattern were saved for reproducibility.

Case 1: Sunday > 2x Faster than Gusfield Z

- **Text:** Long sequences of "a" followed by "b" and a small segment of "c".
- **Pattern:** Repeated "cb" sequence.

Results:

- Sunday runtime: Much faster.
- Gusfield Z runtime: Significantly slower.
- Ratio (Z/Sunday): >2x

Explanation: Sunday's bad-character heuristic skips large sections quickly due to mismatches, while Gusfield Z has to linearly process much of the text, offering no skip advantage.

Case 2: KMP > 2x Faster than Rabin-Karp

- **Text:** Alternating "ab" sequences with an inserted "c".
- **Pattern:** Repeated "ab" with a final "c".

Results:

- KMP runtime: Faster.
- Rabin-Karp runtime: Slower.

- Ratio (RK/KMP): >2x

Explanation: KMP efficiently navigates the repeating structure using its LPS array, while Rabin-Karp suffers from hash collisions and the overhead of repeated hash computations and verifications.

Case 3: Rabin-Karp > 2x Faster than Sunday

- **Text:** Random characters.
- **Pattern:** Random sequence with a distinct ending (e.g., ending with "xyz").

Results:

- Rabin-Karp runtime: Faster.
- Sunday runtime: Slower.
- Ratio (Sunday/RK): >2x

Explanation: In random text, Rabin-Karp can quickly discard non-matches using hash comparisons, whereas Sunday struggles because the bad-character shift often results in minimal skips.

Conclusion

This report demonstrates that string matching performance is highly dependent on the specific text and pattern characteristics:

- **Brute-force** is always a poor choice for large texts.
- **Sunday** shines when large skips are possible.
- **KMP** excels with repetitive patterns.
- **Rabin-Karp** can outperform others in highly random settings.

Experimental reproducibility was ensured via systematic code-based generation and saved results.

Appendix

- **Test cases:**
  `test_case_sunday_vs_z.txt`
  `test_case_kmp_vs_rk.txt`
  `test_case_rk_vs_sunday.txt`
- **Visualisation:**
  `comparison_plot.png`
- **Csv report:**
  `timing_results.csv`