

Algorithmic Flow: KNN Graph Construction (get_graph)

Original vs. evolved strategy for batch-balanced nearest-neighbor graph construction

Original BBKNN
All-vs-all batch query

1. For each batch b :
Build tree index on X_b

2. For each target batch b_{to} :
1. Build tree on $X_{b_{to}}$
2. Query *all* N cells against b_{to}
3. Get k neighbors per cell from b_{to}
4. Store in columns $[b \cdot k, (b+1) \cdot k)$
Every cell queries every batch —
no cross-batch filtering or validation

3. Concatenate: $k_{total} = k \times B$
neighbors per cell (all batches equal weight)

4. Sort all neighbors by distance

$O(N \times B)$ queries;
many spurious cross-batch edges;
no within-batch vs. cross-batch distinction

bbknn_optimized
Within-batch backbone + MNN cross-batch

1. Within-batch backbone:
For each batch b : query cells of b
against tree of b only $\rightarrow k_{in}$ neighbors
Preserves local manifold structure
and cell-type neighborhoods

2. Cross-batch candidates:
For each target batch b_{to} : query all N
cells against b_{to} tree $\rightarrow k'$ candidates
Expansion factor 2–3 \times for MNN pool

3. MNN filter:
For each cell i in batch b_i :
1. Get candidates j in batch b_{to}
2. Check: is i in j 's top- k' of b_i ?
3. Keep only **mutual** pairs (i, j)
4. De-duplicate; sort by distance
Removes one-way false positive
cross-batch connections

4. Cap cross-batch at $0.7 \times k_{cross}$;
concatenate with within-batch backbone;
sort by distance

MNN ensures bidirectional support;
conservative cross-batch budget protects biology

Algorithmic Flow: Connectivity Weighting & Graph Trimming

Edge weighting (COMPUTE_CONNECTIVITIES) and graph sparsification (TRIMMING)

Original BBKNN
UMAP fuzzy set + threshold trim

1. Sparse matrix construction (COO):

Nested Python loops:
`for i in range(N):`
`for j in range(k):`
`rows[i*k+j] = i; cols[...] = idx[i,j]`
 $O(N \cdot k)$ Python iterations

2. Connectivity weighting:

UMAP `fuzzy_simplicial_set`
→ local σ , set-union symmetrization
Batch-agnostic: treats within-batch and cross-batch edges identically; can amplify spurious cross-batch edges

3. Trimming:

1. For each row: find threshold at rank k_{trim}
2. For each row: zero values below threshold
3. Transpose matrix
4. Repeat step 2 on transposed rows

Two-pass filter with Python per-row loops; no mutuality awareness — mutual edges treated same as one-way edges

UMAP can amplify false-positive cross-batch edges; threshold trim may discard biologically important mutual connections

bbknn_optimized
Self-tuning kernel + mutuality-aware trim

1. Sparse matrix construction (COO):

`np.repeat(arange(N), k) + reshape(-1)`
Vectorized mask for invalid (-1) and self edges
Eliminates nested Python loops entirely

2. Batch-aware kernel weighting:

New function: `compute_connectivities_batchaware_kernel`
Self-tuning kernel (Zelnik-Manor & Perona):

$$w_{ij} = \exp\left(-d_{ij}^2 / (\sigma_i \cdot \sigma_j + \varepsilon)\right)$$

Separate scales $\sigma_{\text{in}}(i)$, $\sigma_{\text{cross}}(i)$

selected based on batch membership of j

Cross-batch damping: $w \leftarrow \lambda \cdot w$; $\lambda = 0.5$

Prevents cross-batch edges from dominating local structure

Symmetrize: $W_{\text{union}} = W + W^{\top} - W \odot W^{\top}$
(UMAP-style set union on weights)

3. Mutuality-aware trimming:

1. Detect bidirectional edges via

$$M = \text{sign}(W) \odot \text{sign}(W^{\top})$$

2. `Priority = data + mutual_boost · M`

Mutual edges get large additive boost

3. `np.argpartition` top- k by priority

4. Max-symmetrize: $W_{\text{out}} = \max(W, W^{\top})$

Mutual (bidirectional) edges preferentially preserved

Detailed Pseudocode Comparison

Algorithmic differences in KNN graph construction (GET_GRAPH) and connectivity weighting (COMPUTE_CONNECTIVITIES)

1. KNN Graph Construction (get_graph)

Algorithm 1a: Original BBKNN — All-vs-All Batch Query

Input: PCA matrix $X \in \mathbb{R}^{N \times d}$, batch labels, k neighbors per batch, B batches

Output: KNN indices and distances with $k_{\text{total}} = k \times B$ neighbors per cell

```
1: Allocate  $\text{knn\_idx} \in \mathbb{Z}^{N \times kB}$ ,  $\text{knn\_dist} \in \mathbb{R}^{N \times kB}$ 
2: for each batch  $b_{\text{to}} \in \{1, \dots, B\}$  do
3:   Build tree index on  $X_{b_{\text{to}}}$ 
4:    $(d, \text{idx}) \leftarrow \text{QUERYTREE}(X_{\text{all}}, \text{tree}_{b_{\text{to}}}, k)$ 
5:    $\text{knn\_idx}[:, b \cdot k : (b+1) \cdot k] \leftarrow \text{GlobalIdx}(\text{idx})$ 
6:    $\text{knn\_dist}[:, b \cdot k : (b+1) \cdot k] \leftarrow d$ 
7: end for
8: Sort each row of  $\text{knn\_idx}$ ,  $\text{knn\_dist}$  by distance
```

▷ All N cells query each batch

Algorithm 1b: bbknn_optimized — Within-batch Backbone + MNN Cross-batch

Input: PCA matrix X , batch labels, k_{in} (within-batch neighbors), expansion factor

Output: KNN with $k_{\text{total}} = k_{\text{in}} + k_{\text{cross}}$ neighbors per cell ($k_{\text{cross}} = \lfloor 0.7 \times (B-1) \times k_{\text{in}} \rfloor$)

```
1: Phase 1: Within-batch backbone
2: for each batch  $b$  do
3:   Build tree on  $X_b$ ; query cells of  $b$  against own tree  $\rightarrow k_{\text{in}}$  neighbors
4:    $\text{knn\_in}[\text{cells}_b] \leftarrow$  global indices and distances
5: end for
6: Phase 2: Cross-batch MNN candidates ( $k' = \text{expansion} \times k_{\text{in}}$ )
7: for each target batch  $b_{\text{to}}$  do
8:   Build tree on  $X_{b_{\text{to}}}$ ; query all  $N$  cells  $\rightarrow k'$  candidates per cell
9:   Store in  $\text{cross}[b_{\text{to}}]$ 
10: end for
11: Phase 3: MNN filtering
12: for each cell  $i$  with batch  $b_i$  do
13:   for each other batch  $b_{\text{to}} \neq b_i$  do
14:     Get candidates  $j \in \text{cross}[b_{\text{to}}][i]$  with distances
15:     Mutual test: is  $i \in \text{cross}[b_i][j]$ ?
16:     Keep only mutual pairs  $(i, j)$ 
17:   end for
18:   De-duplicate by keeping closest; sort by distance; cap at  $k_{\text{cross}}$ 
19: end for
20: Concatenate within-batch + cross-batch; sort by distance
```

▷ Preserves local manifold structure

▷ Indexed by target batch

▷ Keep only mutual nearest neighbors

▷ Bidirectional support required

2. Connectivity Weighting & Trimming

Algorithm 2a: Original BBKNN — UMAP Fuzzy Set + Threshold Trim

Input: KNN indices & distances ($N \times k_{\text{total}}$)

Output: Sparse connectivity matrix W

```
1: COO construction: nested Python loops over  $N \times k$  entries
2:  $W \leftarrow \text{UMAPFUZZYSIMPLICIALSET}(\text{knn\_idx}, \text{knn\_dist})$ 
3: Trimming: For each row: find threshold at rank  $k_{\text{trim}}$ ; zero below; transpose; repeat
```

▷ $O(Nk)$ Python iterations

▷ Batch-agnostic weighting

Algorithm 2b: bbknn_optimized — Self-tuning Kernel + Mutuality-aware Trim

Input: KNN indices & distances, batch labels

Output: Sparse connectivity matrix W

- 1: **COO:** $\text{np.repeat}(\text{arange}(N), k) + \text{reshape}(-1)$ (vectorized) ▷ No Python loops
- 2: — **Self-tuning kernel (Zelnik-Manor & Perona)** —
- 3: Classify each edge as within-batch or cross-batch via batch labels
- 4: **for** each cell i **do**
- 5: $\sigma_{\text{in}}(i) \leftarrow k\text{th closest within-batch neighbor distance}$
- 6: $\sigma_{\text{cross}}(i) \leftarrow k\text{th closest cross-batch neighbor distance}$
- 7: **end for**
- 8: $\sigma_i, \sigma_j \leftarrow \text{select } \sigma_{\text{in}} \text{ or } \sigma_{\text{cross}} \text{ based on batch membership of edge } (i, j)$
- 9: $w_{ij} \leftarrow \exp(-d_{ij}^2 / (\sigma_i \cdot \sigma_j + \varepsilon))$ ▷ Self-tuning Gaussian kernel
- 10: **If** cross-batch edge: $w_{ij} \leftarrow \lambda \cdot w_{ij}$; $\lambda = 0.5$ ▷ Cross-batch damping
- 11: Symmetrize: $W \leftarrow W + W^\top - W \odot W^\top$ ▷ UMAP-style fuzzy union
- 12: — **Mutuality-aware trimming** —
- 13: Mutual mask: $M \leftarrow \text{sign}(W) \odot \text{sign}(W^\top)$ ▷ Detect bidirectional edges
- 14: Priority $\leftarrow \text{data} + \text{mutual_boost} \cdot M$ ▷ Mutual edges get priority
- 15: **np.argpartition:** keep top- k_{trim} per row by priority
- 16: Max-symmetrize: $W_{\text{out}} \leftarrow \max(W_{\text{trimmed}}, W_{\text{trimmed}}^\top)$

3. Summary of Key Algorithmic Differences

Component	Original	bbknn_optimized
Graph strategy	All-vs-all: every cell queries every batch; $k_{\text{total}} = k \times B$	Within-batch backbone (k_{in}) + MNN-constrained cross-batch ($0.7 \times \text{cap}$)
Cross-batch filter	None (all edges kept)	Mutual nearest neighbor test: keep only bidirectional matches
Edge weighting	UMAP fuzzy simplicial set (batch-agnostic)	Self-tuning kernel $w = e^{-d^2/(\sigma_i \sigma_j)}$ with cross-batch damping $\lambda=0.5$
Scale estimation	UMAP internal (single σ per cell)	Separate σ_{in} , σ_{cross} per cell based on within/cross neighbor distances
Trimming	Threshold per row, 2-pass transpose; Python loops	Mutual-first priority + <code>argpartition</code> top- k + max-symmetrize
COO construction	Nested Python loops $O(Nk)$	Vectorized <code>np.repeat/reshape</code>
Annoy query	List append per cell	Preallocated numpy arrays filled in-place

4. Performance Comparison

Metric	Original	bbknn_optimized	Δ
Batch mixing score	0.6430	0.6759	+5.1%
Bio conservation score	0.6036	0.7184	+19.0%
Speed score	0.4837	0.8020	+65.8%
Execution time	1.068s	0.247s	$\times 4.3$
Evolution generation	0	7	—
Evolution order	0	225	—

5. Why bbknn_optimized Improved

bbknn_optimized: Comprehensive improvement via MNN selection + batch-aware kernel + mutual trimming

- Bio conservation (+19.0%):** The most significant improvement comes from three complementary mechanisms that protect biological structure:
 - Within-batch backbone** preserves local manifold structure and cell-type neighborhoods by maintaining k_{in} neighbors from each cell's own batch.
 - MNN filtering** removes spurious one-way cross-batch edges that would collapse distinct cell types. Only edges with bidirectional support (cell i is a neighbor of j and j is a neighbor of i) are retained.
 - Self-tuning kernel** with separate $\sigma_{\text{in}}/\sigma_{\text{cross}}$ scales and $\lambda=0.5$ cross-batch damping prevents cross-batch edges from dominating local structure in the connectivity graph.
- Batch mixing (+5.1%):** MNN-validated cross-batch edges are more biologically meaningful than brute-force all-vs-all connections. The mutuality-aware trimming preferentially preserves bidirectional (mutual) edges, which represent genuine cross-batch correspondences. The conservative $0.7 \times \text{cap}$ on cross-batch edges focuses the budget on high-quality connections.
- Speed ($\times 4.3$):** Vectorized COO construction via `np.repeat/reshape` eliminates $O(Nk)$ Python loop iterations. Preallocated numpy arrays for Annoy queries reduce allocation overhead. The batch-aware kernel replaces UMAP's internal `fuzzy_simplicial_set` computation (which involves iterative optimization of local scales) with a direct self-tuning kernel that computes weights in a single vectorized pass.