PATTERN MATCHING ALGORITHMS ON DNA SEQUENCES

Team name :Firefly

PROBLEM STATEMENT

Implement and analyze multiple string and pattern matching algorithms on DNA sequences (e.g., E. coli genome), including KMP, Boyer-Moore, Suffix Trees for exact matching, and Shift-Or and Levenshtein Distance Search for approximate/fuzzy matching.

Evaluate their performance on multiple datasets, compare latency and memory usage with Python’s built-in regex, and visualize results by highlighting matches or motifs.

PROBLEM OVERVIEW

- DNA has billions of characters → searching manually is slow
- Need efficient algorithms to detect motifs/mutations
- Faster pattern matching = faster biological insights

GOAL OF THE PROJECT

- Implement and analyze multiple exact + approximate matching algorithms
- Compare speed, memory, and accuracy
- Benchmark using real DNA datasets (Ex: E.coli)

ALGORITHMS USED :EXACT MATCHING

- **KMP** → avoids rechecking characters using LPS
- **Boyer-Moore** → skips sections using bad-char + good-suffix
- **Naive Suffix Tree** → Directly inserts all suffixes; simple but slow.

BONUS

- **Horspool** → Fast exact matching by skipping characters on mismatches.
- **Suffix Tree (Ukkonen)** → Compact index enabling instant substring search.

ALGORITHMS USED (FUZZY MATCHING)

- **Shift-Or** → fast bit-parallel operations
- **Levenshtein Distance** → allows insert/delete/substitute errors

BONUS

Damerau Levenshtein Edit Distance → Measures similarity by allowing insertions, deletions, substitutions, and transposition of adjacent characters.

KMP

- Uses LPS table to skip re-checking
- Linear time: O(n + m)
- Super stable performance on repetitive DNA
- Great when text and pattern share prefixes

BOYER–MOORE

- Compares from right → left
- Big skips using bad-character + good-suffix rules
- Often fastest in practice for long patterns
- Best when mismatches are frequent

SUFFIX TREE

- Stores all suffixes of the text in a compact structure
- Query any pattern in O(pattern length)
- Ideal for repeated searches on huge DNA
- Space heavy compared to others

SHIFT-OR

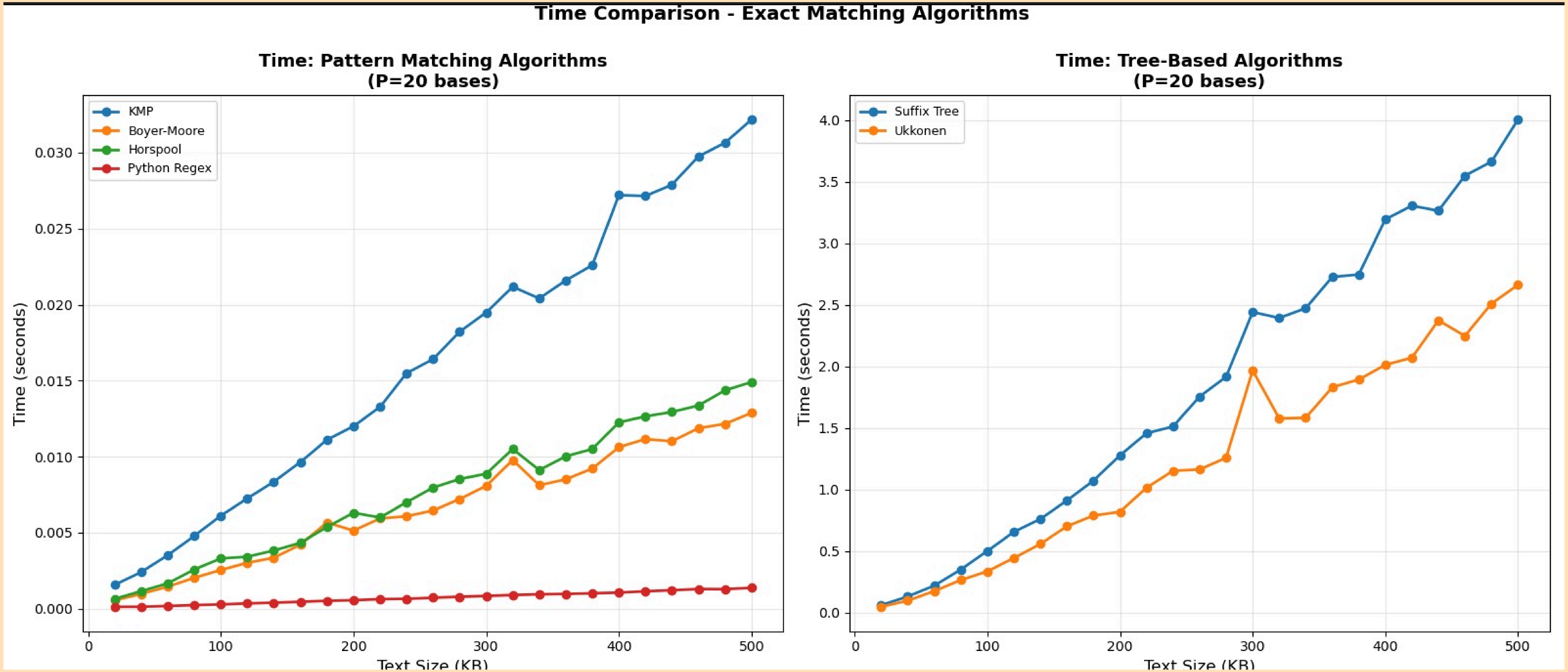
- Uses bit operations for parallel matching
- Very fast for short motifs
- Supports limited mismatches efficiently
- Tiny memory usage

LEVENSHTEIN DISTANCE

- Counts insert / delete / substitute edits
- Detects mutations in DNA sequences
- More accurate but slower ($O(n \times m)$)
- Widely used in bioinformatics alignment

PYTHON BUILTIN REGEX

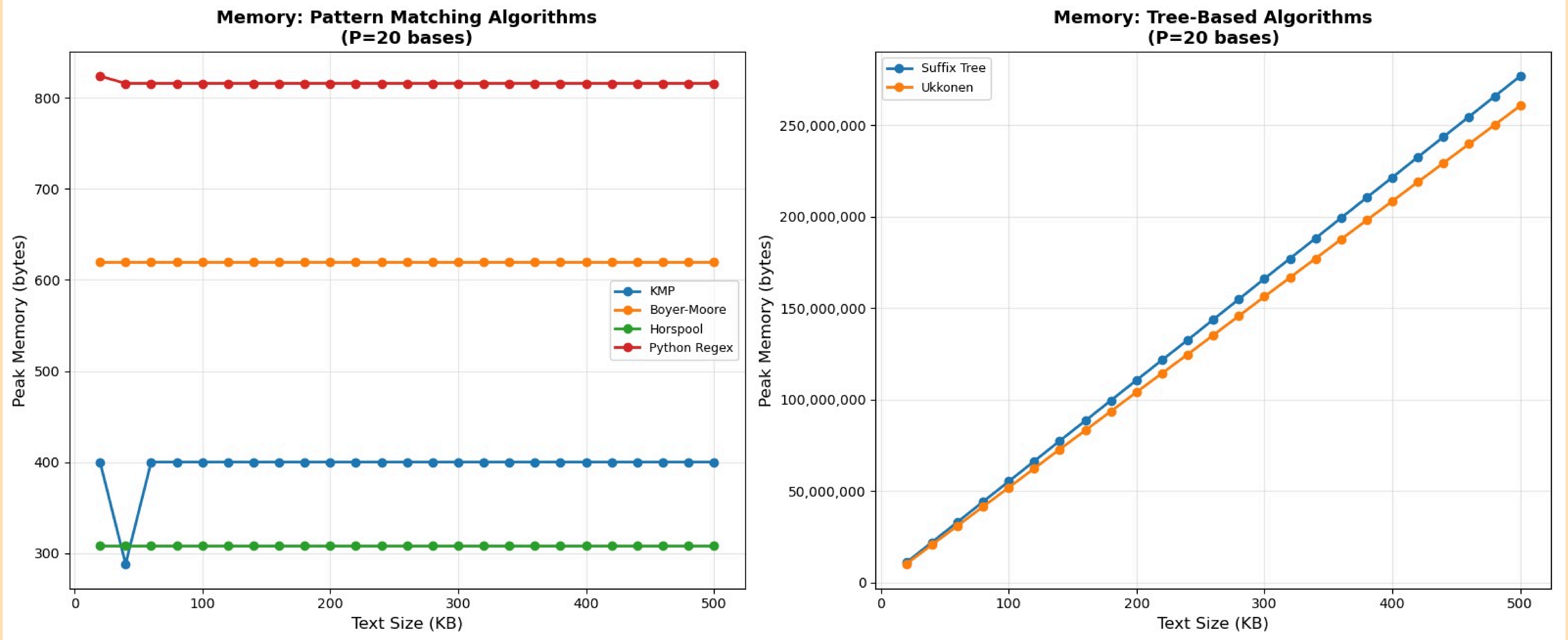
- Uses C-optimized backend, giving faster search performance.
- Very easy to use with built-in **re** library functions.
- Easy-to-use API functions like re.search(), re.findall(), and compiled regex objects via re.compile().
- Low memory usage and no preprocessing required.
- Best for exact matching of short patterns (e.g., DNA motifs).


Time Comparison - Exact Matching Algorithms
Time: Pattern Matching Algorithms (P=20 bases)
KMP
Boyer-Moore
Horspool
Python Regex
Time (seconds)
Text Size (KB)
Time: Tree-Based Algorithms (P=20 bases)
Suffix Tree
Ukkonen
Time (seconds)
Text Size (KB)

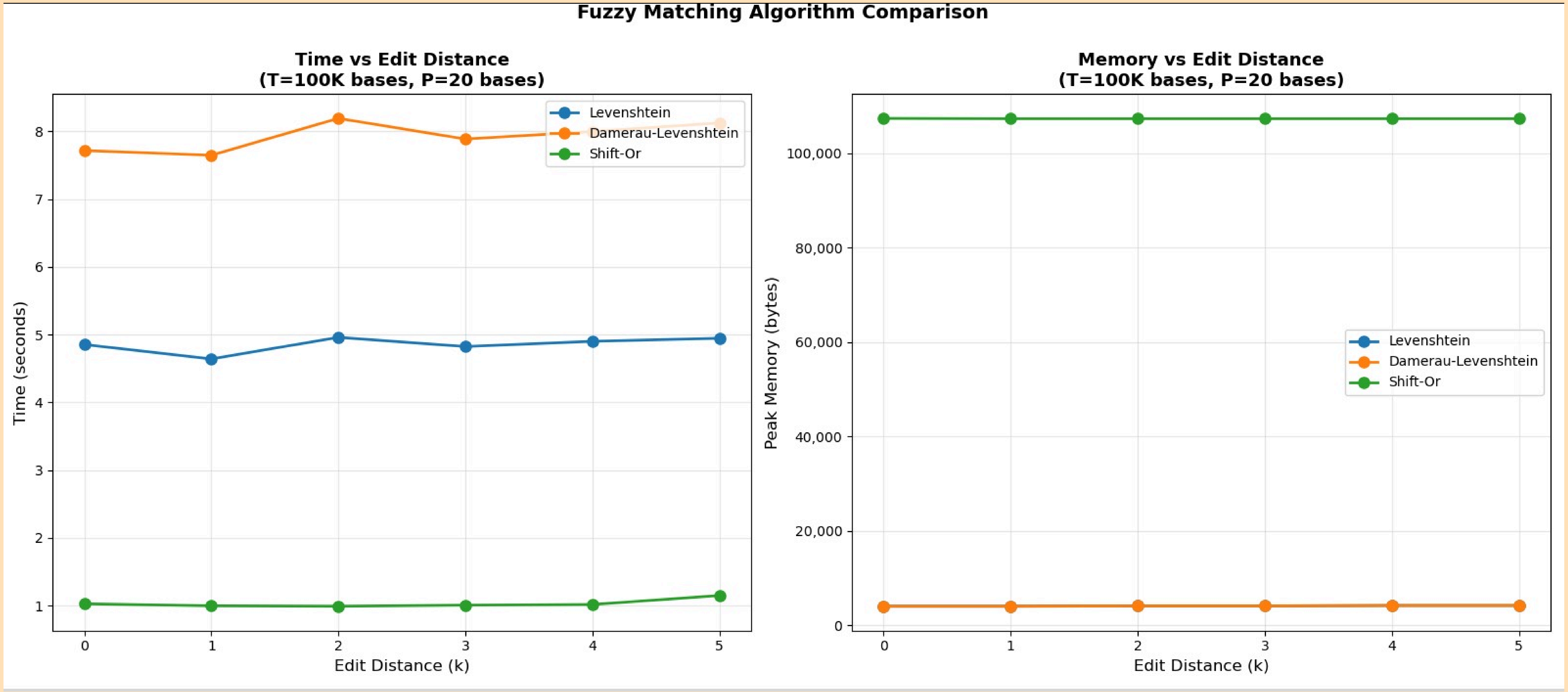
Algorithm Trends

algorithm	Time Trend as Text Size Increases	Relative Speed	Performance Observation
Boyer-Moore	Linear increase, lowest slope	very fast	Efficient due to large character shifts, best for single searches
Horspool	Linear increase, slightly slower than Boyer-Moore	Very Fast	Simple heuristic, competitive performance in practice
Python Regex	Very small linear increase	fastest among all	Highly optimized backend, stable execution times
KMP	Linear increase with some spikes	Slower compared to BM & Horspool	Always scans full text, worst-case behavior dominates
Ukkonen’s Suffix Tree	Time increases steeply	Slowest to execute	High preprocessing cost to build suffix tree

Memory Comparison - Exact Matching Algorithms



algorithm	Memory useage trend	relative memory	Performance Observation
KMP	Constant	Very Low	Uses only prefix table → very small memory footprint
Boyer-Moore	Constant	Low	Only stores bad-character table → memory efficient
Horspool	Constant	Lowest	Minimal lookup table → best space efficiency
Python Regex	Constant	Slightly Higher	Regex engine overhead but still low memory
Naive Suffix Tree	Linear growth	Very High	Stores all suffixes explicitly → heavy memory usage
Ukkonen’s Suffix Tree	Linear growth	High	More compact than naive tree but still large memory overhead



Fuzzy Matching Algorithm Comparison
Time vs Edit Distance
(T=100K bases, P=20 bases)
Time (seconds)
Edit Distance (k)
Levenshtein
Damerau-Levenshtein
Shift-Or
Memory vs Edit Distance
(T=100K bases, P=20 bases)
Peak Memory (bytes)
Edit Distance (k)
Levenshtein
Damerau-Levenshtein
Shift-Or
0
20,000
40,000
60,000
80,000
100,000

Algorithm Trends

Comparative analysis of string searching algorithms performance

Algorithm	Time Performance	Strength	Limitation	Best Use Case	Limitation	Best Use Case
Shift-Or (Bit-Parallel)	Fastest (lowest execution time)	Very efficient due to bit-parallel operations	Limited support for complex edits	Real-time approximate matching with small patterns	Not suitable for complex edit types	Real-time pattern matching with small patterns
Levenshtein Distance	Moderate runtime	Handles common biological mutations	Slower than Shift-Or due to DP computation	General fuzzy matching in DNA sequences	Slower than Shift-Or	General DNA/protein fuzzy matching
Damerau-Levenshtein Distance	Slowest	Supports transposition errors (more accurate)	Highest processing cost	Applications requiring typo/mutatio n toleranc	Highest computational cost	Applications requiring typo/mutation tolerance

Algorithm Trends

Comparative analysis of string searching algorithms performance

Algorithm	Memory Usage	reason	Best Environment
Shift-Or (Bit-Parallel)	Highest	Needs bit-vectors for each pattern state	High-performance systems with enough RAM
Levenshtein Distance	Medium	DP matrix partially stored	Balanced memory constraints
Damerau-Levenshtein Distance	Slightly higher	Additional checks for transposition	Accuracy prioritized over low memory

LIMITATIONS

- Suffix Trees (especially naïve) are slow + memory-heavy for large genomes
- Levenshtein & Damerau have O(nm) runtime — not scalable for huge DNA
- Shift–Or works best only when pattern length $\leq$ word size (64 bits)
- Boyer–Moore optimization drops on repetitive DNA sequences
- Tree-based algorithms scale memory linearly with text size

INTERESTING FINDINGS

- Python Regex was the fastest in most exact matching tests
- Horspool gave surprisingly great real-world performance + lowest memory
- Ukkonen’s algorithm builds suffix trees in true linear time
- Shift–Or was the fastest fuzzy matcher for short motifs
- Trade-offs everywhere: speed vs memory vs accuracy

KEY TAKEAWAYS

- Exact Matching:
- Python Regex ≈ Boyer–Moore > Horspool > KMP
- Fuzzy Matching:
- Shift–Or > Levenshtein > Damerau-Levenshtein
- Suffix Trees dominate when many searches on same genome
- Algorithm choice depends on dataset, pattern length, and mutation tolerance
- Efficient matching directly improves bioinformatics workflows

FUTURE IMPROVEMENTS

- We can test on larger and larger datasets
- We can also try to merge two or more algorithms to get the best out of both of them
- We can further expand and see further more complicated algorithms.