DNA2Graph Technical Manual

Federico Chinello

Version 1.0.1 - May 27, 2026

Contents

1 Introduction **2**

2 Pipeline **2**
2.1 Segmentation Module 3
2.1.1 Validity Mask Computation 4
2.1.2 Pre-processing 4
2.1.3 Bilateral Filtering 5
2.1.4 Unsharp Masking 5
2.1.5 Hysteresis Thresholding 5
2.1.6 Area-based Pruning (I) 6
2.1.7 Morphological Closure 6
2.1.8 Area-based Pruning (II) 6
2.2 Error Correction Module 7
2.2.1 Mask-to-Graph Conversion 8
2.2.2 Repair Connectivity (I) 9
2.2.3 Prune Short Cycles 9
2.2.4 Prune Short Branches 12
2.2.5 Prune by Thickness 14
2.2.6 Repair Connectivity (II) 14
2.2.7 Prune Small Components 15
2.2.8 Graph-to-Mask Conversion 15

3 Spatial Graph Export Format **15**

1 Introduction

DNA2Graph is an open-source image analysis software for the automated segmentation and classification of DNA molecules in electron microscopy images obtained by rotary shadowing. It produces multiple outputs, including binary segmentation masks, ImageJ-compatible ROI annotations, quantitative length measurements, and spatial graph representations for downstream computational analysis.

This document provides a technical description of the algorithms and core logic underlying DNA2Graph. It is intended for technical users who wish to understand the software's internal functioning and configuration parameters. Throughout the document, user-adjustable parameters are denoted in `monospace font`.

2 Pipeline

DNA2Graph is designed as a two-stage processing framework. The pipeline, illustrated in Figure 1, consists of (i) a **Segmentation Module**, which isolates DNA structures from the background, and (ii) a graph-based **Error Correction Module**, which refines these structures to ensure continuity and biological plausibility.

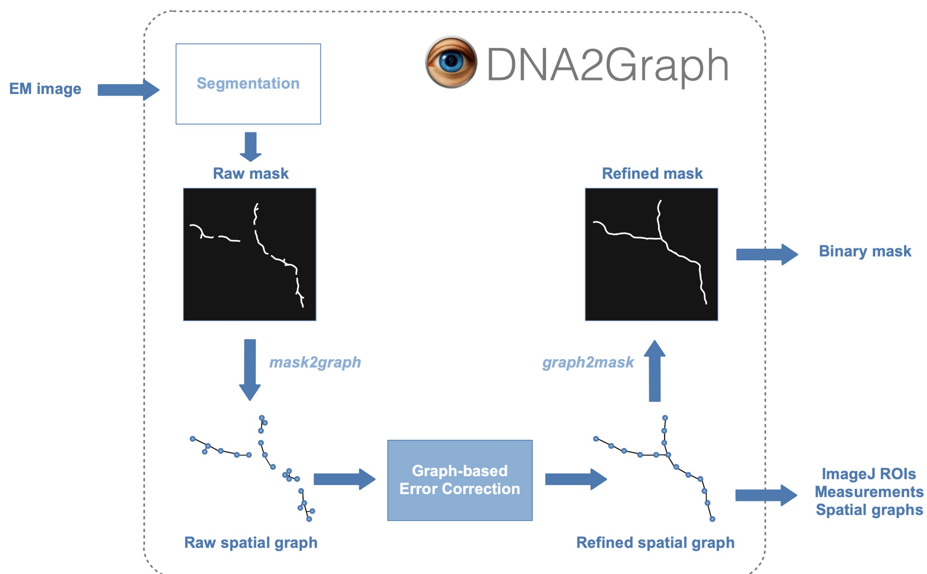

Figure 1: **DNA2Graph.** The DNA2Graph pipeline consists of a Segmentation Module that generates an initial mask, followed by a graph-based Error Correction Module that refines it.

2.1 Segmentation Module

The Segmentation Module processes the input image to produce a coarse binary mask that separates DNA from the background. A high-level summary of the individual steps is provided in Table 1, while the remainder of this section presents a detailed description of each step.

Table 1: **Segmentation Module.**

Step	**Purpose**
Validity mask computation	Compute a binary mask that identifies valid regions of the image, used throughout the pipeline to exclude corrupted or irrelevant areas and prevent them from influencing segmentation.
Preprocessing	Enhance contrast and standardize intensity distributions to reduce inter-image variability; convert 16-bit images to 8-bit and invert intensities so DNA molecules exhibit higher grayscale values than the background.
Bilateral Filter	Reduce background noise while preserving edge information, ensuring that thin DNA molecules remain well-defined for subsequent processing.
Unsharp Masking	Enhance edges and local contrast to better highlight DNA structures against the background.
Hysteresis Thresholding	Generate a binary segmentation by separating DNA molecules from the background.
Area-based Pruning (1)	Remove small, spurious segmented regions based on a predefined area threshold.
Morphological Closing	Fill small gaps and discontinuities between segmented regions to improve the structural continuity of DNA molecules.
Area-based Pruning (2)	Remove small, spurious segmented regions based on a predefined area threshold, which can differ from the one used in the first pruning stage.

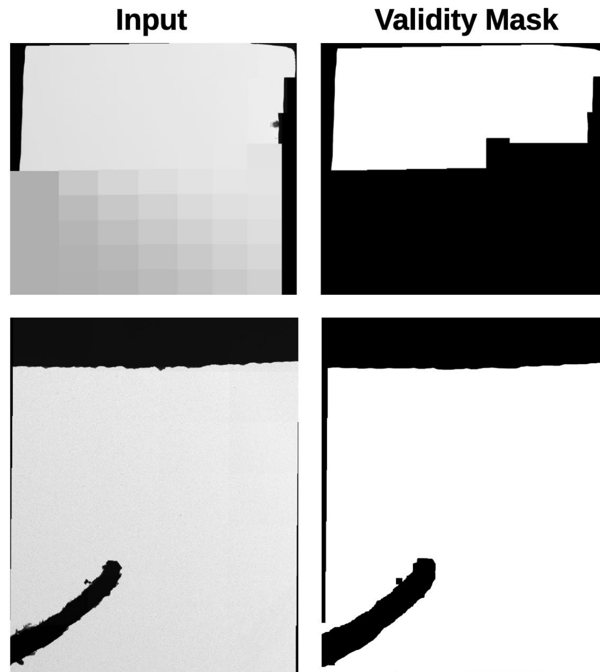

Figure 2: **Validity Mask Computation**. DNA2Graph automatically detects invalid regions in the input image and disregards them throughout the pipeline, so that they cannot influence the final output.

2.1.1 Validity Mask Computation

A validity mask is computed to exclude regions outside the sample or corrupted by image acquisition errors (see Figure 2). These regions are ignored throughout the pipeline. Invalid regions are identified by low local intensity variation. A pixel is marked as invalid if the intensity range within a square neighborhood of side length `block_size` is below `range_factor` times the global image intensity range. The resulting invalid regions are then expanded by `safety_margin` pixels.

2.1.2 Pre-processing

Several pre-processing steps are applied:

- **Contrast enhancement** using percentile-based normalization: the lower and upper 0.175% percentiles of pixel intensities are clipped and the remaining intensity range is linearly stretched to span the full dynamic range of the image.

- **16-bit to 8-bit conversion**, if the input image is 16-bit.
- **Histogram matching** with respect to a reference histogram obtained from a representative image.
- **Inversion**: the image complement is computed (255 - *image*).

2.1.3 Bilateral Filtering

The bilateral filter is a noise-reduction technique that preserves edges by combining spatial proximity and intensity similarity. This allows smoothing of homogeneous regions while maintaining sharp boundaries.

- `bilateral_diameter`: Diameter of the pixel neighborhood used during filtering.
- `bilateral_sigma_color`: Standard deviation in the intensity domain; controls how dissimilar colors are mixed.
- `bilateral_sigma_space`: Standard deviation in the spatial domain; controls the influence of distant pixels.

2.1.4 Unsharp Masking

Unsharp masking enhances image sharpness and improves local contrast by amplifying high-frequency components.

- `high_boost_radius`: Radius of the Gaussian blur applied before detail extraction.
- `high_boost_amount`: Scaling factor controlling the strength of the sharpening effect.

2.1.5 Hysteresis Thresholding

This step binarizes the image using a dual-threshold approach. The lower threshold L and upper threshold H are computed from percentiles of valid pixel intensities.

- L: `hysteresis_low_pct` percentile of valid intensities.
- H: `hysteresis_high_pct` percentile of valid intensities.

Pixels with intensity above H are marked as strong, those below L are discarded, and intermediate pixels are retained only if connected (directly or indirectly) to strong pixels.

2.1.6 Area-based Pruning (I)

Connected components with area smaller than a predefined threshold are removed.

- `min_area_1`: Minimum area required for a component to be retained.

2.1.7 Morphological Closure

Morphological closing is applied to fill small holes and connect nearby regions.

- `closure_kernel_size`: Size of the structuring element used for the closing operation.

2.1.8 Area-based Pruning (II)

A second pruning step removes remaining small components after morphological processing.

- `min_area_2`: Minimum area required for a component to be retained.

2.2 Error Correction Module

The binary mask produced by the Segmentation Module is used to extract a spatial graph representation of the DNA molecules. This spatial graph is then refined by an Error Correction Module consisting of a cascade of custom graph-processing algorithms. These algorithms are summarized at a high level in Table 2 and described in detail in the remainder of this section. Finally, the cleaned graph representation is converted back into image form.

Table 2: **Error Correction Module.**

Step	**Purpose**
Mask-to-Graph Conversion	Convert each segmented DNA molecule in the mask into a spatial graph representation, where nodes correspond to pixels along the molecule with their (x, y) coordinates, and edges encode connectivity.
Repair Connectivity (1)	Operate on spatial graph representations to address fragmentation caused by segmentation errors in thin DNA structures, by selectively adding edges between relevant node pairs to restore connectivity.
Prune Short Cycles	Operate on spatial graph representations to remove spurious short cycles, using a predefined cycle-length threshold.
Prune Short Branches	Operate on spatial graph representations to remove spurious short branches, using a predefined branch-length threshold.
Prune by Thinkness	Clean the spatial graph representations by removing nodes associated with abnormally thick regions, leveraging the prior that DNA structures are thin and thicker regions are typically spurious.
Repair Connectivity (2)	Further refine spatial graph representations by performing a second connectivity repair step, using different parameters from the first repair stage.
Prune Small Components	Discard small spatial graph representations that are likely spurious.
Graph-to-Mask Conversion	Convert the refined spatial graph representations back into mask form for downstream analysis or visualization.

2.2.1 Mask-to-Graph Conversion

A spatial graph representation is extracted from the binary mask generated by the Segmentation Module. The mask is first skeletonized using the Zhang-Suen thinning algorithm. Skeletonization yields a one-pixel-wide representation that preserves the shape and connectivity of each DNA molecule. The skeletonized mask is then converted into a spatial graph $G = (V, C, E)$, where V is the set of nodes, C denotes their spatial coordinates, and E is the set of edges connecting node pairs $(u, v) \in V$.

Graph construction proceeds by mapping each skeleton pixel to a node, with its (x, y) image coordinates stored as spatial attributes (Figure 3). Edges are introduced between nodes whose corresponding pixels are immediate neighbors in the image grid, following the standard notion of 8-connectivity. Each molecule is represented as a connected component in the spatial graph.

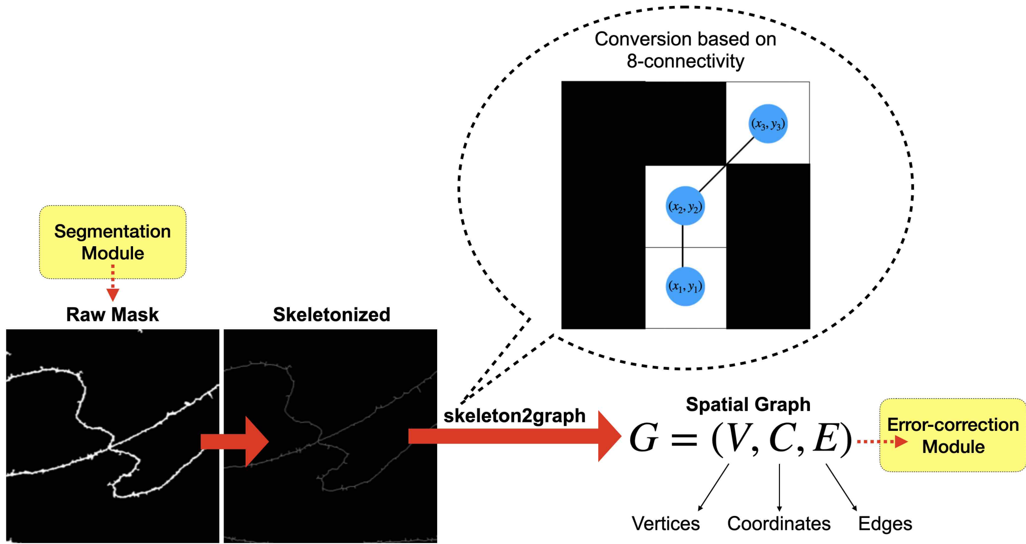

Figure 3: **Skeleton-to-graph Conversion.** To convert a skeleton into a spatial graph, each pixel that lies on the skeleton is mapped to a corresponding node in the graph, with its (x, y) coordinates stored as node attributes. An undirected edge is added between two nodes if their corresponding pixels in the skeleton are 8-connected, i.e., adjacent in the horizontal, vertical, ordiagonal direction.

2.2.2 Repair Connectivity (I)

The algorithm restores connectivity in the spatial graph representation by adding edges between nodes that are close in Euclidean space but far apart in graph distance. It operates on endpoint nodes (i.e., nodes of degree 1) as sources, since these are natural candidates for broken connections.

For each endpoint, a spatial neighborhood is constructed as the set of all nodes within a given Euclidean radius, forming an initial candidate set. This set is then filtered to retain only nodes whose shortest-path distance from the endpoint exceeds a specified threshold. If the resulting set is non-empty, the endpoint is connected to the candidate that is closest in Euclidean space.

Optionally, the candidate set can be restricted to endpoint nodes only, so that connections are introduced exclusively between endpoints.

Examples:

- *Disconnected components.* An endpoint e and a node v are spatially close (their Euclidean distance is within d_e) and belong to distinct connected componets (their shortest-path distance is ∞). An edge (e, v) may be added, merging the two components.

- *Topologically close nodes.* An endpoint e and a node v are spatially close (their Euclidean distance is within d_e) and topologically close (their shortest-path distance is within d_g). No edge is added.

2.2.3 Prune Short Cycles

The algorithm simplifies the spatial graph representation by iteratively contracting short cycles. At each iteration, it first computes a cycle basis and selects the cycles whose length is below a prescribed threshold. These short cycles are then processed one at the time, ensuring that overlapping cycles are not modified within the same iteration. Each cycle is replaced by a single node, whose spatial coordinates are defined as the centroid of the coordinates of its neighboring nodes. If a cycle has no neighboring nodes, it is simply removed. If replacing a cycle would introduce a node with excessively high degree, the entire component is considered unreliable and is discarded.

The procedure is repeated until no further contractions are possible or until a maximum number of iterations, specified as a parameter of the algorithm, is reached. The algorithm operates independently on each connected component, since different components may require a different number of iterations.

Algorithm 1 Repair Connectivity

Require: Spatial graph $G = (V, E)$
Require: Euclidean distance threshold d_e (`max_euclidean_distance`)
Require: Graph-distance threshold d_g (`min_graph_distance`)
Require: Endpoint restriction flag (`restrict_to_endpoints`)

1: $S \leftarrow \{ e \in V \mid \deg(e) = 1 \}$ ▷ Sources
2: **if** endpoint restriction is enabled **then**
3: $\quad C \leftarrow S$
4: **else**
5: $\quad C \leftarrow V$
6: **end if**
7: **for all** $e \in S$ **do**
8: $\quad N_e \leftarrow \{ v \in C \mid \|e - v\| \leq d_e \}$ ▷ Euclidean neighborhood
9: $\quad T_e \leftarrow \{ v \in N_e \mid \mathrm{dist}_G(e, v) \geq d_g \}$ ▷ Graph-distance filtering
10: $\quad$ **if** $T_e \neq \emptyset$ **then**
11: $\quad\quad v^* \leftarrow \arg\min_{v \in T_e} \|e - v\|$
12: $\quad\quad E \leftarrow E \cup \{(e, v^*)\}$
13: $\quad$ **end if**
14: **end for**
15: **return** G

Algorithm 2 Prune Short Cycles

Require: Spatial graph $G = (V, E)$
Require: Minimum cycle length $\ell_{\min}$ (`min_cycle_length`)
Require: Maximum number of iterations $k_{\max}$
Require: Wiring cap $w_{\max}$

```
 1: Decompose G into connected components 𝒞 = {G_1, ..., G_m}
 2: for all G_i ∈ 𝒞 do
 3:     for k = 1 to k_max do
 4:         B ← a cycle basis of G_i
 5:         S ← { c ∈ B | |c| < ℓ_min }                    ▷ Short cycles
 6:         if S = ∅ then
 7:             break
 8:         end if
 9:         R ← ∅                                           ▷ Nodes removed
10:         for all c ∈ S do
11:             if c ∩ R ≠ ∅ then
12:                 continue                    ▷ Skip overlapping cycles
13:             end if
14:             N_c ← { v ∈ V(G_i) \ c | ∃u ∈ c with (u, v) ∈ E(G_i) }
15:             Remove all nodes in c from G_i
16:             R ← R ∪ c
17:             if N_c = ∅ then
18:                 continue                    ▷ Discard isolated cycle
19:             end if
20:             if |N_c| > w_max then
21:                 Clear G_i                   ▷ Discard corrupted component
22:                 break
23:             end if
24:             v* ← centroid(N_c)
25:             Add v* to G_i
26:             Add edges { (v*, v) | v ∈ N_c } to G_i
27:         end for
28:     end for
29: end for
30: G ← recomposition of all components in 𝒞
31: return G
```

2.2.4 Prune Short Branches

The algorithm simplifies the spatial graph representation by iteratively pruning short branches. A branch is defined as a maximal path starting from an endpoint and traversing only nodes of degree 2, until a node of degree different from 2 is reached. This terminal node is referred to as the *joint* of the branch, i.e., the node at which the branch attaches to the rest of the graph.

The algorithm proceeds iteratively: at each iteration, it identifies all branches whose length is below a prescribed threshold. These short branches are then processed in increasing order of length, so that the shortest and most likely spurious branches are removed first. The algorithm prevents the removal of multiple branches that share the same joint within a single iteration. This restriction is essential because a branch that is classified as short at the beginning of an iteration may no longer satisfy this condition after a neighboring branch has been removed.

For instance, consider a Y-shaped structure in which two short branches meet at a joint, while a third segment (the "trunk" of the Y) connects the joint to the rest of the graph. If one of the two short branches is removed, the joint loses one incident edge and its degree drops from 3 to 2, so that it is no longer a junction. As a result, the remaining short branch no longer terminates at that node, but instead continues through it and forms a longer path together with the trunk; consequently, it may no longer be classified as short. For this reason, once a branch incident to a given joint is removed, all other branches attached to that joint are deferred to the next iteration, where they are re-extracted and re-evaluated on the updated graph.

The procedure continues until there are no remaining short branches or until a maximum number of iterations, specified as a parameter of the algorithm, is reached. The algorithm operates independently on each connected component, since different components may require a different number of iterations.

Algorithm 3 Prune Short Branches

Require: Spatial graph $G = (V, E)$
Require: Minimum branch length $\ell_{\min}$ (`min_branch_length`)
Require: Maximum number of iterations $k_{\max}$

```
1: Decompose G into connected components C = {G_1, ..., G_m}
2: for all G_i ∈ C do
3:     for k = 1 to k_max do
4:         S ← { v ∈ V(G_i) | deg(v) = 1 }                ▷ Endpoints
5:         B ← ∅                          ▷ Branches marked for removal
6:         for all e ∈ S do
7:             Extract the branch b_e starting at e
8:             if |b_e| < ℓ_min then
9:                 B ← B ∪ {b_e}
10:            end if
11:        end for
12:        if B = ∅ then
13:            break
14:        end if
15:        J ← ∅                                       ▷ Processed joints
16:        for all b ∈ B in increasing order of length do
17:            v_b ← joint of b
18:            if v_b ∉ J then
19:                Remove all nodes of b except v_b
20:                J ← J ∪ {v_b}
21:            end if
22:        end for
23:    end for
24: end for
25: Recombine all components into G
26: return G
```

2.2.5 Prune by Thickness

DNA molecules are thin filaments and thick regions in the mask produced by the Segmentation Module are typically associated with noise or artifacts. Each node/pixel in the spatial graph representation carries a thickness attribute, which captures the thickness of the binary mask at that pixel. The Prune by Thickness algorithm simplifies the spatial graph by removing nodes that exhibit excessive thickness.

Nodes whose thickness exceeds a prescribed threshold (`max_thickness`) are considered candidates for removal. However, such nodes are only removed if they are not structurally important, as their removal could otherwise disconnect relevant parts of the graph.

A node is defined as structurally important if it plays a significant role in the connectivity of the graph. This is the case if either:

- the node belongs to a cycle; or
- the node has at least two structurally important incident edges.

More precisely, an incident edge is considered structurally important if, when followed away from the node, it either:

- reaches nodes at a distance greater than a prescribed depth cutoff (`structurality_depth_cutoff`), indicating that it connects to a sufficiently extended structure; or
- leads to a cycle anywhere in the graph, indicating connection to a loop-like structure.

Requiring at least two such edges ensures that the node is not merely attached to a single significant structure, but instead acts as a connector between multiple meaningful regions of the graph. Nodes that do not satisfy these conditions are considered non-structural and can be safely removed if their thickness exceeds the prescribed threshold.

2.2.6 Repair Connectivity (II)

The spatial graph representation is further refined by a second pass of the Repair Connectivity algorithm, using a different set of parameters from the first pass.

2.2.7 Prune Small Components

The spatial graph representation is pruned by removing connected components whose largest bounding box side is smaller than the user-defined threshold `minimum_bbox_size`, thereby discarding small and likely spurious structures. Here, the bounding box is defined as the smallest axis-aligned rectangle enclosing all nodes of a connected component.

2.2.8 Graph-to-Mask Conversion

Once error correction is complete, the cleaned spatial graph representation is rendered back onto the pixel grid, yielding a binary segmentation mask that retains the improvements from graph-based repair. This is achieved by initializing an empty image canvas and assigning foreground values to the pixel locations corresponding to the coordinates of the graph nodes.

3 Spatial Graph Export Format

The cleaned spatial graph representation generated by the Error Correction Module is exported in a custom HDF5 format, where each connected component, corresponding to a molecule, is stored as a separate group.

A connected component / molecule is represented as a set of *linear walks*, defined as maximal non-branching paths. Each walk is an ordered sequence of nodes in which all intermediate nodes have degree two. Walks are encoded through two datasets:

- The `coords` dataset is a 2D array of shape $(N, 2)$ containing the concatenated coordinates of all walks.
- The `offsets` dataset is a 1D array of length $K + 1$ that partitions `coords` into K individual walks, such that the i-th walk is given by `coords[offsets[i]:offsets[i+1]]`.

The component / molecule label (non-linear, linear, or boundary) is stored as a group attribute.

A molecule is reconstructed as a set of linear walks, i.e., coordinate sequences, by slicing `coords` according to `offsets`:

```
import h5py
import numpy as np

def load_molecule(h5_grp):
        coords = h5_grp['coords'][:]
        offsets = h5_grp['offsets'][:]

        seqs = []
        for i in range(len(offsets) - 1):
                start, end = offsets[i], offsets[i+1]
                seq = coords[start:end]
                seqs.append(seq)

        return seqs
```

Adjacency between walks is recovered by identifying sequences that share the same terminal coordinates:

```
from collections import defaultdict
import numpy as np

def build_adjacency(seqs):
        n_seqs = len(seqs)
        adjacency = np.zeros((n_seqs, n_seqs))

        terminal_map = defaultdict(list)
        for i, seq in enumerate(seqs):
                first = tuple(int(x) for x in seq[0].tolist())
                last = tuple(int(x) for x in seq[-1].tolist())
                terminal_map[first] += [i]
                terminal_map[last] += [i]

        for indices in terminal_map.values():
                for i in range(len(indices)):
                        for j in range(i + 1, len(indices)):
                                adjacency[indices[i], indices[j]] = 1
                                adjacency[indices[j], indices[i]] = 1

        return adjacency
```

The walks can be easily rendered into a binary mask:

```
import cv2
import numpy as np

def render_walks(seqs):
        all_points = np.concatenate(seqs)
        min_y, min_x = np.min(all_points, axis=0)
        max_y, max_x = np.max(all_points, axis=0)
        width = max_x - min_x + 1
        height = max_y - min_y + 1

        canvas = np.zeros((height, width), dtype=np.uint8)
        y = all_points[:, 0] - min_y
        x = all_points[:, 1] - min_x
        canvas[y, x] = 255

        dilation_kernel = np.ones((5, 5), np.uint8)
        canvas = cv2.dilate(canvas, dilation_kernel, iterations=1)

return canvas
```