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

1. Introduction: Bit-Parallel Matching

The **Shift-Or algorithm** (also known as **Bitap** or **Baeza-Yates-Gonnet algorithm**) is a bit-parallel string matching technique that leverages bitwise operations to achieve high-speed pattern matching. It is especially well-suited for finding a pattern *P* (length *m*) in a text *T* (length *n*).

Its key innovation is encoding the pattern as a set of **bitmasks** and maintaining a **state vector** (a single integer) that tracks all potential matches simultaneously. This allows it to process the text in *O(n)* time, provided the pattern is short enough to fit within a machine word (e.g., $m \le 64$).

This document analyzes the algorithm and its adaptation for approximate (fuzzy) matching, which is highly relevant for DNA sequences where small mutations or errors are common.

2. The Algorithm: Exact Matching

The algorithm operates in two phases: preprocessing the pattern and searching the text.

2.1 Phase 1: Preprocessing (Building Bitmasks)

We create a **mask table** *B* for the pattern *P*. For each character *c* in the alphabet (e.g., {A, C, G, T}), *B[c]* is an *m*-bit integer.

Definition: Bit *i* of the mask *B[c]* is set to **1** if the character *P[i] = c*. Otherwise, the bit is 0.

Note: For the Shift-Or algorithm, it is common to use 0 for a match and 1 for a mismatch to work with bitwise OR. This analysis will follow the pseudo-code's `(1 << i)` logic, where a 1 bit represents the position of a character.

Example: For pattern *P* = "ACG" (*m* = 3). We use 0-based indexing from right-to-left.

- `P[0]` = 'A' → Bit 0
- `P[1]` = 'C' → Bit 1
- `P[2]` = 'G' → Bit 2

The corresponding bitmasks are:

- `B['A'] = 001` (binary)
- `B['C'] = 010` (binary)
- `B['G'] = 100` (binary)
- `B['T'] = 000` (binary)

2.2 Phase 2: Searching (State Vector Updates)

We maintain a single *m*-bit **state vector** *D*.

Interpretation: Bit *i* in *D* is **1** if the first *i+1* characters of the pattern (*P[0...i]*) match the last *i+1* characters of the text scanned so far.

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

Update Rule: For each character *T[j]* in the text, we update *D*:

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

This single line performs three actions:

1. **`D << 1`** : Shifts all partial matches left by one. A match of *P[0...i-1]* now becomes a potential match for *P[0...i]*.
2. **`| 1`** : Sets the 0-th bit to 1. This speculates that the current text character *T[j]* might be the start of a new match (i.e., it matches *P[0]*).
3. **`& B[T[j]]`** : This is the "filter." It keeps a bit in *D* set to 1 *only if* the corresponding pattern character *P[i]* matches the current text character *T[j]*.

Match Detection: A complete match is found at position *j* if the *m-1* bit (the final bit) of *D* is 1.

3. Algorithm Adaptation: Approximate Matching

To allow up to *k* errors (insertions, deletions, or substitutions), we adapt the algorithm by maintaining *k+1* state vectors: $D_0, D_1, D_2, \ldots, D_k$

Where D_i is an *m*-bit vector that tracks all matches of the pattern *P* that end at the current text position with **exactly *i* errors**.

Update Rule: When processing text character *T[j]*, the new states are calculated based on the *previous* states (before processing *T[j]*), which we'll call `prev_D`.

- **For D_0 (0 errors):** Same as the exact algorithm. `D₀ = ((prev_D₀ << 1) | 1) & B[T[j]]`
- **For D_i (1 to *k* errors):** The new state D_i is a bitwise OR of all possibilities that could lead to an *i*-error match:
    1. **Substitution (or Match):** A previous *i*-error match extended by one character. `sub = ((prev_Dᵢ << 1) | 1) & B[T[j]]`
    2. **Insertion:** A previous *(i-1)*-error match, plus one insertion (we skip a pattern character). `ins = prev_Dᵢ₋₁`

3. **Deletion:** A previous *(i-1)*-error match, plus one deletion (we skip a text character). `del = (prev_D_{i-1} << 1) | 1`

The full recurrence relation combines these: `D_i = sub | ins | del | ((prev_D_{i-1} << 1) & B[T[j]])` *(The last term handles a match on top of an i-1 error state).*

Match Detection: A match with *at most k* errors is found at position *j* if the *m-1* bit is set in *any* `D_i` (where $i \le k$).

---

4. Time and Space Complexity Analysis

Let *n* = text length, *m* = pattern length, *σ* = alphabet size, and *k* = number of errors.

Algorithm	Preprocessing Time	Search Time	Space
Shift-Or (Exact)	*O(m·σ)*	*O(n)*	*O(σ)*
Shift-Or (Approx.)	*O(m·σ)*	*O(k·n)*	*O(k·σ)*

Analysis:

- **Time**: The search time is linear *O(n)* for exact matching because each of the *n* text characters is processed with a few constant-time *O(1)* bitwise operations. For approximate matching, we do *k* such operations per character, leading to *O(k·n)*.
- **Space**: Space is dominated by the bitmask table *B*, which stores one *m*-bit integer for each of the *σ* characters. For DNA, *σ* is a small constant (4), so preprocessing time is *O(m)* and space is *O(1)* (relative to *n*).

---

5. Proof of Correctness (Exact Matching)

We prove by induction that the algorithm is correct.

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]*.

Base Case (*j* = 0):

- Initial state: *D* = `0`.
- Update: `D = ((0 << 1) | 1) & B[T[0]]` `= 1 & B[T[0]]`
- Bit 0 of *D* will be 1 ⟺ Bit 0 of `B[T[0]]` is 1.
- Bit 0 of `B[T[0]]` is 1 ⟺ *P[0] = T[0]*.
- Thus, the invariant holds for *i*=0.

Inductive Step: Assume the invariant holds after processing *T[j-1]*. Let this state be `prev_D`. We must show it holds for *T[j]*.

- New state: `D = ((prev_D << 1) | 1) & B[T[j]]`
- Consider bit *i* in the new *D*. For it to be 1:
  1. Bit *i* of `B[T[j]]` must be 1. This means **`P[i] = T[j]`**.
  2. Bit *i* of `(prev_D << 1) | 1` must be 1.
     - If *i* > 0, this requires bit *i-1* of `prev_D` to be 1.
     - By the hypothesis, bit *i-1* of `prev_D` = 1 ⟺ **`P[0...i-1] = T[j-1-(i-1)...j-1]`** (i.e., `P[0...i-1] = T[j-i...j-1]`).
- Combining (1) and (2): Bit *i* of *D* is 1 ⟺ (`P[i] = T[j]`) AND (`P[0...i-1] = T[j-i...j-1]`).
- This is true ⟺ **`P[0...i] = T[j-i...j]`**.
- The invariant holds.

Conclusion: A match is detected when bit *m-1* is 1. By the invariant, this means `P[0...m-1] = T[j-(m-1)...j]`, which is the definition of a full pattern match ending at *j*.

---

6. Pseudo Code

6.1 Exact Matching

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

```
# --- Preprocessing ---
B = {}  # Bitmask table
for 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]

# --- Searching ---
D = 0  # State vector
matches = []

match_bit = 1 << (m-1)  # Bit to check for a full match

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

    if (D & match_bit) != 0:
        matches.append(j - m + 1)  # Store start position

return matches
```

6.2 Approximate Matching (k errors)

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

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

# --- Searching ---
# D[i] stores state for exactly i errors
D = array[k+1] filled with 0

matches = []
match_bit = 1 << (m-1)

for j from 0 to n-1:
    prev_D = copy(D) # Store state from text char j-1

    # Update D[0] (0 errors)
    D[0] = ((prev_D[0] << 1) | 1) & B[T[j]]

    # Update D[1]...D[k]
    for i from 1 to k:
        # 1. Substitution (or match) on prev i-error state
        sub = ((prev_D[i] << 1) | 1) & B[T[j]]
        # 2. Deletion on prev (i-1)-error state
        del = (prev_D[i-1] << 1) | 1
        # 3. Insertion on prev (i-1)-error state
        ins = prev_D[i-1]
        # 4. Match on prev (i-1)-error state
        mat = (prev_D[i-1] << 1) & B[T[j]]

        D[i] = sub | del | ins | mat

    # Check for a match with at most k errors
    if (D[k] & match_bit) != 0:
        matches.append(j - m + 1)

# De-duplicate or refine match positions as needed
return unique(matches)
```

7. Dry Run: Exact Matching on DNA

Text (T): `AACGT` **Pattern (P)**: `ACG` (*m* = 3)

Step 1: Preprocessing

- `P[0]` ='A', `P[1]` ='C', `P[2]` ='G'
- `B['A'] = 001`
- `B['C'] = 010`
- `B['G'] = 100`
- `B['T'] = 000`
- `match_bit = 1 << (3-1) = 100`

Step 2: Searching

`D` is initialized to `000`.

j	T[j]	`D` (before)	`(D << 1) | 1`	`B[T[j]]`	`D` (after)	`D & 100` ?
0	'A'	`000`	`001`	`001` (`B['A']`)	`001`	`000` (No)
1	'A'	`001`	`011`	`001` (`B['A']`)	`001`	`000` (No)
2	'C'	`001`	`011`	`010` (`B['C']`)	`010`	`000` (No)
3	'G'	`010`	`101`	`100` (`B['G']`)	`100`	`100` (Yes!)
4	'T'	`100`	`001` *	`000` (`B['T']`)	`000`	`000` (No)

(At j=4, `D << 1 = 1000`. The high bit is lost, so `000`. `000 | 1 = 001`)

Result: Match found at `j=3`. Starting position = `j - m + 1` = `3 - 3 + 1` = 1. The match is `T[1...3]` = "ACG". This is correct.

8. Situational Performance (On DNA Sequences)

8.1 Strengths

1. **Extreme Speed for Short Patterns**: For patterns where $m \le 64$, the $O(n)$ or $O(k \cdot n)$ time complexity combined with extremely low-constant-factor bitwise operations makes Shift-Or one of the fastest algorithms available.
2. **Ideal for DNA's Small Alphabet**: The $O(m \cdot \sigma)$ preprocessing is trivial for DNA where σ=4.
3. **Built-in Approximate Matching**: The extension to *k* errors is natural and efficient, making it perfect for finding DNA motifs that may contain small mutations (SNPs) or sequencing errors.
4. **Simplicity**: The exact-match algorithm is remarkably simple to implement.

8.2 Weaknesses (Major Performance Issues)

1. **Hard Pattern Length Limit**: The algorithm's primary weakness. Its performance relies on the pattern length *m* fitting into a single machine word (e.g., 64 bits).
   - If $m > 64$, the implementation becomes vastly more complex, requiring an *array* of integers for the state vector and masks, and all $O(1)$ bitwise operations become $O(m/w)$ (where *w* is word size).
   - This "multi-word" Shift-Or loses its core speed advantage and is much slower than algorithms like KMP or Boyer-Moore for long patterns.
2. **Biologically Naive Scoring**: Like Levenshtein, the *k*-errors adaptation is biologically naive. It treats all operations (substitutions, insertions, deletions) as having an equal cost of 1.
   - It cannot distinguish between a biologically common transition (A→G) and a rare transversion (A→T).
   - It cannot use **affine gap penalties** (e.g., a single 10-base-pair deletion is 1 event, not 10 edits).
3. **Not for Large Genomes**: The algorithm is $O(n)$, which is theoretically optimal. However, for searching a 3-billion-base-pair human genome, $O(n)$ is still too slow. This is why genomics relies on *sub-linear* heuristic (BLAST) or indexed (BWA/Bowtie) algorithms.

Conclusion: Shift-Or is an excellent choice for a specific bioinformatics task: **repeatedly searching for many *short* (e.g., < 64bp) motifs, like transcription factor binding sites or primer sequences, within a medium-sized text (like a bacterial genome or a single gene region), especially when a small number of mismatches (*k*) must be tolerated.**

9. References

- Baeza-Yates, R., & Gonnet, G. H. (1992). A new approach to text searching. *Communications of the ACM, 35*(10).
- Wu, S., & Manber, U. (1992). Fast text searching allowing errors. *Communications of the ACM, 35*(10).