Analysis of Shift-Or (Bitap) Algorithm for DNA Sequences

1 Introduction: Bit-Parallel String Matching

The **Shift-Or algorithm** (also known as **Bitap**, **Shift-And**, or **Baeza-Yates-Gonnet algorithm**) is a bit-parallel string matching technique that leverages bitwise operations to achieve exceptionally fast pattern matching. It was originally invented by Bálint Dömölki in 1964 for exact matching and later extended by Ricardo Baeza-Yates and Gaston Gonnet in 1989, with further extensions by Manber and Wu in 1991 to handle approximate matching with errors.

What the Algorithm Solves

The Shift-Or algorithm solves two fundamental problems:

1. **Exact Pattern Matching**: Given a text T of length n and a pattern P of length m, determine whether P appears as a substring in T.
2. **Approximate (Fuzzy) Pattern Matching**: Find all occurrences of P in T allowing up to k errors (insertions, deletions, or substitutions).

The algorithm's key innovation is encoding the pattern as **bitmasks** and maintaining a **state vector** using simple bitwise operations (shift, OR, AND), making it extremely fast for short patterns.

Uses and Applications

This algorithm is particularly well-suited for:

- **DNA Sequence Analysis**: Searching for short genetic motifs (promoter regions, binding sites) in large genomes where sequencing errors may be present.
- **Text Editors and Search Tools**: The Unix utility `agrep` (approximate grep) uses this algorithm for fuzzy text search.
- **Spell Checkers**: Finding words that approximately match user input despite typos.
- **Biological Pattern Discovery**: Detecting regulatory sequences that may have slight variations due to mutations.

The Shift-Or algorithm excels when:

- Pattern length $m \leq$ word size of the machine (typically 32 or 64 bits)
- Alphabet size is small (perfect for DNA's 4-letter alphabet: A, T, C, G)
- Multiple searches are performed on the same text (preprocessing cost is amortized)

2 The Algorithm: Structure and Function

2.1 Exact Matching Version

The Shift-Or algorithm operates in two phases: **preprocessing** and **searching**.

Phase 1: Preprocessing (Building Bitmasks)

For a pattern *P* of length *m*, we create a **mask table** *B* where each character in the alphabet has a corresponding bitmask.

Definition: For each character *c* in the alphabet, *B[c]* is an *m*-bit integer where bit *i* is set to 1 if *P[i] = c*, otherwise 0.

Example: For pattern *P* = "ACGT" (length *m* = 4):

```
Position in P:     3  2  1  0  (right-to-left indexing)
Pattern:           A  C  G  T

B['A'] = 1000 (binary) = 8 (decimal)  // 'A' is at position 3
B['C'] = 0100 (binary) = 4 (decimal)  // 'C' is at position 2
B['G'] = 0010 (binary) = 2 (decimal)  // 'G' is at position 1
B['T'] = 0001 (binary) = 1 (decimal)  // 'T' is at position 0
B['X'] = 0000 (for any other character)
```

Note: Bits are indexed from right to left, with bit 0 being the rightmost (least significant) bit.

Phase 2: Searching (State Vector Updates)

We maintain a **state vector** *D* (an *m*-bit integer) that tracks which prefixes of *P* currently match suffixes of the text processed so far.

Interpretation: Bit *i* in *D* is 1 if the first *i+1* characters of the pattern match the last *i+1* characters of the text scanned so far.

Initial State: *D* = 0 (no matches initially)

Update Rule: For each character *T[j]* in the text:

```
D = ((D << 1) | 1) & B[T[j]]
```

This operation does three things:

1. `D << 1`: Shift *D* left by 1 bit (extends all partial matches by one position)
2. `| 1`: Set the rightmost bit to 1 (every position can start a new potential match)
3. `& B[T[j]]`: Keep only bits where the pattern character matches the text character

Match Detection: A match is found when bit *m-1* (the leftmost bit for a pattern of length *m*) is set to 1, meaning all *m* characters have matched.

```
if D & (1 << (m-1)) != 0:
    # Pattern found ending at position j
```

2.2 Approximate Matching Extension

To allow up to k errors (substitutions, insertions, deletions), we maintain $k+1$ state vectors: $D_0, D_1, D_2, \ldots, D_k$ where D_i represents matches with exactly i errors.

Update Rules:

```
D₀ = ((D₀ << 1) | 1) & B[T[j]]  // Exact match (0 errors)

For i = 1 to k:
    D_i = ((D_i << 1) | 1) & B[T[j]]     // Substitution
          | ((D_{i-1} << 1) | 1)          // Insertion in text
          | (D_{i-1} & B[T[j]])           // Deletion in text
          | ((D_{i-1} << 1) & B[T[j]])    // Match after previous error
```

Match Detection: A match with up to k errors is found when bit $m-1$ is set in any D_i for $i \leq k$.

---

3 Time and Space Complexity Analysis

Let n be the length of the text T, m be the length of the pattern P, σ be the alphabet size, and k be the maximum number of errors allowed.

3.1 Exact Matching

Preprocessing Time: $O(m \cdot \sigma)$

- We must initialize the bitmask $B[c]$ for each character c in the alphabet.
- For DNA sequences with $\sigma = 4$, this is effectively $O(m)$.

Preprocessing Space: $O(\sigma)$

- We store one m-bit mask for each alphabet character.
- For 64-bit words and $\sigma = 256$ (extended ASCII), this is 256 × 8 bytes = 2 KB.
- For DNA ($\sigma = 4$), only 4 × 8 bytes = 32 bytes.

Search Time: $O(n)$

- For each of the n characters in the text, we perform:
  - One left shift operation: $O(1)$
  - One OR operation: $O(1)$
  - One AND operation: $O(1)$
  - One bit test: $O(1)$
- Total: $O(n)$ time, independent of m (unlike naive $O(n \cdot m)$ search).

Search Space: $O(1)$

- We only maintain the state vector D (one m-bit integer).

Constraint: Pattern length m must fit in a machine word (typically ≤ 64 bits). For longer patterns, multiple words must be used, degrading performance.

3.2 Approximate Matching

Preprocessing Time: $O(m \cdot \sigma)$ (same as exact matching)

Preprocessing Space: $O(\sigma)$ (same as exact matching)

Search Time: $O(k \cdot n)$

- For each text character, we must update $k+1$ state vectors.
- Each update involves several bitwise operations (still constant time per vector).
- Total: $O(k \cdot n)$ where k is typically small (e.g., $k \leq 3$ for DNA).

Search Space: $O(k)$

- We maintain $k+1$ state vectors: $D_0, D_1, \ldots, D_k$.
- Each vector requires m bits (one machine word for $m \leq 64$).

---

4 Proof of Correctness

4.1 Exact Matching Correctness

Claim: The Shift-Or algorithm correctly identifies all occurrences of pattern P in text T.

Proof (by induction on text position):

Invariant: After processing text character $T[j]$, bit i in state vector D is 1 if and only if $P[0...i] = T[j-i...j]$ (the first $i+1$ characters of P match the last $i+1$ characters of the text up to position j).

Base Case ($j = 0$):

- Initial state: $D = 0$ (no matches).
- After processing $T[0]$:

```
D = ((0 << 1) | 1) & B[T[0]] = 1 & B[T[0]]
```

- If $T[0] = P[0]$, then bit 0 of $B[T[0]]$ is 1, so bit 0 of D becomes 1. ✓
- If $T[0] \neq P[0]$, then bit 0 of $B[T[0]]$ is 0, so D remains 0. ✓

Inductive Step: Assume the invariant holds after processing $T[j-1]$. We must show it holds after processing $T[j]$.

Consider bit i in the new state D':

```
D' = ((D << 1) | 1) & B[T[j]]
```

Case 1: $i = 0$ (first character of pattern)

- `(D << 1) | 1` sets bit 0 to 1.
- `& B[T[j]]` keeps bit 0 set only if $T[j] = P[0]$.
- Therefore, bit 0 of D' is 1 $\iff T[j] = P[0]$. ✓

Case 2: $i > 0$

- `D << 1` copies bit $i-1$ of D to bit i of the shifted result.
- By the inductive hypothesis, bit $i-1$ of D was 1 $\iff P[0...i-1] = T[j-1-(i-1)...j-1] = T[j-i...j-1]$.
- `& B[T[j]]` keeps bit i set only if $T[j] = P[i]$.
- Therefore, bit i of D' is 1 $\iff P[0...i-1] = T[j-i...j-1]$ AND $P[i] = T[j] \iff P[0...i] = T[j-i...j]$. ✓

Match Detection: When bit $m-1$ is set, the invariant tells us that $P[0...m-1] = T[j-(m-1)...j]$, meaning the entire pattern matches text ending at position j. This is correct. ∎

4.2 Approximate Matching Correctness (Sketch)

The proof extends to approximate matching by considering each error level separately. State vector D_i maintains matches with exactly i errors. The update rules correctly handle:

1. **Substitution**: `((D`$_i$ `<< 1) | 1) & B[T[j]]` — same as exact match but from state with i errors
2. **Insertion in text** (deletion in pattern): `(D`$_{i-1}$ `<< 1) | 1` — advance text without consuming pattern character
3. **Deletion in text** (insertion in pattern): `D`$_{i-1}$ `& B[T[j]]` — consume pattern character without advancing text
4. **Match after error**: `(D`$_{i-1}$ `<< 1) & B[T[j]]` — characters match but we already have $i-1$ errors

The combination ensures all possible error scenarios are tracked. ∎

---

5 Pseudo Code

5.1 Exact Matching

```
function ShiftOrExact(text T, pattern P):
    m = length(P)
    n = length(T)

    # Constraint check
    if m > 64:
        return "Pattern too long for single-word implementation"

    # Step 1: Preprocessing - Build bitmasks
    B = {}  # Dictionary: character → bitmask
    for each character c in alphabet:
        B[c] = 0
```

```
    for i from 0 to m-1:
        B[P[i]] = B[P[i]] | (1 << i)  # Set bit i for character P[i]

    # Step 2: Searching
    D = 0  # Initial state: no matches
    matches = []

    for j from 0 to n-1:
        # Update state vector
        D = ((D << 1) | 1) & B[T[j]]

        # Check for complete match
        if D & (1 << (m-1)) != 0:
            matches.append(j - m + 1)  # Store starting position

    return matches
```

5.2 Approximate Matching (with up to k errors)

```
function ShiftOrApproximate(text T, pattern P, max_errors k):
    m = length(P)
    n = length(T)

    # Step 1: Preprocessing (same as exact)
    B = {}
    for each character c in alphabet:
        B[c] = 0
    for i from 0 to m-1:
        B[P[i]] = B[P[i]] | (1 << i)

    # Step 2: Initialize state vectors for each error level
    D = array of size (k+1)  # D[0], D[1], ..., D[k]
    for i from 0 to k:
        D[i] = 0

    # Step 3: Searching
    matches = []

    for j from 0 to n-1:
        # Update D[0] (exact match)
        old_D0 = D[0]
        D[0] = ((D[0] << 1) | 1) & B[T[j]]

        # Update D[i] for i = 1 to k (with errors)
        for i from 1 to k:
            old_Di = D[i]

            # Four possible transitions to state with i errors:
            D[i] = ((D[i] << 1) | 1) & B[T[j]]     # Substitution
            D[i] = D[i] | ((old_Di << 1) | 1)       # Insertion (skip text
```

```
char)
            D[i] = D[i] | (old_D[i-1] & B[T[j]])   # Deletion (skip pattern
char)
            D[i] = D[i] | ((old_D[i-1] << 1) & B[T[j]])  # Match after
error

            old_D[i-1] = D[i-1]  # Save for next iteration

        # Check for matches with up to k errors
        for i from 0 to k:
            if D[i] & (1 << (m-1)) != 0:
                matches.append((j - m + 1, i))  # (position, num_errors)
                break  # Report only the best match (fewest errors)

    return matches
```

---

6 Dry Run: Exact Matching on DNA Sequence

Let's trace the algorithm on a simple DNA example.

Text: T = "AACGT"
Pattern: P = "CG"
Expected: Pattern found at position 2

Step 1: Preprocessing

Pattern P = "CG" has length m = 2.

Build bitmasks (*right-to-left indexing, bit 0 is rightmost*):

```
Position:  1  0
Pattern:   C  G

B['C'] = 10 (binary) = 2 (decimal)
B['G'] = 01 (binary) = 1 (decimal)
B['A'] = 00 (binary) = 0 (decimal)
B['T'] = 00 (binary) = 0 (decimal)
```

Step 2: Searching

j	*T[j]*	*D* before	`(D << 1) | 1`	`B[T[j]]`	*D* after	Bit 1 set?	Match?
-	-	`00`	-	-	`00`	No	-
0	A	`00`	`01`	`00`	`00`	No	No
1	A	`00`	`01`	`00`	`00`	No	No
2	C	`00`	`01`	`10`	`00`	No	No

j	*T[j]*	*D* before	`(D << 1) | 1`	`B[T[j]]`	*D* after	Bit 1 set?	Match?
3	G	`00`	`01`	`01`	`01`	No	No

Wait, this doesn't match! Let me recalculate more carefully with proper state tracking:

j	*T[j]*	*D* (binary)	`D << 1`	`(D << 1) | 1`	`B[T[j]]`	`& B[T[j]]`	Bit 1?
-	-	`00`	-	-	-	`00`	No
0	A	`00`	`00`	`01`	`00`	`00`	No
1	A	`00`	`00`	`01`	`00`	`00`	No
2	C	`00`	`00`	`01`	`10`	`00`	No
3	G	`00`	`00`	`01`	`01`	`01`	No

Still not matching! The issue is we need to track state properly:

Corrected trace:

Initial: *D* = `00`

j = 0, T[0] = 'A':

- `D << 1` = `00 << 1` = `00`
- `(D << 1) | 1` = `00 | 01` = `01`
- `B['A']` = `00`
- `D` = `01 & 00` = `00`

j = 1, T[1] = 'A':

- Same as above, *D* = `00`

j = 2, T[2] = 'C':

- `D << 1` = `00 << 1` = `00`
- `(D << 1) | 1` = `01`
- `B['C']` = `10`
- `D` = `01 & 10` = `00`

Hmm, still `00`. The issue is bit 0 of `B['C']` is 0 (because 'C' is at position 1, not position 0).

Wait, let me reconsider the bitmask construction. If pattern is "CG":

- P[0] = 'C' → bit 0 of B['C'] should be 1
- P[1] = 'G' → bit 1 of B['G'] should be 1

So:

```
B['C'] = 01 (binary) = 1
B['G'] = 10 (binary) = 2
```

Retry:

j = 2, T[2] = 'C':

- `(D << 1) | 1` = `01`
- `B['C']` = `01`
- `D` = `01 & 01` = `01` ✓ (bit 0 set: 'C' matches P[0])

j = 3, T[3] = 'G':

- `D << 1` = `01 << 1` = `10`
- `(D << 1) | 1` = `10 | 01` = `11`
- `B['G']` = `10`
- `D` = `11 & 10` = `10` ✓ (bit 1 set: "CG" matches!)

Match detected: Bit 1 (= m-1) is set at position j = 3.
Starting position: 3 - 2 + 1 = **2** ✓

---

7 Graphical Visualization

For pattern "CG" in text "AACGT":

```
Text positions:   0   1   2   3   4
Text:             A   A   C   G   T
                            ^^^
                            match

State evolution:
j=0: D = 00 (no match)
j=1: D = 00 (no match)
j=2: D = 01 (bit 0 set: first char 'C' matches)
j=3: D = 10 (bit 1 set: full pattern "CG" matches)
     └─▶ MATCH FOUND at position 2
j=4: D = 00 (reset)
```

Visualization of Approximate Matching (1 error allowed)

Text: "ACCT"
Pattern: "ACG"
Max errors: k = 1

Bitmasks:

```
B['A'] = 001  (bit 0)
B['C'] = 010  (bit 1)
B['G'] = 100  (bit 2)
B['T'] = 000
```

State vectors D_0 (exact) and D_1 (1 error):

j	$T[j]$	D_0	D_1	Detection
0	A	`001`	`001`	-
1	C	`010`	`011`	-
2	C	`000`	`110`	Bit 2 set in D_1 → Match with 1 error!
3	T	`000`	`001`	-

Result: Pattern "ACG" matches "ACC" with 1 substitution (G→C) at position 0.

---

8 Situational Performance on DNA Sequences

8.1 Strengths

1. **Blazing Fast for Short Patterns**: For patterns ≤ 64 bases (fits in one machine word), the algorithm runs in true *O(n)* time with very low constant factors. Bitwise operations are among the fastest CPU instructions.
2. **Optimal for DNA's Small Alphabet**: With only 4 characters (A, T, C, G), bitmask preprocessing is trivial (*O(4m) = O(m)*) and requires minimal memory (32 bytes for 64-bit masks).
3. **Efficient Approximate Matching**: Unlike Levenshtein distance which requires *O(n·m)* time, Shift-Or handles k errors in *O(k·n)* time. For small k (typical in biological contexts: 1-3 mismatches), this is near-linear.
4. **Parallelism-Friendly**: Multiple pattern bits are updated simultaneously in a single bitwise operation, providing inherent parallelism.
5. **Cache-Efficient**: Minimal memory access (only state vectors and bitmask table), leading to excellent cache performance.

8.2 Weaknesses

1. **Pattern Length Limitation** (CRITICAL): The algorithm's performance degrades significantly for patterns longer than the machine word size:
   - 32-bit systems: $m \leq 32$
   - 64-bit systems: $m \leq 64$
   - For longer patterns, multiple words must be used, requiring manual carry-bit handling and destroying the simplicity advantage.
2. **Not Suitable for Long Genomic Motifs**: Many biologically relevant patterns exceed 64 bases:
   - Gene promoters: often 100-200 bp
   - Transcription factor binding sites: 6-20 bp (✓ suitable)
   - Full genes: thousands of bp (✗ unsuitable)

3. **No Substring Reuse**: Unlike suffix trees, Shift-Or doesn't preprocess the *text*. Each new search on the same genome requires *O(n)* time. For multiple pattern searches, suffix structures are more efficient.

4. **Limited Gap Penalty Control**: The approximate matching extension treats all errors equally (cost = 1). Biology often requires sophisticated scoring:

    - Affine gap penalties (opening a gap is costly, extending is cheap)
    - Position-specific scoring matrices (PSSM)
    - Different costs for transitions (A ↔ G, C ↔ T) vs transversions (A ↔ C, A ↔ T, G ↔ C, G ↔ T)

5. **Comparison to Specialized Tools**:

    - **BLAST**: *O(n+m)* expected time for database searches using heuristic seed-and-extend, far more practical for large-scale genomics.
    - **Bowtie/BWA**: Use FM-index (compressed suffix array) for billions of short reads, *O(m)* per query after *O(n)* preprocessing.

8.3 Ideal Use Cases in DNA Analysis

✅ **Perfect for**:

- **Primer searching**: Short primers (15-30 bp) in PCR design
- **Restriction site finding**: Restriction enzymes recognize 4-8 bp sequences
- **SNP detection**: Single nucleotide polymorphisms with 1-2 nearby variants
- **Quality-filtered read alignment**: Short reads (36-100 bp) with low error tolerance

✅ **Good for**:

- **Transcription factor binding motifs**: Typically 6-20 bp with 1-2 mismatches
- **MicroRNA target sites**: ~22 nucleotides
- **Ribosome binding sites**: Short conserved sequences (~10 bp)

× **Poor for**:

- **Whole gene alignment**: Genes are thousands of base pairs long
- **Chromosome-wide searches**: Better served by indexed structures
- **Homology searches**: Require sophisticated scoring models (use BLAST family)
- **Long-read sequencing alignment**: PacBio/Nanopore reads exceed 10kb (use Minimap2)

---

9 Adaptation for DNA Pattern Discovery

9.1 Scanning for Promoter Motifs

Problem: Find all occurrences of the -10 consensus sequence "TATAAT" (Pribnow box) in *E. coli* genome, allowing up to 2 mismatches.

Solution:

```
genome = load_ecoli_genome()  # 4.6 Mbp
pattern = "TATAAT"  # 6 bp
```

```
max_errors = 2

matches = ShiftOrApproximate(genome, pattern, max_errors)

for (position, errors) in matches:
    context = genome[position-10:position+16]  # Extract context
    if is_valid_promoter_context(context):
        print(f"Potential promoter at {position} with {errors} mismatches")
```

Performance: *O(2 × 4.6M)* = ~9.2M operations, completing in milliseconds.

9.2 Multiple Patterns: Aho-Corasick vs Repeated Shift-Or

For searching *k* different short patterns in the same genome:

- **Repeated Shift-Or**: *O(k · n)* total time
- **Aho-Corasick automaton**: *O(n + k · m)* preprocessing, *O(n)* search for all patterns

Recommendation: For *k* < 10-20 and short patterns, repeated Shift-Or is simpler and sufficiently fast. For *k* > 100, build an Aho-Corasick automaton.

9.3 Handling Complementary Strands

DNA is double-stranded, so patterns may appear on either strand. Modify the algorithm:

```
def search_both_strands(genome, pattern, max_errors):
    forward_matches = ShiftOrApproximate(genome, pattern, max_errors)

    reverse_complement = get_reverse_complement(pattern)
    reverse_matches = ShiftOrApproximate(genome, reverse_complement,
max_errors)

    return merge_results(forward_matches, reverse_matches)
```

---

10 Comparison with Other Algorithms

Algorithm	Preprocessing	Search Time	Space	Best Use Case
Shift-Or (exact)	*O(m·σ)*	*O(n)*	*O(σ)*	Short patterns, small alphabet
Shift-Or (approx)	*O(m·σ)*	*O(k·n)*	*O(k·σ)*	Short patterns with few errors
KMP	*O(m)*	*O(n)*	*O(m)*	Any pattern length, guaranteed linear

Algorithm	Preprocessing	Search Time	Space	Best Use Case
Boyer-Moore	$O(m+\sigma)$	$O(n/m)$ best	$O(m+\sigma)$	Long patterns, large alphabet
Suffix Tree	$O(n)$	$O(m)$	$O(n)$	Multiple queries on same text
Levenshtein	None	$O(n \cdot m)$	$O(n \cdot m)$ or $O(min(n,m))$	Global alignment, any length

Shift-Or wins when:

- $m \leq 64$ (pattern fits in one word)
- σ is small (4 for DNA)
- k is small (≤ 3 errors)
- Real-time performance matters

Shift-Or loses when:

- $m > 64$ (requires multi-word implementation)
- Need complex scoring (use Smith-Waterman)
- Multiple patterns (use Aho-Corasick)
- Text is reused (use suffix structure)

---

11 Conclusion

The Shift-Or (Bitap) algorithm represents an elegant fusion of bit-level parallelism and pattern matching theory. For DNA sequence analysis involving short motifs (≤64 bp) with small error tolerances, it offers unmatched speed and simplicity. The algorithm's $O(n)$ exact search and $O(k \cdot n)$ approximate search complexity, combined with trivial memory requirements, make it ideal for real-time applications like primer validation, restriction site discovery, and regulatory motif scanning.

However, the hard constraint on pattern length ($m \leq$ word size) limits its applicability to longer genomic features. For comprehensive DNA analysis pipelines, Shift-Or should be viewed as a specialized tool:

- **Use for**: Short, critical patterns where speed is paramount
- **Combine with**: Suffix structures for long patterns, BLAST for homology searches, specialized aligners for sequencing data

In the context of our project comparing string matching algorithms on *E. coli* genomic data, Shift-Or will excel at finding short regulatory sequences and demonstrating the power of bit-parallel computation, while exposing the trade-offs inherent in algorithm design: the tension between generality and specialization, between simplicity and sophistication.

References:

1. Baeza-Yates, R., & Gonnet, G. H. (1989). *A new approach to text searching*. Communications of the ACM.

2. Manber, U., & Wu, S. (1991). *Fast text searching allowing errors*. Communications of the ACM.
3. Wu, S., & Manber, U. (1992). *Agrep – A fast approximate pattern-matching tool*. USENIX Technical Conference.

This document provides a comprehensive analysis of the Shift-Or (Bitap) algorithm suitable for understanding its application in DNA sequence matching as part of the comparative algorithm analysis project.