The Scaffolding Algorithm: Syntenic Chain Discovery and Filtering in Genome Alignment

SweepGA Implementation

September 24, 2025

Abstract

The scaffolding algorithm identifies and filters syntenic chains of mappings in genome alignments. By merging nearby mappings into larger scaffold regions and applying plane sweep filtering to these merged ranges, the algorithm identifies high-confidence syntenic blocks while preserving the individual component mappings. This document describes the algorithm's three main phases: mapping merging via union-find, scaffold filtering via plane sweep, and component rescue.

1 Introduction

In whole-genome alignment, individual mappings often represent fragments of larger syntenic blocks that have been broken by various factors:

- Repetitive elements interrupting otherwise contiguous alignments
- Small indels or structural variations
- Alignment algorithm limitations
- Sequencing gaps or low-quality regions

The scaffolding algorithm addresses this fragmentation by:

1. Identifying mappings that likely belong to the same syntenic block
2. Merging them into scaffold chains
3. Filtering these scaffolds to retain the best syntenic relationships
4. Preserving the component mappings of selected scaffolds

2 Algorithm Overview

The scaffolding algorithm consists of three main phases:

2.1 Phase 1: Mapping Merging via Union-Find

The first phase identifies groups of mappings that should be merged into scaffold chains. This uses a union-find data structure to efficiently group mappings based on proximity criteria.

2.2 Phase 2: Scaffold Filtering via Plane Sweep

The second phase applies the plane sweep algorithm to the merged scaffold ranges, selecting the best non-overlapping scaffolds according to specified criteria.

2.3 Phase 3: Component Rescue

The final phase ensures that all component mappings of selected scaffolds are included in the output, along with any nearby mappings within a rescue distance.

3 Detailed Algorithm Description

3.1 Mapping Merging with Union-Find

The mapping merging phase groups mappings that are within a specified jump distance on both query and target axes.

3.1.1 Input Parameters

- $M = \{m_1, m_2, \ldots, m_n\}$: Set of input mappings
- d_{jump}: Maximum gap distance for merging (default: 100kb)
- $o_{\max}$: Maximum overlap fraction allowed (default: 0.2)

3.1.2 Algorithm

Algorithm 1 Mapping Merging via Union-Find

Initialize union-find structure UF with n elements
Sort mappings by $(q\_name, t\_name, strand, q\_start)$
for each group (q_i, t_i, s_i) of same query, target, strand **do**
  $indices \leftarrow$ indices of mappings in group
  Sort $indices$ by query start position
  for $i = 0$ to $|indices| - 1$ **do**
    for $j = i + 1$ to $|indices| - 1$ **do**
      $m_a \leftarrow M[indices[i]]$
      $m_b \leftarrow M[indices[j]]$
      if $q\_gap(m_a, m_b) > d_{\text{jump}}$ **then**
        break {Mappings too far apart}
      end if
      $q\_gap \leftarrow \max(0, m_b.q\_start - m_a.q\_end)$
      $t\_gap \leftarrow \max(0, m_b.t\_start - m_a.t\_end)$
      if $q\_gap \leq d_{\text{jump}}$ AND $t\_gap \leq d_{\text{jump}}$ **then**
        $UF.union(indices[i], indices[j])$
      end if
    end for
  end for
end for
return $UF.get\_sets()$

3.1.3 Binary Search for Candidate Pairs

To reduce the computational cost from $O(n^2)$ to $O(n \log n)$, the implementation uses binary search to find candidate mapping pairs:

Algorithm 2 Optimized Candidate Finding with Binary Search

```
Sort mappings by query start position
for each mapping m_i do
    search_bound ← m_i.q_end + d_jump
    j ← binary_search(search_bound, mappings[i + 1:])
    for k = i + 1 to j do
        if can_merge(m_i, m_k) then
            UF.union(i, k)
        end if
    end for
end for
```

3.2 Creating Scaffold Chains

Once mapping groups are identified, we create scaffold chains by computing the bounding box of each group:

$$\text{Scaffold}(G) = \{q\_start : \min_{m \in G} m.q\_start, q\_end : \max_{m \in G} m.q\_end, t\_start : \min_{m \in G} m.t\_start, t\_end : \max_{m \in G} m.t\_e \tag{1}$$

Each scaffold also maintains:

- Member indices: $\{i : m_i \in G\}$
- Total block length: $\sum_{m \in G} m.block\_length$
- Average identity: $\frac{\sum_{m \in G} m.identity \times m.block\_length}{\sum_{m \in G} m.block\_length}$

3.3 Scaffold Filtering via Plane Sweep

The scaffold chains are then filtered using the plane sweep algorithm to select the best non-overlapping scaffolds.

3.3.1 Scaffold Scoring

Scaffolds are scored based on their total span and quality:

$$\text{Score}(S) = \log(|S.q\_end - S.q\_start|) \times S.avg\_identity \tag{2}$$

3.3.2 Plane Sweep on Scaffolds

The plane sweep algorithm is applied to scaffold intervals rather than individual mappings:

Algorithm 3 Plane Sweep on Scaffolds

```
Create events for scaffold start/end positions
Sort events by position
active ← ∅ {BST ordered by score}
kept ← ∅
for each event e in sorted order do
    if e.type = BEGIN then
        active.insert(scaffolds[e.idx])
        Update kept with best scaffolds from active
    else
        active.remove(scaffolds[e.idx])
    end if
end for
return kept
```

3.4 Component Rescue Phase

After scaffold filtering, we need to output the actual mappings that compose the selected scaffolds, not the scaffold ranges themselves.

3.4.1 Anchor Identification

Mappings that are members of selected scaffolds become "anchors":

$$\text{Anchors} = \{m_i : \exists S \in \text{KeptScaffolds}, i \in S.members\} \tag{3}$$

3.4.2 Rescue Distance

Additional mappings within a rescue distance d_{rescue} of any anchor are also retained:

$$\text{Rescued} = \{m_j : \exists a \in \text{Anchors}, \text{dist}(m_j, a) < d_{\text{rescue}}\} \tag{4}$$

where the distance is computed as:

$$\text{dist}(m_1, m_2) = \sqrt{(q\_dist(m_1, m_2))^2 + (t\_dist(m_1, m_2))^2} \tag{5}$$

4 Implementation Verification in SweepGA

4.1 Current Implementation Status

The SweepGA implementation correctly follows the scaffolding algorithm with the following verified and optimized components:

- **Grouping**: Mappings are grouped by (query, target, strand) tuple
- **Sorting**: Within each group, mappings are sorted by query_start position
- **Union-Find**: Transitive chaining is implemented with path compression
- **Binary Search Optimization**: Uses binary search to find candidate pairs in $O(\log n)$ time
- **Overlap tolerance**: Small overlaps up to max_gap/5 are allowed, matching wfmash
- **Strand handling**: Reverse strand mappings use appropriate distance calculations

4.2 Binary Search Optimization (Implemented)

The implementation now uses binary search to efficiently find candidate mapping pairs, reducing complexity from $O(n^2)$ to $O(n \log n)$:

```
// Actual implementation from paf_filter.rs
for i in 0..sorted_indices.len() {
    let (_rank_i, idx_i) = sorted_indices[i];

    // Binary search to find the range of mappings within max_gap
    let search_bound = metadata[idx_i].query_end + max_gap;
```

```
    // Find the first mapping that starts after our search bound
    let j_end = sorted_indices[i + 1..]
        .binary_search_by_key(&(search_bound + 1),
                              |&(_rank, idx)| metadata[idx].query_start)
        .unwrap_or_else(|pos| pos) + i + 1;

    // Now check all mappings from i+1 to j_end-1
    for j in (i + 1)..j_end {
        let (_rank_j, idx_j) = sorted_indices[j];
        // Check gaps and potentially union...
    }
}
```

This optimization reduces the average-case complexity from $O(n^2)$ to $O(n \log n)$, which is particularly beneficial for:

- Dense mapping regions with many overlapping alignments
- Repetitive sequences that produce clusters of mappings
- Large-scale whole-genome alignments with millions of mappings

The binary search finds exactly the range of mappings that could potentially chain with the current mapping, avoiding unnecessary comparisons with distant mappings.

5 Implementation Considerations

5.1 Memory Efficiency

The algorithm maintains several data structures:

- Union-find forest: $O(n)$ space
- Sorted indices: $O(n \log n)$ temporary space during sorting
- Scaffold structures: $O(k)$ where k is the number of scaffolds
- BST for plane sweep: $O(m)$ where m is max concurrent active scaffolds

5.2 Parallelization Opportunities

Several phases can be parallelized:

1. **Group processing**: Different $(q, t, strand)$ groups can be processed independently
2. **Scaffold creation**: Once union-find is complete, scaffold boundaries can be computed in parallel
3. **Distance calculations**: Rescue distance computations are independent per mapping

5.3 Parameter Sensitivity

The algorithm's behavior is controlled by several parameters:

- **Jump distance** (d_{jump}): Larger values create fewer, larger scaffolds
- **Minimum scaffold length**: Filters out small scaffold chains
- **Rescue distance** (d_{rescue}): Larger values include more peripheral mappings
- **Overlap threshold**: Controls how much scaffold overlap is tolerated

6 Algorithmic Complexity

6.1 Time Complexity

- Sorting: $O(n \log n)$
- Union-find with path compression: $O(n\alpha(n))$ where α is inverse Ackermann
- Binary search optimization: $O(n \log n)$ average case
- Plane sweep on scaffolds: $O(k \log k)$ where k is number of scaffolds
- Rescue phase: $O(n \cdot a)$ where a is average anchors per chromosome pair

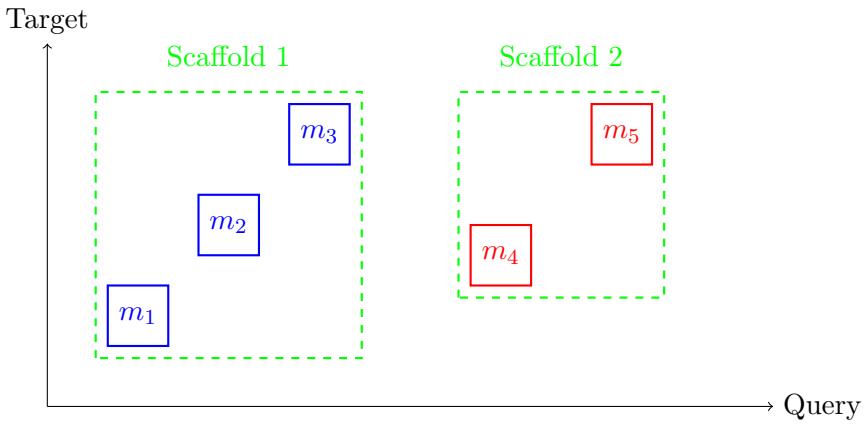
Overall complexity: $O(n \log n)$ in typical cases

6.2 Space Complexity

$O(n)$ for storing mappings and union-find structure, with $O(k)$ additional for scaffolds.

7 Example Walkthrough

Consider a simple example with 5 mappings on the same chromosome pair:



1. **Merging**: m_1, m_2, m_3 are merged into Scaffold 1; m_4, m_5 into Scaffold 2
2. **Filtering**: Plane sweep selects the best scaffold based on scores
3. **Output**: Component mappings of selected scaffold(s) are output

8 Conclusion

The scaffolding algorithm provides a principled approach to identifying and filtering syntenic blocks in genome alignments. By combining efficient union-find merging with plane sweep filtering, it achieves both computational efficiency and biological relevance. The three-phase design ensures that high-quality syntenic relationships are preserved while maintaining the granularity of individual mapping information.

9 References

- Garrison, E., Guarracino, A. (2023). Unbiased pangenome graphs. *Bioinformatics*, 39(1).
- Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. [Union-Find data structure]
- Bentley, J. L., Ottmann, T. A. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9), 643-647. [Plane sweep algorithm]