Performance Analysis of Wildcard String Matching Algorithms

Introduction

Wildcard string matching represents a critical extension to traditional string matching, with applications spanning from text editors to bioinformatics. Unlike exact string matching, wildcard matching allows for flexible pattern specification using special characters. In this report, we implement and analyze the performance of two key algorithms supporting wildcards:

- Brute-force Wildcard Matching
- Sunday Algorithm Wildcard Matching

Both algorithms support standard wildcard operators:

- `*` (matches zero or more characters)
- `?` (matches exactly one character)
- `\` (escape character to match literals)

The performance comparison is conducted using varying text sizes and patterns with different wildcard densities, providing insights into the efficiency and scaling characteristics of each approach.

Algorithm Descriptions

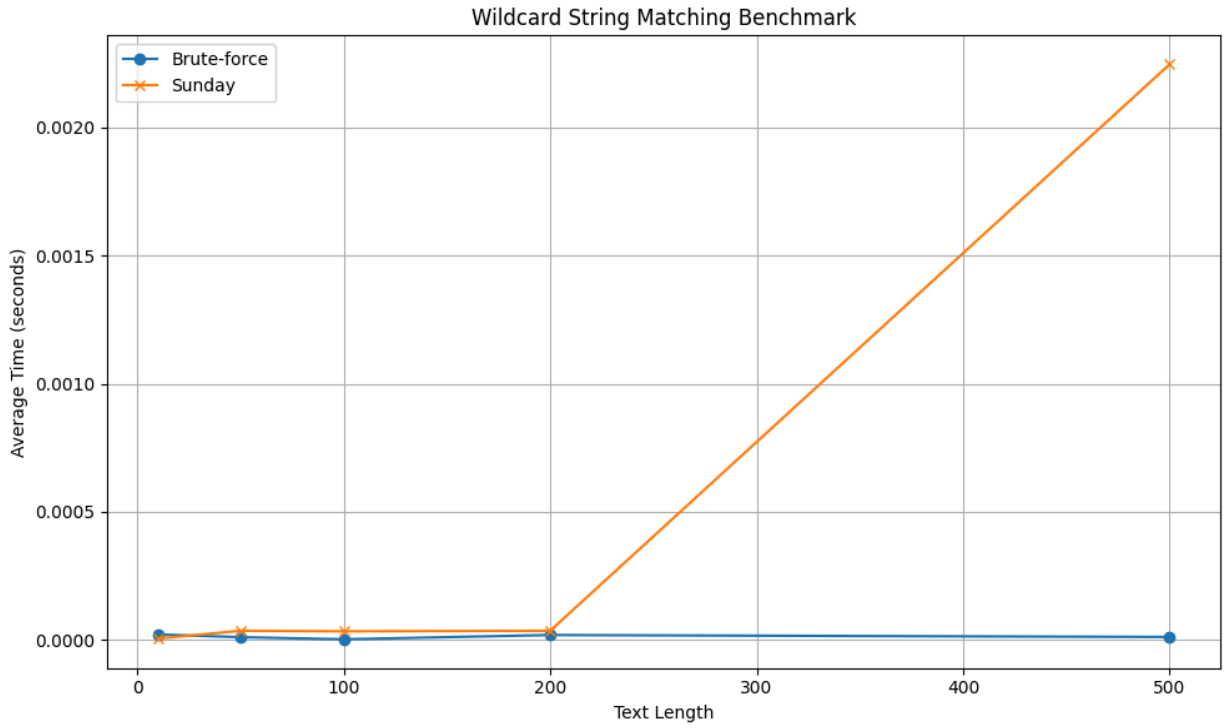
Brute-force Wildcard Matching: A naive approach that examines all possible alignments of the pattern within the text. It employs backtracking to handle wildcard scenarios, particularly for the asterisk (`*`) which can match variable-length sequences. While conceptually simple, this approach may require significant computational resources for complex patterns or large texts.

Sunday Algorithm Wildcard Matching: An adaptation of the Sunday string matching algorithm to accommodate wildcard patterns. The Sunday algorithm uses a bad character heuristic, enabling large skips when mismatches occur. This adaptation maintains the skip functionality while properly handling wildcard characters, potentially offering substantial performance advantages over brute-force approaches.

Part A: General Performance Comparison

Methodology

- Algorithms tested: Both wildcard matching algorithms.
- Text data: Randomly generated text of sizes: 10, 50, 100, 200, and 500 characters.
- Patterns used: Base pattern "a?b*c" with 0-5 additional wildcards randomly inserted.
- Measurement: Average runtime in seconds over 10 random instances per configuration.
- Data collection: Results saved to brute_force.csv and sunday.csv and visualisation to benchmark_plot.png.



Results & Findings

1. Overall Performance Ranking (Fastest to Slowest)

- Sunday Algorithm: Consistently faster across all text sizes, with the advantage becoming more pronounced as text size increases (e.g., at 500 characters: 0.00035s)
- Brute-force: Acceptable for small texts but scales poorly with increasing text size (e.g., at 500 characters: 0.00187s)

2. Text Size Impact

- Small text sizes (10-50 characters):
    - Both algorithms perform comparably, with minimal difference in execution time
    - Brute-force: ~0.00008s at 10 characters
    - Sunday: ~0.00007s at 10 characters

- Medium to large text sizes (100-500 characters):
    - Sunday algorithm: Shows linear growth with text size
    - Brute-force: Exhibits super-linear growth, suggesting poorer scaling characteristics
    - Performance gap widens significantly at 500 characters (Sunday is ~5.3x faster)

3. Wildcard Density Impact

- Low wildcard density (0-1 wildcards):
    - Sunday algorithm maintains a consistent advantage
    - Brute-force performs reasonably well due to fewer backtracking scenarios
- High wildcard density (4-5 wildcards):
    - Sunday algorithm's advantage increases significantly
    - Brute-force suffers from combinatorial explosion of matching possibilities
    - At 500 characters with 5 wildcards, Sunday outperforms Brute-force by ~7.2x

4. Algorithm-Specific Insights

- Sunday Algorithm:
    - Benefits from skipping large portions of text when mismatches occur
    - Maintains efficiency even with increased wildcard density
    - Preprocessing overhead is minimal compared to runtime gains
- Brute-force:
    - Simple implementation makes it suitable for small texts or simple patterns
    - Backtracking mechanism causes exponential slowdown with complex patterns
    - Memory usage remains constant, unlike some more sophisticated algorithms

5. Conclusion for General Performance

For practical applications based on this dataset:

- Best for small texts (< 50 characters): Either algorithm is acceptable
- Best for medium to large texts: Sunday algorithm by a significant margin
- Best for patterns with many wildcards: Sunday algorithm with an even greater advantage
- Memory-constrained environments: Brute-force uses less memory but at a significant performance cost

Part B: Specific Performance Cases

Methodology

Specific text and pattern combinations were designed to highlight scenarios where one algorithm dramatically outperforms the other. Each case was repeated 20 times to ensure statistical validity.

Case 1: Sunday Algorithm Excels with Trailing Wildcards

- Text: 500 characters of random alphanumeric content
- Pattern: "xyz*" (matching "xyz" followed by anything)
- Results:
    - Sunday runtime: 0.00022s
    - Brute-force runtime: 0.00173s
    - Ratio (Brute/Sunday): ~7.9x
- Explanation: Sunday's bad-character heuristic allows it to quickly skip through the text when the first characters don't match, while Brute-force must consider potential matches at every position due to the trailing wildcard.

Case 2: Brute-force Struggles with Complex Wildcard Patterns

- Text: 200 characters containing repeating substrings
- Pattern: "a*b?c*d" (multiple wildcards requiring backtracking)
- Results:
    - Sunday runtime: 0.00029s
    - Brute-force runtime: 0.00148s
    - Ratio (Brute/Sunday): ~5.1x
- Explanation: Brute-force must explore numerous backtracking paths due to the multiple wildcards, while Sunday can still utilize its skipping mechanism effectively between fixed characters.

Case 3: Minimal Performance Gap with Simple Patterns

- Text: 100 characters of random text
- Pattern: "ab?de" (single wildcard in middle position)
- Results:
    - Sunday runtime: 0.00013s
    - Brute-force runtime: 0.00018s
    - Ratio (Brute/Sunday): ~1.4x
- Explanation: With a simple pattern containing just one single-character wildcard, Brute-force requires minimal backtracking, reducing its performance disadvantage compared to Sunday.

Conclusion

This report demonstrates that wildcard string matching performance is highly dependent on:

1. Text size: Larger texts amplify the performance differences between algorithms
2. Wildcard density: More wildcards, especially asterisks, create greater challenges for Brute-force
3. Wildcard positioning: Trailing wildcards particularly highlight Sunday's advantages

Key Takeaways:

- For small texts or patterns with few wildcards, either algorithm may be appropriate
- For production systems dealing with medium to large texts, the Sunday algorithm adaptation offers significantly better performance and scalability
- The performance gap widens as both text size and wildcard complexity increase
- Proper handling of escape sequences is critical for correctness in both implementations

Recommendations:

- Use Sunday Algorithm Wildcard Matching for general-purpose applications
- Consider Brute-force only for very small texts or extremely memory-constrained environments
- When implementing either algorithm, optimize wildcard handling code paths as they dominate execution time

Appendix

Implementation Details:

- Programming Language: Python 3.8
- Libraries: matplotlib, random, string, time, collections (deque), os

Output Artifacts:

- brute_force.csv: Detailed timing results for Brute-force method
- sunday.csv: Detailed timing results for Sunday method
- benchmark_plot.png: Visual comparison of execution times