Two-Piece Gap-Affine Penalties for Partial Order Alignment

Mathematical Framework

Current Gap-Affine Model

The current POASTA implementation uses a standard affine gap penalty model:

$$g_{\text{affine}}(k) = \begin{cases} 0 & \text{if } k = 0 \\ \alpha + \beta \cdot (k-1) & \text{if } k > 0 \end{cases}$$

where: - α = gap opening penalty - β = gap extension penalty
- k = gap length

Two-Piece Gap-Affine Model

The two-piece (bilinear) gap-affine model introduces a breakpoint k_0 where the gap extension penalty changes:

$$g_{\text{two-piece}}(k) = \begin{cases} 0 & \text{if } k = 0 \\ \alpha + \beta_1 \cdot (k-1) & \text{if } 1 \leq k \leq k_0 \\ \alpha + \beta_1 \cdot (k_0 - 1) + \beta_2 \cdot (k - k_0) & \text{if } k > k_0 \end{cases}$$

where: - α = gap opening penalty (same as before) - β_1 = gap extension penalty for short gaps ($k \leq k_0$) - β_2 = gap extension penalty for long gaps ($k > k_0$) - k_0 = breakpoint length (typically 3-4 based on biological data)

Typically, $\beta_2 < \beta_1$ to reduce over-penalization of long gaps.

Biological Motivation

Research has shown that gap length distributions in protein alignments exhibit bilinear behavior with a break at length 3, where: - Short gaps (≤3) follow one linear trend (higher penalty per base) - Long gaps (>3) follow another linear trend (lower penalty per base)

This reflects the biological reality that long indels are relatively common in evolution and shouldn't be over-penalized.

Algorithmic Design for Graph Alignment

State Space Modification

Current POASTA uses a 3-state model: - $M(i, v)$ = Match state at query position i, graph node v - $I(i, v)$ = Insertion state (gap in graph)
- $D(i, v)$ = Deletion state (gap in query)

For two-piece gap-affine, we need to track gap lengths, requiring extended state spaces:

Enhanced State Space

$$S = \{M(i, v), I_1(i, v, l), I_2(i, v, l), D_1(i, v, l), D_2(i, v, l)\}$$

where: - $M(i, v)$ = Match state - $I_1(i, v, l)$ = Insertion state, short gap of length $l \leq k_0$ - $I_2(i, v, l)$ = Insertion state, long gap of length $l > k_0$
- $D_1(i, v, l)$ = Deletion state, short gap of length $l \leq k_0$ - $D_2(i, v, l)$ = Deletion state, long gap of length $l > k_0$

Dynamic Programming Recurrence Relations

Match State

$$M(i, v) = \min \begin{cases} M(i-1, u) + s(q_i, v) \\ I_1(i-1, u, l) + s(q_i, v) \\ I_2(i-1, u, l) + s(q_i, v) \\ D_1(i-1, u, l) + s(q_i, v) \\ D_2(i-1, u, l) + s(q_i, v) \end{cases}$$

where u are predecessors of v in the graph, and $s(q_i, v)$ is the substitution score.

Short Insertion States ($l \leq k_0$)

$$I_1(i, v, l) = \min \begin{cases} M(i-1, v) + \alpha + \beta_1 \cdot (l-1) & \text{if } l = 1 \\ I_1(i-1, v, l-1) + \beta_1 & \text{if } 1 < l \leq k_0 \end{cases}$$

Long Insertion States ($l > k_0$)

$$I_2(i, v, l) = \min \begin{cases} I_1(i-1, v, k_0) + \beta_2 & \text{if } l = k_0 + 1 \\ I_2(i-1, v, l-1) + \beta_2 & \text{if } l > k_0 + 1 \end{cases}$$

Short Deletion States ($l \leq k_0$)

$$D_1(i, v, l) = \min \begin{cases} M(i, u) + \alpha + \beta_1 \cdot (l-1) & \text{if } l = 1 \\ D_1(i, u, l-1) + \beta_1 & \text{if } 1 < l \leq k_0 \end{cases}$$

Long Deletion States ($l > k_0$)

$$D_2(i, v, l) = \min \begin{cases} D_1(i, u, k_0) + \beta_2 & \text{if } l = k_0 + 1 \\ D_2(i, u, l-1) + \beta_2 & \text{if } l > k_0 + 1 \end{cases}$$

Implementation Strategy

1. Data Structure Changes

Enhanced VisitedCell Structure

```
struct VisitedCellTwoPieceAffine {
    visited_m: Score,
    visited_i1: [Score; MAX_SHORT_GAP],  // Short insertions
    visited_i2: FxHashMap<usize, Score>, // Long insertions
    visited_d1: [Score; MAX_SHORT_GAP],  // Short deletions
    visited_d2: FxHashMap<usize, Score>, // Long deletions
}
```

New Gap Scoring Structure

```rust
pub struct GapTwoPieceAffine {
    cost_mismatch: u8,
    cost_gap_open: u8,
    cost_gap_extend_short: u8,  //
    cost_gap_extend_long: u8,   //
    breakpoint: usize,          // k
}
```

2. Modified A* Algorithm

State Representation

```rust
#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub struct AlignmentStateTwoPiece {
    pub query_offset: usize,
    pub graph_node: POANodeIndex,
    pub state_type: AlignmentStateType,
    pub gap_length: usize,
}

#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub enum AlignmentStateType {
    Match,
    InsertionShort,
    InsertionLong,
    DeletionShort,
    DeletionLong,
}
```

Gap Cost Calculation

```rust
impl GapTwoPieceAffine {
    fn gap_cost(&self, gap_length: usize) -> usize {
        if gap_length == 0 {
            return 0;
        }

        if gap_length <= self.breakpoint {
            // Short gap:   +  (k-1)
            self.cost_gap_open as usize +
            (self.cost_gap_extend_short as usize * (gap_length - 1))
        } else {
            // Long gap:   +  (k-1) +  (k-k)
            self.cost_gap_open as usize +
            (self.cost_gap_extend_short as usize * (self.breakpoint - 1)) +
            (self.cost_gap_extend_long as usize * (gap_length - self.breakpoint))
        }
    }
}
```

3. Heuristic Function Updates

The heuristic function needs modification to account for two-piece gap costs:

```rust
fn min_gap_cost_two_piece(&self, remaining_query: usize) -> usize {
    if remaining_query == 0 {
        return 0;
    }

    // Compute minimum possible gap cost for remaining query
    if remaining_query <= self.gap_model.breakpoint {
        self.gap_model.cost_gap_open as usize +
        (self.gap_model.cost_gap_extend_short as usize * (remaining_query - 1))
    } else {
        self.gap_model.cost_gap_open as usize +
        (self.gap_model.cost_gap_extend_short as usize * (self.gap_model.breakpoint - 1)) +
        (self.gap_model.cost_gap_extend_long as usize * (remaining_query - self.gap_model.breakpoint))
    }
}
```

4. Memory Optimization

To manage the increased memory requirements:

1. **Lazy State Creation**: Only create gap states when needed
2. **Gap Length Pruning**: Limit maximum tracked gap length
3. **Blocked Storage**: Extend current blocked storage to handle multiple gap states
4. **State Compression**: Use bit packing for small gap lengths

5. Parameter Selection

Based on biological data, recommended parameters: - $k_0 = 3$ (breakpoint at length 3) - $\beta_1 = 2$ (current gap extend penalty) - $\beta_2 = 1$ (reduced penalty for long gaps) - $\alpha = 6$ (current gap open penalty)

Complexity Analysis

Time Complexity

- Current: $O(|V| \cdot |E| \cdot n)$
- Two-piece: $O(|V| \cdot |E| \cdot n \cdot k_{\text{max}})$

where k_{max} is the maximum gap length tracked.

Space Complexity

- Current: $O(|V| \cdot n)$
- Two-piece: $O(|V| \cdot n \cdot k_{\text{max}})$

The complexity increase is manageable by setting reasonable limits on k_{max} (e.g., 20-50).

Implementation Plan

Phase 1: Core Data Structures

1. Implement `GapTwoPieceAffine` scoring model
2. Create enhanced `VisitedCellTwoPieceAffine` structure
3. Extend `AlignmentState` to track gap lengths

Phase 2: Algorithm Modification

1. Update A* recurrence relations
2. Modify heuristic functions
3. Implement gap length tracking logic

Phase 3: Optimization

1. Add memory optimization strategies
2. Implement state pruning
3. Performance testing and tuning

Phase 4: Integration

1. Update configuration system
2. Add parameter validation
3. Update CLI interface

This design provides a mathematically rigorous and computationally feasible approach to implementing two-piece gap-affine penalties in the POASTA partial order alignment system.