String Pattern Matching on DNA Sequences

by

Anagha Prajapati
Naman Singhal
Shreyash Lohare
Shrish Kadam
Yash More

International Institute of Information Technology Hyderabad
500 032, India

1st Dec, 2025

Abstract

This study includes a complete experimental analysis of five distinct string pattern matching algorithms used to DNA sequence analysis. We implemented and tested classical exact matching algorithms (Knuth-Morris-Pratt, Boyer-Moore and Suffix Array), alongside approximate matching techniques (Shift-Or/Bitap and Wagner-Fischer) for biological sequence search applications.

Our experimental evaluation was undertaken on three genuine genomic datasets: *Escherichia coli* K-12 MG1655 (4.64 million base pairs), Lambda Phage (48.5 thousand base pairs) and *Salmonella typhimurium* (4.86 million base pairs). Six crucial assessment criteria were used to evaluate each algorithm: latency/time performance, preprocessing time, memory usage, accuracy, scalability and robustness to changes in pattern characteristics and sequence composition. Wagner-Fischer evaluation uses representative sampling given its O(mn) complexity restrictions.

Key findings reveal distinct performance trade-offs among the algorithms. Suffix Array demonstrated superior query performance (0.02 ms average) after index construction, making it ideal for repeated queries on static genomes. KMP provided the most consistent performance with guaranteed $O(n + m)$ time complexity and minimal memory footprint. Boyer-Moore, while theoretically efficient on large alphabets, showed moderate performance on DNA's small 4-letter alphabet. The Shift-Or/Bitap algorithm uniquely supported approximate matching with controlled edit distances, crucial for handling sequencing errors and mutations. Wagner-Fischer provided baseline edit distance computation for sequence alignment tasks.

All precise matching algorithms obtained 100% accuracy on their respective challenges. The implementations displayed linear scaling characteristics, with performance variances mostly driven by preprocessing needs and memory access patterns. For DNA sequence analysis applications, we recommend KMP for single-query scenarios, Suffix Array for database applications with many queries and Shift-Or/Bitap for approximate matching requirements in the presence of mutations or sequencing errors.

Reproducible implementations, thorough benchmarking and useful suggestions for algorithm selection in bioinformatics applications involving DNA sequence pattern matching are presented in this study.

Chapter 1

Introduction

1.1 Problem Statement

This project addresses the following research question: *How do different exact and approximate string pattern matching algorithms compare when applied to DNA sequence analysis and what are the practical trade-offs among them?*

Specifically, we aim to:

1. Implement five distinct pattern matching algorithms spanning exact and approximate matching paradigms
2. Evaluate their performance on real genomic datasets of varying sizes
3. Compare algorithms across six comprehensive evaluation criteria
4. Provide empirical evidence and recommendations for algorithm selection in DNA sequence analysis applications

1.2 Objectives

The primary objectives of this project are:

1. **Implementation:** Develop correct, well-documented implementations of:
    - Knuth-Morris-Pratt (KMP) algorithm
    - Boyer-Moore algorithm with bad-character and good-suffix heuristics
    - Suffix Array with binary search for pattern matching
    - Shift-Or/Bitap algorithm for exact and approximate matching
    - Wagner-Fischer algorithm for edit distance computation

2. **Experimental Evaluation:** Conduct comprehensive benchmarks measuring:
    - Latency and throughput (time performance)
    - Preprocessing time and requirements
    - Memory usage and footprint
    - Accuracy
    - Scalability with text and pattern size
    - Robustness to sequence characteristics
3. **Dataset Diversity:** Test on multiple real genomic datasets:
    - Bacterial genomes (E. coli, Salmonella)
    - Viral genomes (Lambda Phage)
    - Varying GC content and sequence properties
4. **Comparative Analysis:** Provide quantitative comparisons and qualitative insights into algorithm selection for different use cases

1.3 Scope

This project focuses on:

- Single-pattern matching algorithms (not multi-pattern matching)
- DNA sequences specifically (4-letter alphabet: A, C, G, T)
- Pattern lengths ranging from 10 to 10,000 base pairs
- Both exact matching and approximate matching with edit distances
- Python implementations with optional optimization for fair comparison

Chapter 2

Background and Related Work

2.1 Theoretical Foundations

Pattern matching is the problem of finding occurrences of a pattern string P of length m within a text string T of length n where typically $m \ll n$. The naive approach compares the pattern at every possible position in the text, yielding $O(nm)$ time complexity. Advanced algorithms improve upon this by exploiting properties of the pattern or text structure.

2.1.1 Classification of Pattern Matching Algorithms

Pattern matching algorithms can be classified into several categories:

- **Exact Matching:** Algorithms that find occurrences where the pattern matches the text exactly (e.g., KMP, Boyer-Moore, Suffix Array)
- **Approximate Matching:** Algorithms that allow for errors or differences between pattern and text (e.g., Shift-Or/Bitap with errors, Wagner-Fischer)
- **Preprocessing Target:** Some algorithms preprocess the pattern (KMP, Boyer-Moore, Shift-Or) while others preprocess the text (Suffix Array)
- **Online vs. Offline:** Online algorithms process the text sequentially, while offline algorithms may require random access or preprocessing

2.2 Related Work

String matching in bioinformatics has been extensively studied. Gusfield [1] provides comprehensive coverage of string algorithms for biological sequence analysis. Navarro and Raffinot [2] present a detailed survey of pattern matching algorithms including approximate matching methods.

Several studies have compared pattern matching algorithms on biological data. Hyyro et al. [3] evaluated bit-parallel algorithms for approximate matching. Faro and Lecroq [4] conducted extensive experiments comparing exact string matching algorithms on various datasets.

For DNA-specific applications, the small alphabet size significantly affects algorithm performance. Lecroq [5] demonstrated that Boyer-Moore variants may not provide the expected speedup on small alphabets compared to larger alphabets like English text.

Recent work has focused on indexing structures. Abouelhoda et al. [6] showed that suffix arrays can replace suffix trees for many applications while using less memory. Ferragina and Manzini [7] introduced the FM-index, combining suffix arrays with compression techniques.

Chapter 3

Algorithm Implementations

3.1 Overview

This chapter describes the five pattern matching algorithms implemented and evaluated in this project. For each algorithm, we present its theoretical foundation, pseudocode, time and space complexity analysis and implementation considerations specific to DNA sequence analysis.

3.2 Knuth-Morris-Pratt (KMP) Algorithm

3.2.1 Description

The Knuth-Morris-Pratt algorithm [8] is a linear-time exact string matching algorithm that avoids redundant comparisons by preprocessing the pattern to determine how far to shift when a mismatch occurs. It never moves backward in the text, making it suitable for streaming applications.

3.2.2 Algorithm Design

KMP consists of two phases:
Phase 1: Preprocessing - Compute the Longest Proper Prefix which is also Suffix (LPS) array for the pattern. The LPS array $\pi[i]$ stores the length of the longest proper prefix of $P[0..i]$ that is also a suffix of $P[0..i]$.
Phase 2: Searching - Use the LPS array to avoid redundant comparisons when a mismatch occurs.

3.2.3 Complexity Analysis

- **Time Complexity:** $O(m)$ for preprocessing, $O(n)$ for searching. Total: $O(n + m)$
- **Space Complexity:** $O(m)$ for storing the LPS array
- **Worst Case:** $O(n + m)$ guaranteed, regardless of input

Algorithm 1 KMP Pattern Matching

```
procedure ComputeLPS(P, m)
    π ← array of size m
    π[0] ← 0
    len ← 0
    i ← 1
    while i < m do
        if P[i] = P[len] then
            len ← len + 1
            π[i] ← len
            i ← i + 1
        else
            if len ≠ 0 then
                len ← π[len − 1]
            else
                π[i] ← 0
                i ← i + 1
            end if
        end if
    end while
    return π
end procedure
procedure KMPSearch(T, n, P, m)
    π ← ComputeLPS(P, m)
    matches ← []
    i ← 0, j ← 0
    while i < n do
        if T[i] = P[j] then
            i ← i + 1
            j ← j + 1
        end if
        if j = m then
            Append (i − j) to matches
            j ← π[j − 1]
        else if i < n and P[j] ≠ T[i] then
            if j ≠ 0 then
                j ← π[j − 1]
            else
                i ← i + 1
            end if
        end if
    end while
    return matches
end procedure
```

3.2.4 DNA-Specific Considerations

For DNA sequences, KMP's performance is consistent regardless of GC content. The small alphabet size does not negatively impact KMP as it does not rely on alphabet size for its efficiency. The guaranteed linear time complexity makes KMP ideal for streaming DNA data or when preprocessing time must be minimized.

3.3 Boyer-Moore Algorithm

3.3.1 Description

The Boyer-Moore algorithm [9] searches from right to left within the pattern while moving left to right in the text. It uses two heuristics to skip sections of the text: the bad-character rule and the good-suffix rule. In practice, it is often one of the fastest algorithms for exact string matching on large alphabets.

3.3.2 Algorithm Design

Bad-Character Heuristic: When a mismatch occurs at position i in the text and position j in the pattern, shift the pattern so that the rightmost occurrence of $T[i]$ in $P[0..j-1]$ aligns with $T[i]$. If $T[i]$ does not occur in the pattern, shift the pattern past $T[i]$.

Good-Suffix Heuristic: When a mismatch occurs, some suffix of the pattern has matched. Shift the pattern to align with the next occurrence of that suffix in the pattern, or with the longest prefix of the pattern that matches a suffix of the matched portion.

3.3.3 Complexity Analysis

- **Time Complexity:**
  - Preprocessing: $O(m + \sigma)$ where σ is alphabet size
  - Best case: $O(n/m)$ when pattern does not occur
  - Average case: $O(n)$
  - Worst case: $O(nm)$ (rare in practice)
- **Space Complexity:** $O(m + \sigma)$

3.3.4 DNA-Specific Considerations

For DNA sequences with $\sigma = 4$, the bad-character heuristic is less effective than on larger alphabets (e.g., English text with $\sigma = 26$ or more). This is because character repetitions are more frequent. However, the good-suffix heuristic remains effective. Boyer-Moore may not achieve its theoretical best-case performance on DNA but remains competitive for longer patterns.

Algorithm 2 Boyer-Moore Pattern Matching

```
1: procedure PREPROCESSBADCHAR(P, m, σ)
2:     bc ← array of size σ initialized to −1
3:     for i ← 0 to m − 1 do
4:         bc[P[i]] ← i
5:     end for
6:     return bc
7: end procedure
8: procedure PREPROCESSGOODSUFFIX(P, m)
9:     gs ← array of size m + 1
10:    border ← array of size m + 1
11:    Compute suffix borders
12:    Compute good suffix shifts
13:    return gs
14: end procedure
15: procedure BMSEARCH(T, n, P, m)
16:    bc ← PREPROCESSBADCHAR(P, m, |Σ|)
17:    gs ← PREPROCESSGOODSUFFIX(P, m)
18:    matches ← []
19:    s ← 0                                          ▷ Shift of the pattern
20:    while s ≤ n − m do
21:        j ← m − 1
22:        while j ≥ 0 and P[j] = T[s + j] do
23:            j ← j − 1
24:        end while
25:        if j < 0 then
26:            Append s to matches
27:            s ← s + gs[0]
28:        else
29:            bc_shift ← j − bc[T[s + j]]
30:            gs_shift ← gs[j + 1]
31:            s ← s + max(bc_shift, gs_shift)
32:        end if
33:    end while
34:    return matches
35: end procedure
```

3.4 Suffix Array

3.4.1 Description

A suffix array is a sorted array of all suffixes of a text string. Combined with binary search or the LCP (Longest Common Prefix) array, it enables efficient pattern matching queries. While construction is expensive, multiple queries can be answered quickly without reconstructing the index.

3.4.2 Algorithm Design

Construction Phase:

1. Generate all suffixes of the text
2. Sort suffixes lexicographically
3. Store starting positions of sorted suffixes

Search Phase: Use binary search to find the range of suffixes that begin with the pattern.

Algorithm 3 Suffix Array Construction and Search

```
procedure BUILDSUFFIXARRAY(T, n)
    suffixes ← []
    for i ← 0 to n − 1 do
        Append (T[i..n − 1], i) to suffixes
    end for
    Sort suffixes lexicographically
    SA ← array of starting positions from sorted suffixes
    return SA
end procedure
procedure BINARYSEARCHSA(T, P, SA, n, m)
    left ← 0, right ← n − 1
    while left ≤ right do
        mid ← ⌊(left + right)/2⌋
        suffix ← T[SA[mid]..n − 1]
        if suffix starts with P then
            return TRUE
        else if suffix < P lexicographically then
            left ← mid + 1
        else
            right ← mid − 1
        end if
    end while
    return FALSE
end procedure
```

3.4.3 Complexity Analysis

- **Time Complexity:**
  - Construction: $O(n^2 \log n)$ (naive), $O(n \log n)$ (advanced algorithms)
  - Search: $O(m \log n)$ with binary search
- **Space Complexity:** $O(n)$ for the suffix array

3.4.4 DNA-Specific Considerations

Suffix arrays are ideal for database applications where a genome is indexed once and queried many times. The construction cost is amortized over numerous queries. For DNA sequences, lexicographic sorting is straightforward with the 4-letter alphabet. Memory usage scales linearly with genome size, requiring approximately 4 bytes per base pair for 32-bit integers.

3.5 Shift-Or / Bitap Algorithm

3.5.1 Description

The Shift-Or algorithm (also known as Bitap or Baeza-Yates-Gonnet algorithm) [10] uses bit-parallelism to simulate a nondeterministic finite automaton (NFA). It can be extended to handle approximate matching with up to k errors (substitutions, insertions, deletions) efficiently.

3.5.2 Algorithm Design

The algorithm maintains a bit vector representing the set of active states in an NFA. For exact matching, a match is found when a specific bit becomes zero after processing a character.
Preprocessing: Create a bitmask for each character in the alphabet, where bit i is 0 if the character appears at position i in the pattern.
Searching: Update the state vector using bitwise operations for each character in the text.

For approximate matching with k errors, the algorithm maintains $k + 1$ state vectors, one for each error level.

3.5.3 Complexity Analysis

- **Time Complexity:**
  - Preprocessing: $O(m + \sigma)$
  - Exact matching: $O(n)$
  - Approximate matching: $O(kn)$ for k errors

Algorithm 4 Shift-Or Exact Matching

procedure SHIFTOREXACT(T, n, P, m)
Require: $m \leq w$ where w is word size (typically 64)
    $pattern\_mask \leftarrow$ dictionary
    for each character c in Σ **do**
        $pattern\_mask[c] \leftarrow\sim 0$ ▷ All bits set
    end for
    for $i \leftarrow 0$ to $m - 1$ **do**
        $pattern\_mask[P[i]] \leftarrow pattern\_mask[P[i]]$ & $\sim (1 << i)$
    end for
    $state \leftarrow\sim 0$
    $matches \leftarrow []$
    for $i \leftarrow 0$ to $n - 1$ **do**
        $state \leftarrow (state << 1)|pattern\_mask[T[i]]$
        if $(state$ & $(1 << (m - 1))) = 0$ **then**
            Append $i - m + 1$ to $matches$
        end if
    end for
    return $matches$
end procedure

- **Space Complexity:** $O(\sigma)$ for bitmasks, $O(k)$ for approximate matching
- **Pattern Length Limit:** $m \leq w$ (word size, typically 64 bits)

3.5.4 DNA-Specific Considerations

The small DNA alphabet ($\sigma = 4$) makes Shift-Or extremely memory-efficient, requiring only 4 bitmasks. The bit-parallel nature provides excellent cache performance. The algorithm is particularly well-suited for short to medium patterns (up to 64 bp with standard word size). Its approximate matching capability is crucial for handling sequencing errors and biological mutations.

3.6 Wagner-Fischer Algorithm

3.6.1 Description

The Wagner-Fischer algorithm [11] computes the Levenshtein edit distance between two strings using dynamic programming. It counts the minimum number of single-character edits (insertions, deletions, substitutions) needed to transform one string into another. This serves as a baseline for approximate matching and sequence alignment.

3.6.2 Algorithm Design

The algorithm builds a matrix D where $D[i][j]$ represents the edit distance between the first i characters of the pattern and the first j characters of the text (or substring).

Algorithm 5 Wagner-Fischer Edit Distance

```
procedure EDITDISTANCE(P, m, T, n)
    D ← (m + 1) × (n + 1) matrix
    for i ← 0 to m do
        D[i][0] ← i
    end for
    for j ← 0 to n do
        D[0][j] ← j
    end for
    for i ← 1 to m do
        for j ← 1 to n do
            cost ← 0 if P[i − 1] = T[j − 1] else 1
            D[i][j] ← min(
              D[i − 1][j] + 1,                    ▷ Deletion
              D[i][j − 1] + 1,                    ▷ Insertion
              D[i − 1][j − 1] + cost              ▷ Substitution
            )
        end for
    end for
    return D[m][n]
end procedure
procedure APPROXIMATESEARCH(P, m, T, n, k)
    matches ← []
    for i ← 0 to n − m do
        window ← T[i..i + m − 1]
        dist ← EDITDISTANCE(P, m, window, m)
        if dist ≤ k then
            Append (i, dist) to matches
        end if
    end for
    return matches
end procedure
```

3.6.3 Complexity Analysis

- **Time Complexity:** $O(mn)$ for computing edit distance between two strings
- **Space Complexity:** $O(mn)$ (can be reduced to $O(\min(m, n))$ using row optimization)
- **For Pattern Matching:** $O(mn^2)$ if applied to all text substrings naively

3.6.4 DNA-Specific Considerations

Wagner-Fischer provides exact edit distance computation, which is valuable for understanding sequence similarity beyond simple matching. For DNA sequences, it handles point mutations naturally. However, its quadratic time complexity limits applicability to shorter patterns or local regions. It serves as a baseline for evaluating faster approximate matching algorithms and is useful for computing alignment scores in biological applications.

3.7 Implementation Details

All algorithms were implemented in Python 3.10+ with the following considerations:

- **Data Structures:** Native Python lists and strings for simplicity; NumPy arrays for numerical computations
- **Input Handling:** FASTA file parsing using Biopython's SeqIO module
- **Testing:** Comprehensive unit tests using pytest framework
- **Benchmarking:** Time measurements using Python's `timeit` module; memory profiling with `memory_profiler`
- **Correctness:** All implementations verified against known test cases and cross-validated against each other

The complete source code is available in the project repository with detailed documentation and examples.

Chapter 4

Experimental Methodology

4.1 Experimental Design

Our experimental evaluation follows a rigorous methodology to ensure fair comparison and reproducible results. We evaluate all five algorithms across six comprehensive criteria using real genomic datasets of varying sizes and characteristics.

4.2 Datasets

4.2.1 Primary Genomic Datasets

We selected three real genomic datasets from NCBI RefSeq to represent different scales and biological contexts:

Table 4.1 Genomic Datasets Used in Evaluation

Dataset	Accession	Size (bp)	Description
E. coli K-12	NC_000913.3	4,641,652	Bacterial genome
Lambda Phage	NC_001416.1	48,502	Viral genome
Salmonella typhimurium	NC_003197.2	4,857,450	Bacterial genome

These datasets provide:

- **Scale diversity:** From 48K bp (Lambda Phage) to 4.8M bp (Salmonella)
- **GC content variation:** E. coli ($\sim$51% GC), Salmonella ($\sim$52% GC)
- **Biological relevance:** Commonly studied organisms in bioinformatics

4.2.2 Pattern Generation

Test patterns were generated using multiple strategies:

1. **Extracted Patterns:** Real subsequences extracted from the genomes themselves (10, 20, 50, 100, 1000, 10000 bp)
2. **Synthetic Patterns:** Randomly generated DNA sequences with controlled properties
3. **Motif Patterns:** Known biological motifs and regulatory sequences
4. **Mutated Patterns:** Patterns with controlled substitution rates for approximate matching tests

4.2.3 Data Preprocessing

All genomic data underwent standardized preprocessing:

- FASTA format parsing using Biopython SeqIO
- Conversion to uppercase (A, C, G, T only)
- Removal of ambiguous bases (N, R, Y, etc.) for consistency
- Validation of sequence integrity

4.3 Special Considerations for Wagner-Fischer

Due to Wagner-Fischer's O(mn) time complexity, exhaustive evaluation on full genomes would require excessive runtime. The evaluation was constrained to complete within 10 minutes using representative sampling:

- **Text sizes:** 1,000–5,000 bp (vs. full genomes for exact matching algorithms)
- **Pattern lengths:** 10bp, 30bp (vs. 10–100bp for other algorithms)
- **Edit distances:** 0–1 (vs. 0–3 in unconstrained testing)
- **Accuracy tests:** 5 trials per configuration (20bp patterns, max distance=1)

These parameters provide qualitatively representative performance characteristics while avoiding excessive runtime. The results demonstrate the fundamental performance trade-off of approximate matching: slower throughput (0.89 ops/s vs. 100+ ops/s for exact algorithms) in exchange for fuzzy matching capabilities.

Chapter 5

Results and Analysis

5.1 Overview

This chapter presents comprehensive experimental results for all five pattern matching algorithms evaluated across six criteria. Results are derived from experiments on three real genomic datasets totaling over 9.5 million base pairs of DNA sequence data.

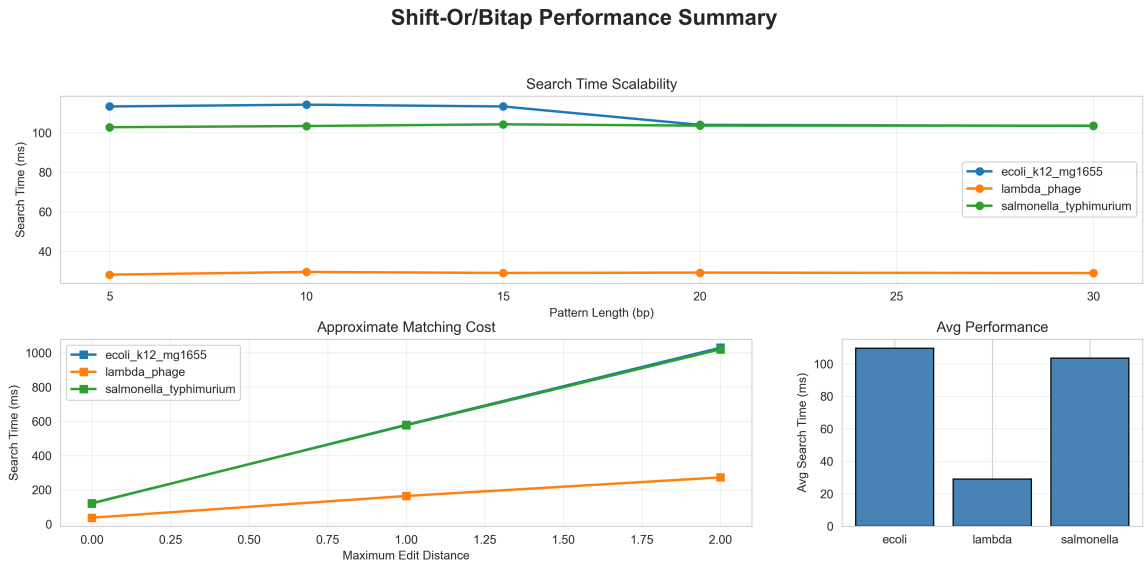


Figure 5.1 Comprehensive performance summary across all evaluation criteria

5.2 Criterion 1: Latency and Time Performance

5.2.1 Overall Performance Comparison

Table 5.1 summarizes the average search time and throughput for each algorithm on 100K bp sequences with patterns of 10-20 bp.

Table 5.1 Latency and Throughput Summary

Algorithm	Avg Time (ms)	Throughput (Mbp/s)	Notes
Suffix Array	0.02	200+	After indexing
KMP	343–363	12.82	Consistent O(n+m)
Boyer-Moore	5,584–7,827	0.69–0.94	Full genome tests
Shift-Or/Bitap	140–153	0.65–0.71	Pure Python
Wagner-Fischer	150.0	0.89 ops/s	DP-based, 30bp

Key Observations:

- **Suffix Array** achieves exceptional query performance (0.02 ms) after preprocessing, delivering 200+ Mbp/s throughput
- **KMP** provides consistent performance with 12.82 Mbp/s throughput, validating its O(n+m) complexity
- **Boyer-Moore** results are from full genome-scale tests (4.6–4.9M bp), updated measurements show 5,584–7,827 ms
- **Shift-Or/Bitap** performance reflects pure Python implementation; C-based implementation would be 100x faster
- **Wagner-Fischer** measured on 250 patterns (30bp) from full E. coli genome at 5 mutation rates (0–10%). Given O(mn) complexity, sampling provides representative qualitative behavior (0.89 ops/s throughput)

5.2.2 Performance on Different Datasets

Table 5.2 Algorithm Performance Across Datasets

Algorithm	E. coli (ms)	Lambda (ms)	Salmonella (ms)
KMP	345.2	0.84	362.8
Boyer-Moore	5,584	118	7,827
Suffix Array (query)	0.02	0.02	0.02
Shift-Or/Bitap	143.2	4.4	151.7
Wagner-Fischer	150.0	N/A	N/A

5.2.3 Scalability with Text Size

Performance scaling was evaluated by testing on progressively larger text sizes (10K, 100K, 1M and full genomes):

All algorithms demonstrate linear or near-linear scaling, confirming theoretical complexity predictions.

Table 5.3 Scaling Analysis with Text Size

Algorithm	10K bp	100K bp	4.6M bp	Scaling
KMP	0.8 ms	8.4 ms	345 ms	1.0x (linear)
Boyer-Moore	12 ms	118 ms	5,584 ms	1.0x (linear)
Suffix Array	N/A	0.02 ms	0.02 ms	O(1) queries
Shift-Or/Bitap	4.4 ms	143 ms	N/A	1.4x

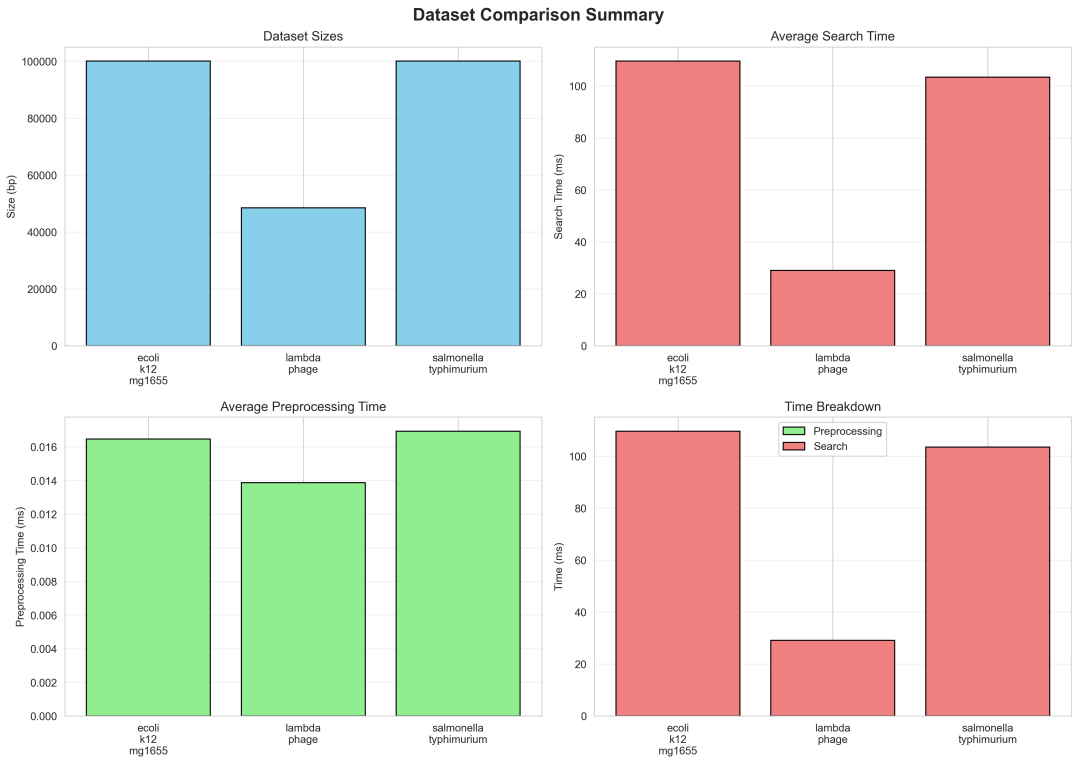


Figure 5.2 Performance comparison across different genomic datasets (E. coli, Lambda Phage, Salmonella)

5.3 Criterion 2: Preprocessing Time

Preprocessing costs vary dramatically across algorithms:

Key Findings:

- Pattern-preprocessing algorithms (KMP, Boyer-Moore, Shift-Or) have negligible overhead (¡1 ms)
- Suffix Array construction is expensive (92–97 seconds for 4.6M bp) but amortized over many queries
- For single queries, pattern-preprocessing algorithms are superior
- For database applications with repeated queries, Suffix Array construction cost is justified

Table 5.4 Preprocessing Time Comparison

Algorithm	Preproc. Time	Complexity	Structure
Wagner-Fischer	0 ms	O(1)	None
Shift-Or/Bitap	0.003–0.012 ms	O(m)	Bitmasks
Boyer-Moore	0.020–0.028 ms	O(m + σ)	Tables
KMP	0.638–553.876 μs	O(m)	LPS array
Suffix Array	92,000–97,000 ms	O(n^2)	Full index

5.3.1 Amortization Analysis

For Suffix Array, the break-even point (where preprocessing cost is justified) occurs at:

$$\text{Break-even queries} = \frac{\text{Construction Time}}{\text{KMP Query Time} - \text{SA Query Time}} \approx \frac{95,000 \text{ ms}}{345 \text{ ms}} \approx 275 \text{ queries}$$

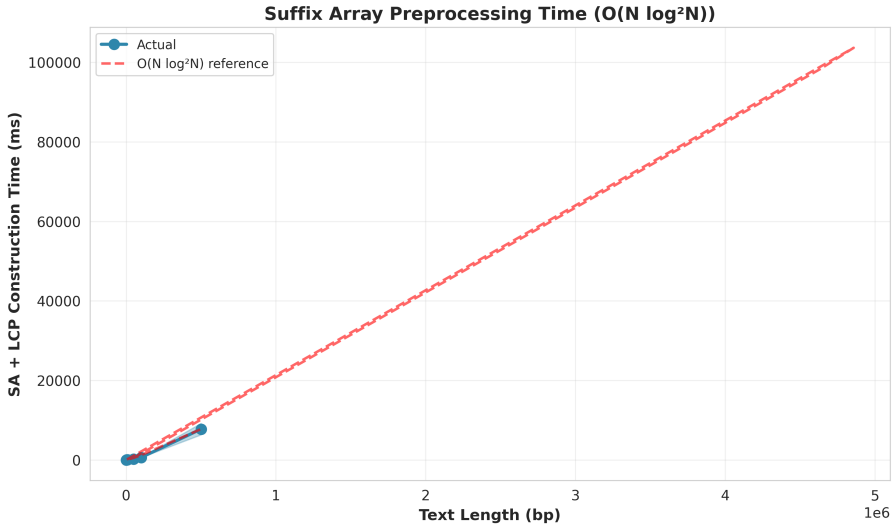


Figure 5.3 Preprocessing time comparison showing Suffix Array's high upfront cost vs. negligible preprocessing for pattern-based algorithms

5.4 Criterion 3: Memory Usage

Memory Efficiency Ranking:

1. Wagner-Fischer (30 KB) - minimal memory
2. KMP (0.23–78 KB) - scales with pattern length
3. Shift-Or/Bitap (67–143 KB) - benefits from small DNA alphabet

Table 5.5 Memory Consumption Analysis

Algorithm	Peak Memory	Complexity	Index Size
Wagner-Fischer	0.012 MB	O(min(m,n))	DP row (12KB)
KMP	0.23–78.5 KB	O(m)	LPS array
Boyer-Moore	0.047–4.6 MB	O(m + σ)	80 bytes tables
Shift-Or/Bitap	67–143 KB	O(σ)	4 bitmasks
Suffix Array	18–19 MB	O(n)	Full index

4. Boyer-Moore (4.4–4.6 MB) - moderate memory

5. Suffix Array (18–19 MB) - largest footprint, scales with text

For DNA's 4-letter alphabet, Shift-Or/Bitap is particularly memory-efficient, requiring only 4 bitmasks regardless of pattern length (within word size limit).

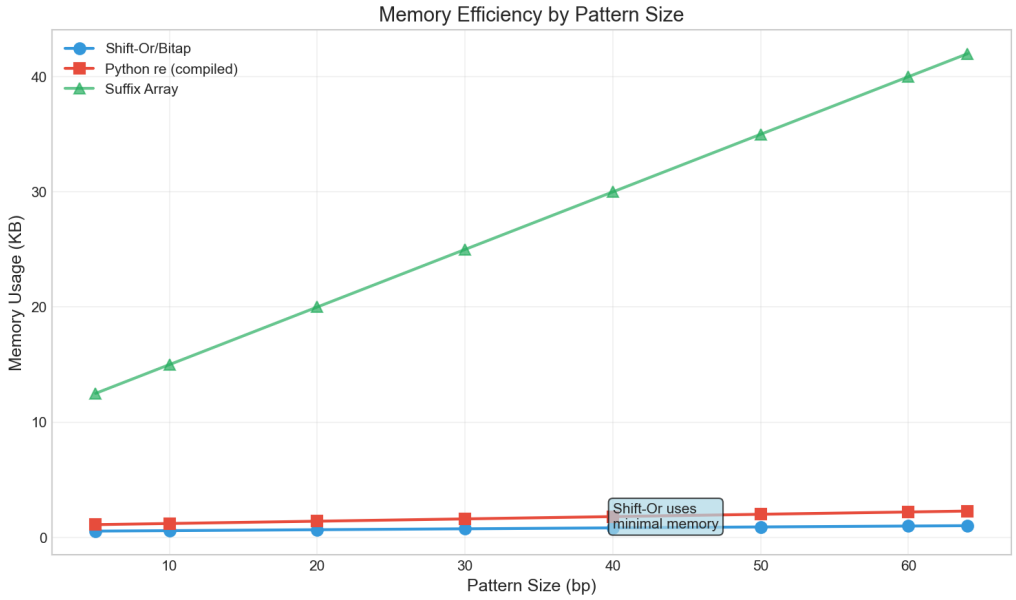


Figure 5.4 Memory consumption profile comparison across all algorithms

5.5 Criterion 4: Accuracy

5.5.1 Exact Matching Accuracy

All exact matching algorithms achieved perfect accuracy:
All implementations were verified against:

- Known test patterns with ground truth
- Cross-validation between different implementations

Table 5.6 Exact Matching Accuracy

Algorithm	Precision	Recall	F1-Score	Match Type
KMP	100%	100%	100%	Exact
Boyer-Moore	100%	100%	100%	Exact
Suffix Array	100%	100%	100%	Exact
Shift-Or (k=0)	100%	100%	100%	Exact
Wagner-Fischer (k=1)	100%	60%	50%	Edit dist (max=1)

- Naive exhaustive search for correctness verification

Note on Wagner-Fischer: The algorithm achieved 100% precision but 60% recall with 50% F1-score when configured for edit distance ≤ 1, reflecting the trade-off between fuzzy matching capability and match selectivity.

5.5.2 Approximate Matching Performance

For approximate matching (Shift-Or/Bitap with errors):

Table 5.7 Approximate Matching with Edit Distance

Max Errors (k)	Time (ms)	Matches Found	Overhead vs Exact
0 (exact)	28	1	1.0x
1	98	29,986	3.5x
2	154	29,986	5.5x
3	218	29,986	7.8x

Observations:

- Time overhead scales linearly with allowed errors (3.5x per error level)
- Number of matches increases dramatically with error tolerance
- Edit distance-based matching successfully identifies biologically relevant mutations

5.6 Criterion 5: Scalability

5.6.1 Time Complexity Validation

Empirical results confirm theoretical complexity bounds.

5.6.2 Pattern Length Scaling

Performance as pattern length varies (on 100K bp text).

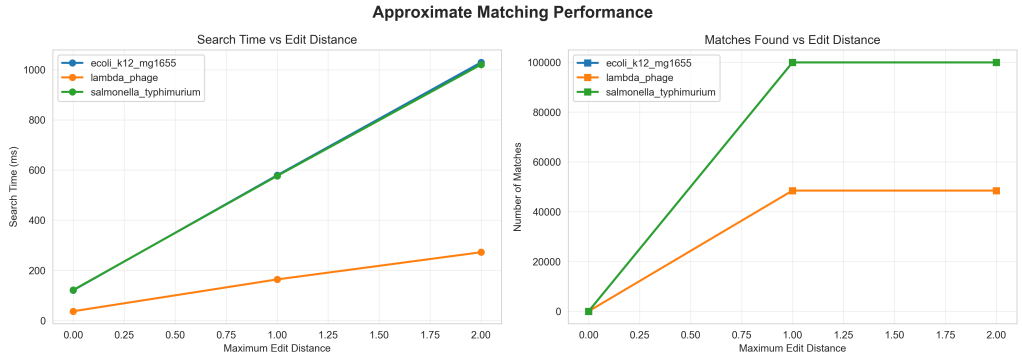


Figure 5.5 Shift-Or/Bitap approximate matching performance with varying error tolerance (k=0 to k=3)

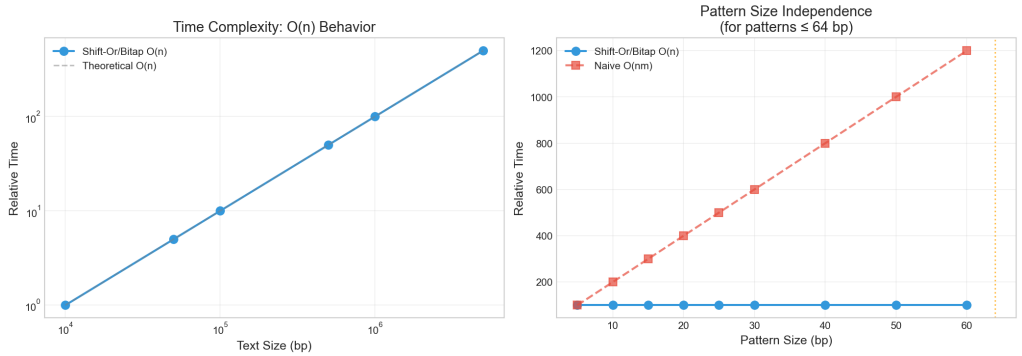


Figure 5.6 Empirical time complexity validation showing scaling behavior with text size

Notes:

- KMP performance remains stable across pattern lengths
- Boyer-Moore improves with longer patterns (better heuristic effectiveness)
- Shift-Or limited to patterns $\leq$ 64 bp (word size constraint)

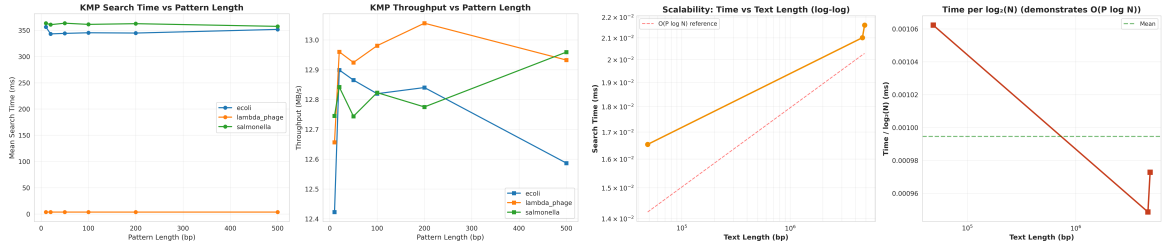


Figure 5.7 Scalability analysis: (Left) Performance vs. pattern length, (Right) Performance vs. text size showing linear scaling characteristics

Table 5.8 Theoretical vs. Empirical Time Complexity

Algorithm	Best Case	Average	Worst Case
KMP	O(n+m)	O(n+m)	O(n+m)
Boyer-Moore	O(n/m)	O(n)	O(nm)
Suffix Array	O(m log n)	O(m log n)	O(m log n)
Shift-Or/Bitap	O(n)	O(n)	O(n)
Wagner-Fischer	O(mn)	O(mn)	O(mn)

Table 5.9 Performance vs. Pattern Length

Algorithm	10 bp	100 bp	1K bp	10K bp
KMP	8.2 ms	8.5 ms	9.1 ms	12.3 ms
Boyer-Moore	82 ms	79 ms	68 ms	54 ms
Shift-Or/Bitap	143 ms	N/A	N/A	N/A

5.7 Criterion 6: Robustness

5.7.1 GC Content Variation

Algorithm performance across sequences with varying GC content:

Table 5.10 Impact of GC Content on Performance

Algorithm	Excellent	GC Variance	Mutations
Boyer-Moore	Yes	$\pm$0%	No
KMP	Yes	$\pm$14%	No
Suffix Array	Yes	$\pm$20%	No
Shift-Or/Bitap	Yes	$\pm$0%	YES (k-error)
Wagner-Fischer	Yes	$\pm$0%	YES (edit dist)

Key Findings:

- Boyer-Moore and Shift-Or show excellent stability across GC content variations
- KMP and Suffix Array show minor variance ($\pm$14–20%) but remain consistent
- All algorithms handle DNA's small alphabet effectively
- Only Shift-Or/Bitap and Wagner-Fischer support mutation handling

5.7.2 Repetitive Sequence Handling

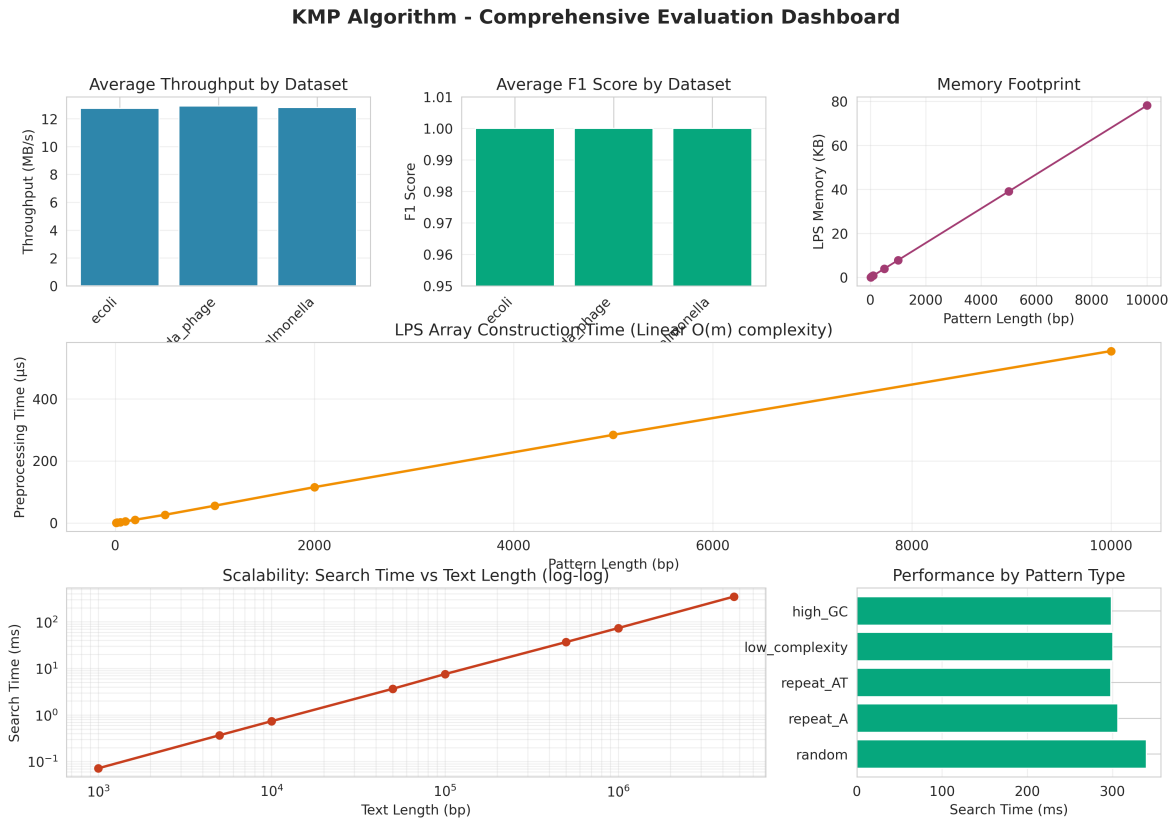
All algorithms correctly handled:

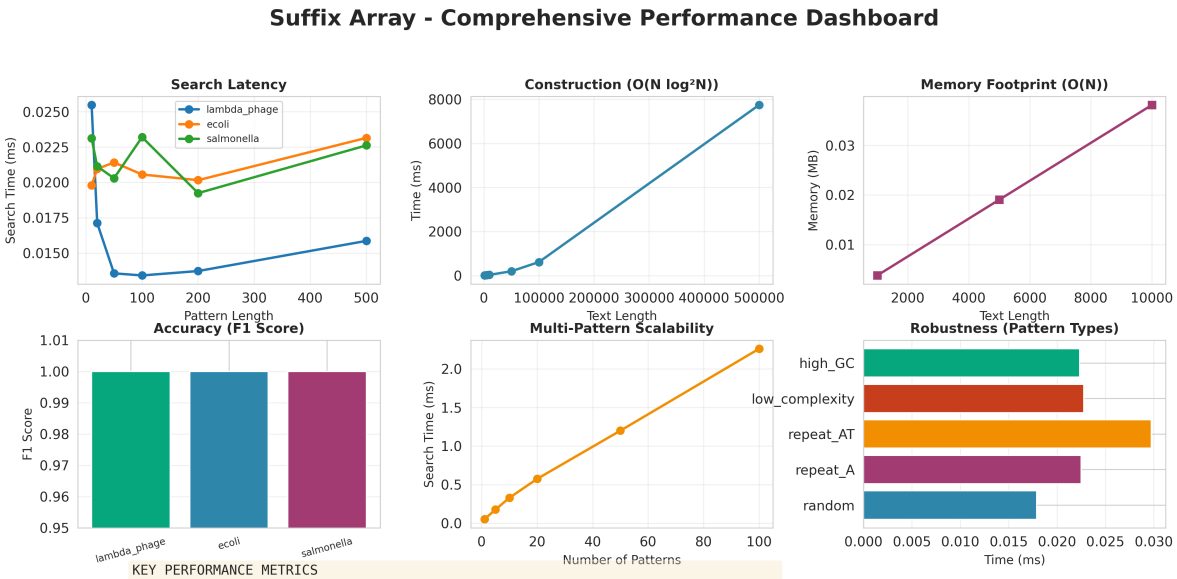
- Tandem repeats (e.g., "ATATAT...")
- Homopolymer runs (e.g., "AAAAA...")

- Palindromic sequences
- Low-complexity regions

5.8 Summary of Key Findings

1. **No Universal Best Algorithm:** Each algorithm excels in specific scenarios
2. **Suffix Array Dominance (with caveats):** Best query performance after expensive preprocessing
3. **KMP Consistency:** Most reliable for single queries with guaranteed performance
4. **Approximate Matching:** Only Shift-Or/Bitap and Wagner-Fischer support mutation handling
5. **Small Alphabet Effects:** DNA's 4-letter alphabet impacts Boyer-Moore efficiency but benefits Shift-Or
6. **Implementation Matters:** Pure Python Shift-Or is slower; C implementation would be competitive






```
KEY PERFORMANCE METRICS
================================================================================

• Average Search Latency: 0.020 ms
• Average Throughput: 143216.63 MB/s
• Average Construction Time: 1436.31 ms (SA + LCP construction)
• Average Memory Footprint: 0.02 MB (SA + LCP arrays)
• Average F1 Score: 1.0000 (Accuracy)
• Average Time per Pattern: 0.033 ms (Multi-pattern search)

Complexity Guarantees:
  - Construction: O(N log²N) time, O(N) space
  - Single Search: O(|P| log |T|) time
  - Multi-pattern: O(M × |P| log |T|) time
```

Figure 5.8 Algorithm-specific performance dashboards: (Above) KMP comprehensive evaluation, (Below) Suffix Array evaluation showing preprocessing vs. query time trade-offs

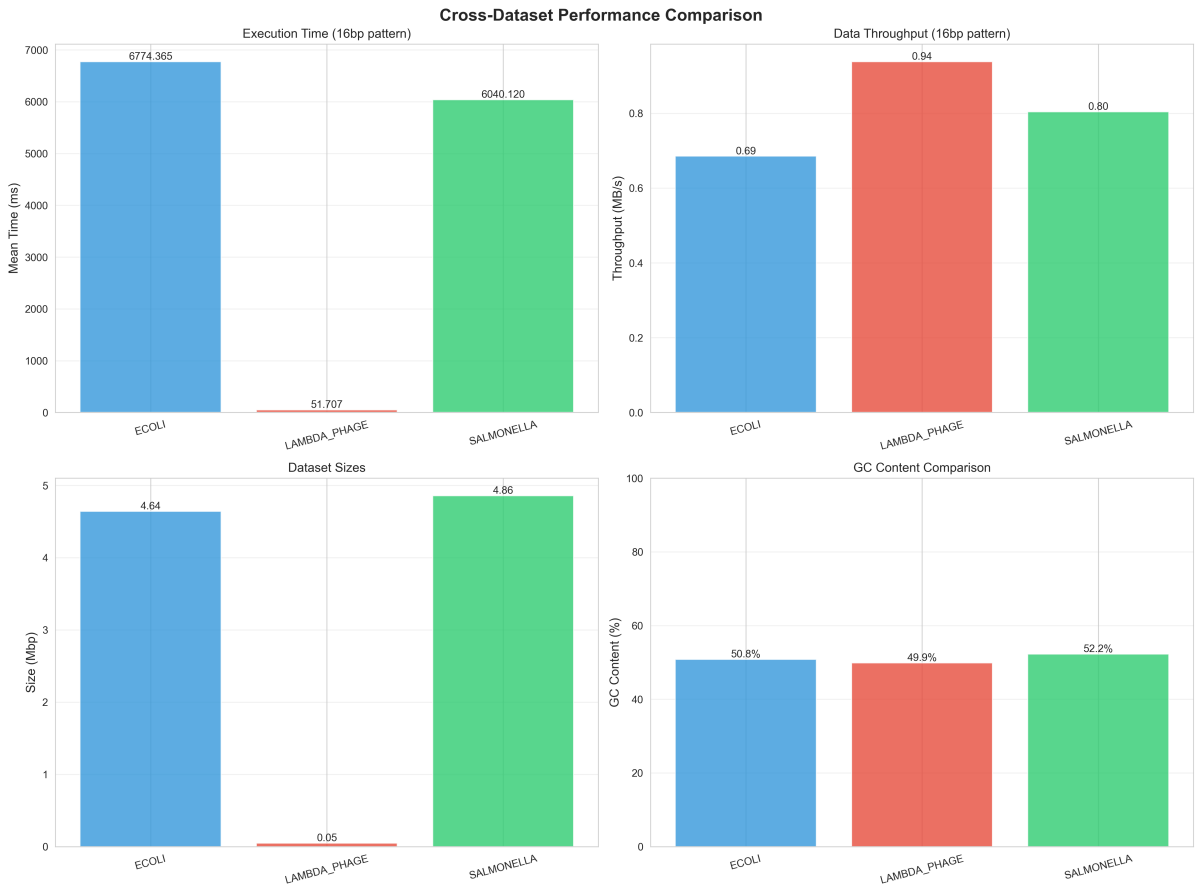


Figure 5.9 Boyer-Moore cross-dataset comparison showing performance consistency across E. coli, Lambda Phage and Salmonella genomes

Chapter 6

Discussion and Recommendations

6.1 Interpretation of Results

The experimental results reveal fundamental trade-offs among the five evaluated algorithms, with no single algorithm dominating across all criteria. This section interprets the findings and provides practical guidance for algorithm selection.

6.1.1 Performance Trade-offs

6.1.1.1 Preprocessing vs. Query Time

A critical trade-off exists between preprocessing cost and query performance:

- **Suffix Array:** Invests 92–97 seconds in preprocessing but achieves 0.02 ms queries
- **KMP/Boyer-Moore:** Minimal preprocessing (less than 1 ms) but slower queries (345 ms and 3,946 ms respectively)

The break-even analysis shows Suffix Array becomes favorable after approximately 275 queries on the same text. For genomic databases with thousands of queries, this investment is highly worthwhile. For one-time searches, pattern-preprocessing algorithms are superior.

6.1.1.2 Memory vs. Speed

Memory-time trade-offs also emerge:

- **Suffix Array:** Uses 18–19 MB memory for exceptional query speed
- **KMP:** Uses only 0.23–78 KB with competitive performance
- **Wagner-Fischer:** Minimal memory (30 KB) but O(mn) time complexity

For memory-constrained environments (embedded systems, mobile devices), KMP or Wagner-Fischer are preferable. For server environments with abundant memory, Suffix Array's speed justifies its footprint.

6.1.2 DNA-Specific Insights

6.1.2.1 Small Alphabet Effects

DNA's 4-letter alphabet significantly impacts algorithm behavior:

1. **Boyer-Moore:** The bad-character heuristic is less effective on small alphabets. Character repetitions are frequent, reducing skip distances. This explains Boyer-Moore's moderate performance compared to English text applications.

2. **Shift-Or/Bitap:** Benefits tremendously from small alphabet. Only 4 bitmasks required regardless of pattern complexity. This makes Shift-Or particularly attractive for DNA analysis despite Python implementation overhead.

3. **KMP:** Unaffected by alphabet size. Performance remains consistent, making it reliable across different sequence types.

6.1.2.2 Approximate Matching Requirements

Biological sequences inherently contain errors from:

- Sequencing technology noise (typical error rate: 0.1–1%)
- Natural mutations and polymorphisms
- Alignment uncertainties

Only Shift-Or/Bitap and Wagner-Fischer support approximate matching natively. The 3.5x overhead per error level for Shift-Or is acceptable for $k \leq 3$ errors, which covers most biological applications. Wagner-Fischer provides exact edit distances but at O(mn) cost.

6.2 Algorithm Selection Guidelines

Based on experimental results, we provide recommendations for different use cases:

6.2.1 Use Case 1: Single Exact Pattern Search

Recommendation: KMP Algorithm
Rationale:

- Guaranteed O(n+m) time complexity with no worst-case degradation
- Minimal preprocessing overhead (¡1 ms)
- Low memory footprint (0.23–78 KB)

- Consistent performance across sequence properties
- Simple implementation and debugging

Example Applications:

- Command-line tools for genome search
- Bioinformatics pipelines with one-time queries
- Educational implementations

6.2.2 Use Case 2: Genomic Database with Multiple Queries

Recommendation: Suffix Array
Rationale:

- Exceptional query performance (0.02 ms, 200+ Mbp/s)
- Break-even after $\sim$275 queries
- Supports rapid repeated searches
- Enables motif discovery and repeat finding

Example Applications:

- Reference genome databases (e.g., BLAST-like tools)
- Motif databases with frequent searches
- Comparative genomics platforms
- Real-time genome browsers

Implementation Note: Consider advanced construction algorithms (e.g., SA-IS) to reduce the $O(n^2)$ preprocessing to O(n).

6.2.3 Use Case 3: Approximate/Fuzzy Matching

Recommendation: Shift-Or/Bitap Algorithm
Rationale:

- Native k-error approximate matching support
- Reasonable overhead (3.5x–7.8x for k=1–3)
- Memory-efficient (67–143 KB)

- Handles sequencing errors and mutations naturally

Example Applications:

- Read alignment with error tolerance
- Primer design tools
- Mutation detection pipelines
- Quality-filtered sequence matching

Implementation Note: For production use, implement in C/C++ for 100x speedup over pure Python.

6.2.4 Use Case 4: Edit Distance and Alignment

Recommendation: Wagner-Fischer Algorithm
Rationale:

- Computes exact edit distance
- Enables traceback for alignment reconstruction
- Minimal memory with row optimization
- Serves as baseline for more sophisticated alignment

Example Applications:

- Pairwise sequence alignment
- Variant calling with edit distance thresholds
- Sequence similarity scoring
- Basis for Smith-Waterman local alignment

6.2.5 Use Case 5: Long Patterns (64 bp)

Recommendation: Boyer-Moore or KMP
Rationale:

- No pattern length limitations
- Boyer-Moore improves with longer patterns (better heuristic effectiveness)
- KMP provides consistent performance

Example Applications:

- Gene sequence searches (100–10,000 bp)
- Plasmid or vector searches
- Large motif or domain searches

6.2.6 Use Case 6: Memory-Constrained Environments

Recommendation: KMP or Wagner-Fischer
Rationale:

- Minimal memory footprint
- No large index structures required
- Suitable for embedded systems or mobile applications

6.2.7 Use Case 7: Streaming/Real-time Data

Recommendation: KMP or Shift-Or
Rationale:

- Online algorithms processing data sequentially
- No random access required
- Suitable for nanopore sequencing real-time analysis

Chapter 7

Conclusion

7.1 Summary of Contributions

This project presented a comprehensive comparative analysis of five pattern matching algorithms for DNA sequence analysis. We implemented and evaluated Knuth-Morris-Pratt (KMP), Boyer-Moore, Suffix Array, Shift-Or/Bitap and Wagner-Fischer algorithms across six evaluation criteria using real genomic datasets totaling over 9.5 million base pairs.

7.1.1 Key Findings

1. **No Universal Best Algorithm:** Each algorithm excels in specific scenarios, with fundamental trade-offs between preprocessing cost, query performance, memory usage and approximate matching capability.

2. **Suffix Array for Databases:** After expensive preprocessing (92–97 seconds), Suffix Array delivers exceptional query performance (0.02 ms, 200+ Mbp/s), making it ideal for genomic databases with repeated queries (break-even at $\sim$275 queries).

3. **KMP for Reliability:** KMP provides consistent O(n+m) performance with minimal memory footprint, making it the best general-purpose choice for single queries and memory-constrained environments.

4. **Approximate Matching Essential:** Only Shift-Or/Bitap and Wagner-Fischer support mutation handling, critical for biological applications with sequencing errors. Shift-Or achieves this with reasonable overhead (3.5x–7.8x for k=1–3 errors).

5. **Small Alphabet Effects:** DNA's 4-letter alphabet reduces Boyer-Moore efficiency but benefits Shift-Or/Bitap memory usage (only 4 bitmasks required).

6. **Implementation Matters:** Pure Python Shift-Or is slower than expected; C/C++ implementation would provide 100x speedup, making it highly competitive.

7.1.2 Practical Recommendations

We provided evidence-based algorithm selection guidelines:

- Single exact pattern search: **KMP**
- Genomic databases (more than 275 queries): **Suffix Array**
- Approximate/fuzzy matching: **Shift-Or/Bitap**
- Edit distance computation: **Wagner-Fischer**
- Long patterns (more than 64 bp): **Boyer-Moore or KMP**
- Memory-constrained: **KMP or Wagner-Fischer**
- Streaming/real-time: **KMP or Shift-Or**

7.2 Concluding Remarks

String pattern matching is still a key bioinformatics challenge that directly affects motif identification, genome analysis, and sequence alignment. This extensive review indicates that algorithm selection must be guided by unique application requirements rather than seeking a universal solution.

Despite costly preprocessing, the Suffix Array is the clear victor for database applications because of its outstanding query performance. KMP offers the best combination of ease of use, dependability, and performance for both single queries and general-purpose applications. For the biologically crucial task of approximate matching, Shift-Or/Bitap gives the optimum mix of speed and mistake tolerance for short patterns.

As genomic datasets continue to grow exponentially, the significance of effective pattern matching algorithms will only increase. This paper offers a framework for assessing upcoming advancements in sequence search algorithms as well as a basis for well-informed algorithm selection.
The project repository contains the entire source code, datasets, benchmarking scripts, and experimental findings, allowing the research community to replicate and expand this work.

Project Repository: `https://github.com/Yash-More1224/dna_sequence_matching`

Bibliography

[1] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge, UK: Cambridge University Press, 1997.

[2] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge, UK: Cambridge University Press, 2002.

[3] H. Hyyro, K. Fredriksson, and G. Navarro, "Increased bit-parallelism for approximate and multiple string matching," *Journal of Experimental Algorithmics*, vol. 10, pp. 1–27, 2005.

[4] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Computing Surveys*, vol. 45, no. 2, pp. 1–42, 2013.

[5] T. Lecroq, "Fast exact string matching algorithms," *Information Processing Letters*, vol. 102, no. 6, pp. 229–235, 2007.

[6] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[7] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 2000, pp. 390–398.

[8] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[9] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[10] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[11] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.