Vision, Modeling, and Visualization (2026)
T. Leimkühler, T. Weinkauf and M. Campen (Eds.)

Stroke Vectorization via Involution Prediction

Philipp Gerstner, Tanguy Magne and Olga Sorkine-Hornung

ETH Zurich, Zurich, Switzerland
pgerstner@student.ethz.ch, tanguy.magne@inf.ethz.ch, olga.sorkine@inf.ethz.ch

Abstract

Line drawing vectorization converts raster sketches into editable curves and is widely used in animation, illustration and design. A high-quality vectorization must preserve not only the visual appearance of the drawing but also its topology: the individual strokes, their endpoints and the way they connect at intersections. At junctions, existing methods often produce noisy geometry and misrouted branches, reducing the drawing to a collection of short segments, rather than a faithful representation of the original drawing. We propose a vectorization pipeline that separates the recovery of drawing geometry from stroke routing. A learned image-to-image model first predicts a centerline representation of the drawing, from which a planar straight-line graph with single-vertex junctions is built. A second learned model then predicts for each junction the incident branches that continue through it as a single stroke. We formulate this routing decision as the prediction of an involution on half-branches, allowing it to be made jointly over the entire drawing. On a benchmark of 369 professional line drawings, our pipeline recovers substantially more ground-truth intersections and produces fewer oversegmented strokes than four prior baselines, while remaining competitive on standard geometric metrics. By recovering the correct topology, our method yields results that are visually more appealing and easier to edit. Our code is available at github.com/verven/InvolutionStrokeVectorization.

Keywords: line drawings, vectorization, involution, vector graphics

CCS Concepts
• ***Computing methodologies*** → ***Image processing;*** *Shape inference;*

1. Introduction

Vectorizing drawings from images is a long-standing problem in computer graphics, driven by the need to convert raster sketches into editable curves for creative work or engineering design. Although digital drawing tools are widely available, popular ones such as Procreate [Sav26] are raster-based, and many artists still prefer to work with physical media such as pen and paper, making raster-to-vector conversion practically relevant. *Line drawings* are images made of distinct, well-separated strokes, whose structure makes it feasible to recover each stroke as an individual centerline curve rather than a shaded region (see Fig. 1). Downstream tasks, such as stroke-level editing, benefit from high quality vectorization that preserves not only the appearance of the drawing but also its topology: the strokes, their endpoints and how they connect at intersections.

Existing methods typically underperform at junctions in one of two ways. The stroke geometry can be faulty: branches end before or after the true intersection, leaving gaps or overlaps. In other cases, the geometry is clean, but the method makes no explicit decision about which incoming branch continues into which outgoing branch, so the output is a collection of segments that terminate at every junction or ignore the semantics of the drawing. Recent learning-based methods improve junction geometry [PNCB21, YLA*24], with DeepSketch [YLA*24] representing the state of the art in visual fidelity, though performance remains drawing-dependent. Stroke

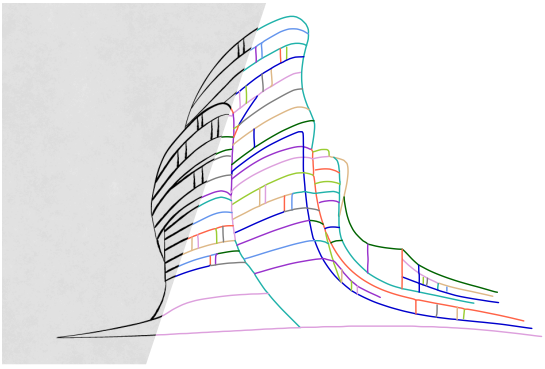


Figure 1: *A raster line drawing (left) and the vectorization produced by our method (right), in which each stroke is recovered as an individual curve, shown in a distinct color.*

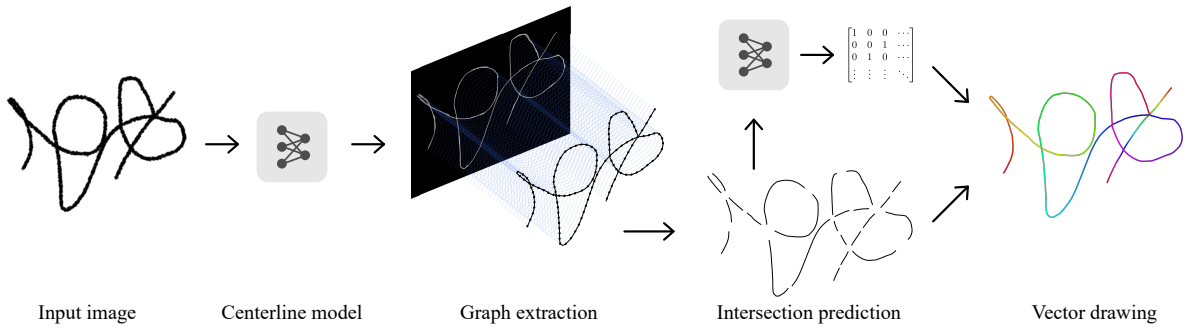


Figure 2: *Overview of our method. An image-to-image model predicts a centerline representation of the input drawing, which is converted to a planar straight-line graph whose intersections are resolved to single junction points. Each branch is then split into two, and a set-transformer predicts an involution encoding the connectivity of the half-branches, from which the final vector strokes are recovered.*

routing through junctions has received far less attention: Guo et al. [GZH*19] treat it directly but rely only on local context, leading to geometric imprecision and over-segmentation, while Magne and Sorkine-Hornung [MSH25] address the narrow case of a single self-intersecting stroke. Correctly vectorizing the topology of general line drawings remains a challenge.

To address the challenge, we propose a vectorization pipeline that separates recovering drawing geometry from stroke routing. A learned image-to-image model first predicts a centerline representation of the input drawing, which is converted into a planar straight-line graph whose intersections are resolved to single, topologically consistent vertices. A second model then predicts, for each junction, which pairs of incident branches continue through it as a single stroke. We cast this routing decision as the prediction of an *involution* on half-branches, which lets it be made globally rather than independently per junction. The firsts steps exist to make this decision well posed, but the involution formulation itself does not depend on how the graph is obtained. Our junction involution prediction model is a set-transformer encoder trained with a symmetric Gumbel-Sinkhorn relaxation on a synthetic intersection dataset. At inference, the discrete involution is recovered by reduction to a maximum-weight matching problem. Full strokes are reconstructed by traversing the graph according to the predicted involution.

Our pipeline vectorizes line drawings of varying complexity, producing clean strokes with crisp single-point junctions and explicit routing decisions. On a benchmark of 369 professional drawings, compared to four prior baselines, we recover substantially more ground truth intersections and produce fewer oversegmented strokes while remaining competitive on standard geometric metrics. Vectorization is fast, with a median inference time under two seconds per drawing on a consumer laptop.

2. Related works

Line drawing vectorization has been studied extensively; we review the main directions here. Early approaches relied on skeletonization methods, such as morphological thinning [LLS92] or medial-axis extraction [ZY01]. Later work improves these pipelines using image gradients [NHS*13] or shape boundary information [DCP19], or by simplifying the skeletonization output [FLB16]. Such pipelines are simple but often produce spurious branches and fail near intersections, where a single high-valence vertex is replaced by a cluster of degree-three nodes. Optimization-based methods [BS19] capture the drawing better by representing it as a frame field. This approach is refined with keypoint detection [PNCB21] and singularity-free frame fields [GHB*23]. These methods produce high-quality results on clean drawings, but the optimization is costly and remains sensitive to noisy input. Deep-learning methods have recently gained popularity. Recurrent neural networks, supervised by differentiable rasterization, are used to draw vector strokes one by one [MSSG*21]. Liu et al. [LLLW22] follow a similar strategy but with a transformer architecture. Both struggle to preserve local details due to the global embeddings they rely on. Yan et al. [YLA*24] cast vectorization as surface extraction from an unsigned distance field via neural dual-contouring. This method is the state of the art in terms of visual appearance, but it struggles to resolve junctions into single points and often oversegments strokes.

Stroke routing and drawing order. None of the methods above explicitly address the problem of routing strokes through intersections. Some works consider this direction: SmartTracer [HGW26] disambiguates intersections based on user input. Magne and Sorkine-Hornung [MSH25] address routing with a small convolutional network, but only for single-line drawings with degree-four junctions. Guo et al. [GZH*19] come closest in the general multi-stroke setting: their TRNet determines connectivity of partial curves at junctions and outputs full strokes. However, the network uses only local context, and in the limited results available, junction positions are not always cleanly resolved and some long strokes appear oversegmented. In contrast, our method correctly positions intersections and leverages global context for routing decisions, enabling robust handling of long strokes.

3. Method

Our method works on any input image depicting a line drawing with clearly distinguishable lines. First, an image-to-image model predicts the centerlines of the strokes. From this, we extract a planar straight line graph and partition it into disjoint polylines, each representing a branch. An intersection model then pairs branches

Figure 3: *Brushes used for rasterizing the vector strokes.*

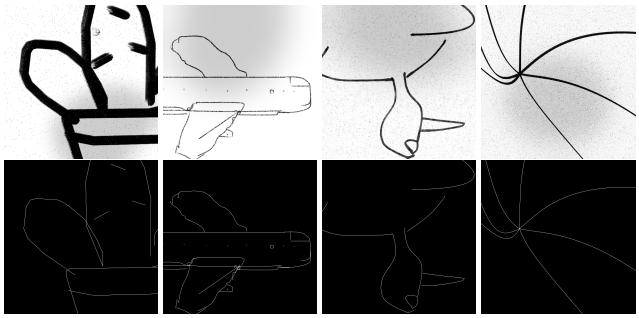


Figure 4: *Top: training samples from our dataset, shown with a range of augmentations. Bottom: corresponding ground-truth centerlines, transformed by the same geometric augmentations.*

around intersections, and the final paths are reconstructed from these pairings. The result is a set of polyline paths, each representing one stroke. An overview of our pipeline is presented in Fig. 2, and we discuss each step in detail in the remainder of this section.

3.1. Centerline model

We construct an image-to-image model that takes a line drawing as input and outputs a *centerline image*: a bitmap of the same size with 1 px wide antialiased lines for each stroke.

Dataset. To train our model, we generate a large dataset of 589k drawing-centerline pairs, of which 5% is used for validation. To that end, we take vector line drawings from existing datasets and rasterize them. The vector inputs are taken from the QuickDraw [Goo17, HE18], TU-Berlin [EHA12] and GeCreative [GGZP21] datasets. We also generate a synthetic dataset of star-like shapes (see Fig. 4, right), which lets the model learn to predict more accurate centerlines and reduces uncertainty around junctions. Details on its generation are provided in the supplementary material. Each vector drawing is rasterized on a blank canvas. The target images are rasterized with CairoSVG [Cou23] using a 0.5 px black stroke. The input drawings are rasterized with 10 different brushes, shown in Fig. 3, which cover a broad range of thickness, texture and stroke-local width variation. Parameters such as size and angle are randomized per sample to increase variance. During training, images are inverted to white strokes on a black background. We apply standard augmentations to both input and target: flips, rotations by multiples of 90 degrees and random crops. We also add Gaussian blur and noise to the input image, as well as random blurred ellipses simulating shadows, and random stroke brightness changes. Multiple samples from the training dataset are presented in Fig. 4.

Model. We use a fully convolutional image-to-image U-Net architecture [RFB15] and draw inspiration from [PNCB21] for the design details. We increase the model size by adding an extra pair of down/upsampling layers and modify the architecture so that it outputs a single centerline image, passed through a sigmoid function.

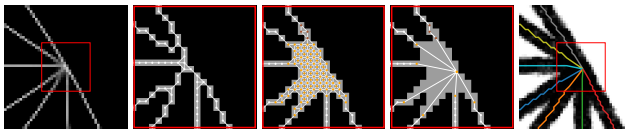


Figure 5: *Graph construction steps. The three middle images are zoomed in on the center of the image. Left: centerline image. Middle-left: binary skeleton. Middle: dilation of all pixels with connectivity greater than 3. Middle-right: clusters detected and resolved to junction points. Right: result overlaid on the input.*

Training. We use the following loss function:

$$\mathcal{L} = \|I_{GT} - \hat{I_C}\|_2 + \beta \|\hat{I_C}(1-\hat{I_C})\|_1, \quad (1)$$

where I_{GT} and $\hat{I_C}$ are the ground truth and predicted centerline images, respectively. The second term favors values close to 0 or 1, encouraging crisp edges rather than blurry dilated regions of uncertainty. In practice, we set $\beta = 0.001$. The model is trained for 8 epochs, taking 20.5 hours on a cluster of 8 RTX 4090 GPUs.

3.2. Polyline graph extraction

Given the centerline image, we aim to obtain a planar straight-line graph representing the drawing with clean junctions. We first convert the predicted centerline into a binary image and skeletonize it to obtain a graph G. We then identify clusters of triangles in the graph, which typically indicate intersections. Using geometric heuristics and image brightness, these clusters are resolved into intersection points. Finally, artifacts are removed, and polylines are recovered. Fig. 5 illustrates the graph extraction steps, and pseudocode for the whole stage is presented in the supplementary material.

Graph construction. We obtain a binary image from the grayscale centerline by applying a global threshold $T_1 = 25$ and discarding connected components whose maximum intensity falls below $T_2 = 85$, removing faint noise regions. To suppress broad shaded regions while preserving thin strokes, we subtract an adaptive thresholding of the centerline from this result. After filling small holes and removing single-pixel spikes, we skeletonize the image [ZS84] to obtain a one-pixel-wide medial axis. We then use this skeleton to refine the binary image by dilating the foreground around intersections. All branching pixels, i.e., pixels with more than 2 neighbors in the 8-connected skeleton, are dilated with a 5×5 mask, and pixels within this mask exceeding an intensity threshold of the original image are added back. This improves intersection recovery later. After filling holes and removing spikes once more, the main graph G is built from the binary image using 8-adjacency. Each foreground pixel becomes a node connected to its 8-neighborhood. Since diagonal edges cross when four pixels are mutually adjacent, we detect all geometric edge intersections and insert new nodes there, splitting the affected edges.

Triangle cluster identification. In the graph G, the regions around line intersections, and sometimes thick strokes, tend to contain dense clusters of triangular faces. We identify all such clusters: two triangles lie in the same cluster if they share an edge, a relation we take to be transitive. Each cluster is annotated with a set of *boundary edges*, that is, all edges with exactly one node inside the cluster and one node outside of it, and a set of *boundary nodes*, the nodes from the boundary edges that belong to the cluster. The degree of the cluster is defined as the number of boundary nodes. The *boundary polygon* of a cluster (see Fig. 6) is the sequence of nodes forming its outer perimeter, obtained by an algorithm similar to convex hull computation (see details in the supplementary material). Rare cases of clusters with holes are considered degenerate and removed from the graph. Finally, we assign each boundary node u an *outbound vector* $w[u] \in \mathbb{R}^2$ that estimates the direction of the cluster's outgoing branch (see Fig. 6). Let N_{out} be the set of neighbors of u outside the cluster, typically a single node. For each $v \in N_{\text{out}}$ we compute a unit direction vector from u towards v; when v has degree 2 we instead use the vector between the midpoints of its two incident segments, giving a smoother direction estimate. If N_{out} is a singleton, $w[u]$ is this direction vector, otherwise it is the bisector of the two direction vectors enclosing the widest angle.

Triangle cluster resolution. Each triangle cluster is resolved to replace the triangle structure with a topologically clean junction. The rules below are tried in order, from the most specific to the most general, and the first one that applies is used. Pseudocode for this step is provided in the supplementary material.

Special cases. Degree-0 clusters have no external connections and are removed as isolated noise. For degree-1 clusters, we keep the shortest path from the boundary node to the most distant cluster node and remove all other interior nodes. For degree-2 clusters, we compare the two outbound vectors: if they point in similar directions (positive dot product), the cluster lies on a bend, and we keep the longest boundary arc between the boundary nodes (see Fig. 6, left), otherwise the cluster lies on a nearly straight line and we keep the shortest interior path between the boundary nodes.

General case. For unresolved clusters, we seek a junction node. We sample a grid of candidate points at 0.25 px spacing inside the boundary polygon, keeping only those whose line segments to all boundary nodes do not cross background pixels. Among these, we select the point maximizing the total alignment between the outbound vectors and the directions toward the candidate, and replace the cluster interior with it as the new junction node (see Fig. 6, right). If no single point sees all boundary nodes (i.e., if there is no unobstructed line of sight), the cluster likely captures more than one junction. To test whether two points are sufficient, we determine, for each candidate, the boundary nodes it can reach without crossing background pixels, and search for a pair of candidate points that together reach every boundary node. Among valid pairs, we select the one maximizing total alignment and replace the cluster interior with these two new junction nodes.

Fallback. Any remaining cluster is replaced by the medial axis of its boundary polygon (approximated via a Voronoi-based method, see supplementary material), from which we prune the branches that do not lead to a boundary node. The graph obtained that way is inserted into G, replacing the cluster interior. The medial axis is defined for any simple polygon and determines a junction graph that follows the cluster's shape, so this fallback applies to any cluster not handled by the previous cases.

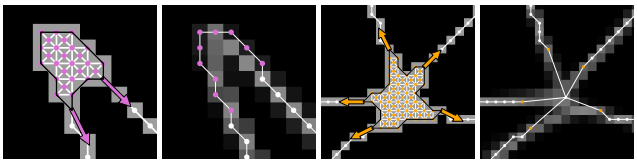


Figure 6: *A selection of cluster resolution cases. Arrows represent the outbound vector at each boundary node, and the black polygons are the boundary polygons of each cluster. Left: a degree-2 cluster that lies on a bend, where the longer arc of the boundary polygon is retained. Right: general case for a degree-5 cluster.*

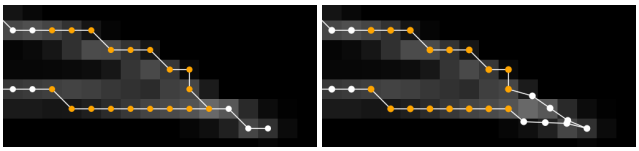


Figure 7: *A trailing branch (left) is replaced by an offset polyline forming a single V-shaped tip (right).*

Artifact removal. Once all triangle clusters are resolved, we apply three cleanup steps. First, we remove spurious short paths that do not correspond to actual strokes. All degree-1 paths (a chain of degree-2 nodes ending at a terminal node) shorter than 6 pixels and with a terminal intensity below a brightness threshold are discarded. Second, we improve sharp tips. We identify each degree-1 node connected to a degree-3 node along a path. Such a configuration is a sharp V-shaped tip rather than a true branching point when the path is short enough and the outbound vectors of the junction's two other branches point in similar directions (dot product threshold). In that case, we replace the tip-to-junction path with a polyline offset to both sides by a linearly increasing offset distance, and remove the original path (see Fig. 7). As a last step, we extract each branch from the resulting graph by collecting continuous runs of degree-2 nodes, and smooth them to reduce the noisy pixel-line geometry by iteratively moving each point toward the average of its neighbors.

3.3. Intersection prediction

To reconstruct complete strokes from the polylines obtained in the previous step, we must pair the branches. Routing is well posed only when every junction is a single vertex with clear incident branches. Existing vectorization methods rarely provide such a graph, which is why we develop the first two stages of our pipeline.

Involutions and half-branches An *involution* on a set $\{1,\dots,m\}$ is a permutation π with $\pi(\pi(k)) = k$ for all k, so every element is either a fixed point ($\pi(k) = k$) or part of a 2-cycle ($\pi(u) = v$ and $\pi(v) = u$ with $u \neq v$). We refer to an involution as a *symmetric permutation*, since its permutation matrix is symmetric.

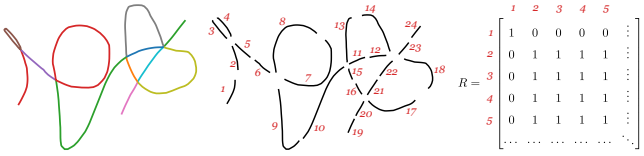


Figure 8: *Each polyline that forms the graph (left) is split into 2 half-branches (middle). Valid connection candidates are encoded in the adjacency constraint matrix R (right).*

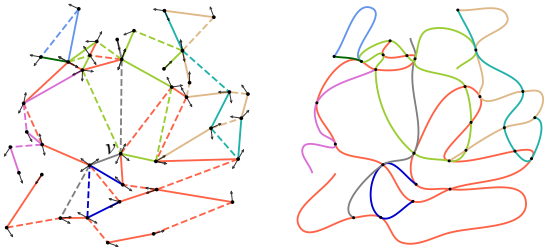


Figure 9: *Illustration of the sampling process. Left: a random planar graph is sampled and partitioned into edge-disjoint paths. For visualization purposes, every other edge is dashed. Tangents are then made consistent; at node v, for instance, the orange and green paths cross while the gray and orange paths are tangent. Right: splines are traced through each path under the tangency constraints.*

Our key insight is that pairing strokes locally around an intersection corresponds exactly to predicting an involution: a 2-cycle matches one stroke to another, while a fixed point means a stroke ends at the intersection. Because we match endpoints of branches rather than branches themselves, we split every polyline into two *half-branches*, each carrying one endpoint (see Fig. 8). The involution can be defined not only per intersection but also globally over the whole drawing. This is valid as long as half-branches around one intersection are never matched to those around another. We can therefore predict a single involution for the entire drawing, represented by a symmetric permutation matrix $S \in \{0,1\}^{n\times n}$, where n is the number of half-branches. This lets us use the full drawing context rather than a local window around each intersection. To restrict matches to valid neighbors, we build an adjacency constraint matrix $R \in \{0,1\}^{n\times n}$ encoding which half-branches share an intersection (see Fig. 8, right). The solution space then consists of all symmetric permutation matrices S with $S_{ij} \le R_{ij}$.

Intersection model. Our model takes the set of half-branch polylines and the adjacency constraint matrix R as input, and predicts a cost matrix C from which the involution matrix S is obtained.

Model architecture. The model starts with a polyline encoder that maps each half-branch to a latent vector. Each polyline is first resampled to a fixed number of points N by piecewise linear interpolation. To preserve high-frequency geometric details that a plain coordinate representation would lose, we apply a *Fourier feature mapping* [TSM*20]. Each coordinate x is expanded into $[x, \sin(2^0\pi x), \cos(2^0\pi x), \ldots, \sin(2^{F-1}\pi x), \cos(2^{F-1}\pi x)]$ with F frequency bands, to obtain $2+4F$ features per point. These per-point features are concatenated and passed through a three-layer MLP with ReLU activations to produce the half-branch latent vector. We use $N = 100$ and $F = 10$, so the embedding dimension is $N \cdot (2+4F) = 4200$. All latent vectors are then collected and processed by a transformer encoder. Since the half-branches form an unordered set rather than a sequence, the transformer uses neither positional encodings nor a causal mask, making the architecture permutation-equivariant. We use 4 layers of pre-norm multi-head self-attention with 4 heads each. Finally, the output head computes an attention matrix A, which we mask by setting $A_{ij} = -\infty$ wherever $R_{ij} = 0$ and symmetrize into $C = \frac{1}{2}(A + A^\mathsf{T})$. It thus holds that $C_{ij} = -\infty$ for pairs not sharing an intersection.

Training. The forward pass yields a symmetric matrix C, scoring how strongly each half-branch relates to another. To compare it against the ground truth symmetric permutation matrix S^{GT}, we apply the Gumbel-Sinkhorn operator [MBLS18], which normalizes a matrix into a doubly stochastic one and preserves symmetry (see supplementary material). The normalization step consists of adding a symmetric Gumbel noise matrix and applying Sinkhorn iterations to obtain $\hat{S}$, a symmetric doubly stochastic matrix. The Gumbel noise acts as a regularizer, since stochastic perturbation can smooth the loss surface in expectation [YPAW26] and effectively pushes the model to predict more extreme scores [BSH24].

To optimize the model weights, we use a loss combining a cross-entropy term and a Frobenius norm:

$$\mathcal{L} = \mathcal{L}_{\mathrm{CE}}(\hat{S}, S^{\mathrm{GT}}) + \|\hat{S} - S^{\mathrm{GT}}\|_F. \tag{2}$$

By double stochasticity, each row of $\hat{S}$ is a probability vector, so the cross-entropy loss can be written as:

$$\mathcal{L}_{\mathrm{CE}}(\hat{S}, S^{\mathrm{GT}}) = \sum_{i=1}^{n} \left(-\sum_{j=1}^{n} S^{\mathrm{GT}}_{i,j} \log(\hat{S}_{i,j}) \right). \tag{3}$$

Training dataset. We train our model on a synthetic dataset of lines extracted from random planar graphs, which lets us control features such as intersection degree and tangency and access the ground truth involutions, which are hard to obtain at scale from real drawings. To create each dataset sample, we draw points uniformly on a canvas, subject to a minimum pairwise distance, and build a Delaunay triangulation whose edges form candidate stroke segments. We remove about 30% of the edges at random, then decompose the remaining graph into edge-disjoint paths covering every edge, starting at odd-degree vertices when possible and extending greedily to a random valid neighbor. Paths meeting at a degree-2 node are merged to avoid over-segmented strokes. Tangents at nodes with degree higher than 2 are derived from the directions to the neighbors: crossing paths keep their own tangents while tangent paths (meeting without crossing) share an averaged tangent. Each path is then fitted with a piecewise cubic Bézier spline through the node coordinates under these tangency constraints and densely sampled into a polyline (see Figure 9). Optionally, free-form lines from a fractal random walk are added. Finally, each polyline is split at its intersections, and each branch is divided at its midpoint into two half-branches. The dataset stores the half-branch polylines in vector format together with the ground truth involution S^{GT} and the adjacency matrix R. In total, we create 100,000 samples and use 5% for validation.

To train the model on the kind of polyline geometries produced by the graph extraction step, we also generate a smaller dataset from its output. We create star-like shapes with the same method as for the centerline dataset, yielding reference polylines. We rasterize them and pass the obtained image through the centerline model and graph extraction. After verifying that the number of extracted polylines matches the reference, we pair them one-to-one to the reference polylines by comparing the geodesic centers of the polylines. This process recovers the adjacency information, giving the ground truth involution and constraint matrix R. We generate 15,000 samples using this process.

Inference. At inference, we must recover a discrete involution matrix S from the symmetric cost matrix C. A per-row max operator does not generally yield a permutation, and the Hungarian algorithm [Kuh55] for linear sum assignment returns a permutation that is not necessarily symmetric. To guarantee an involution, we reformulate the problem as maximum-weight matching. Intuitively, we decide which half-branches to pair so that the total benefit of pairing them over leaving them separate is maximized.

An involution decomposes into fixed points and 2-cycles. The diagonal entry C_{ii} is the cost of leaving half-branch i a fixed point. Thus, leaving both i and j fixed costs $C_{ii}+C_{jj}$, while pairing them into a 2-cycle costs $C_{ij}+C_{ji}=2C_{ij}$ by symmetry. Pairing is therefore preferable whenever $w_{ij}=2C_{ij}-C_{ii}-C_{jj}>0$. We build an undirected graph $G_u=([n],E)$ on the n half-branches, adding an edge (i,j) of weight w_{ij} for every pair with $w_{ij}>0$. Any independent edge set (matching) $M\subseteq E$ then maps directly to an involution: unmatched nodes become fixed points, and matched nodes form 2-cycles. Maximizing $\sum_{(i,j)\in M} w_{ij}$ is exactly the *maximum-weight matching* problem, solved with Edmonds' blossom algorithm [Edm65]. We allow non-perfect matchings, leaving some half-branches as fixed points. The involution matrix S is recovered from the maximum-weight matching.

3.4. Path reconstruction

The final step is to reconstruct the complete stroke paths. We denote by $h(i)$ the counterpart of half-branch i, so that i and $h(i)$ form the complete branch. Recall that we have the predicted involution π associated to the matrix S. Each half-branch i now has exactly two neighbors: $h(i)$, the other half of its branch, and $\pi(i)$, the half-branch it is matched to at an intersection. If $\pi(i)=i$, then i is an endpoint.

We reconstruct a path by alternating between the two maps. Starting from an unvisited half-branch i_0, we step to $h(i_0)$ (visiting the other half of the branch), then to $\pi(h(i_0))$ (following the match at the intersection), and continue alternating, marking visited half-branches, until we reach a fixed point or revisit a half-branch (forming a loop). Repeating this process from $\pi(i_0)$ grows the path in the other direction. Applied to every unvisited half-branch, this yields a set of paths that partition all half-branches. Each path is a sequence of half-branches whose polylines are retrieved, reoriented so that consecutive endpoints match, and concatenated into the final stroke.

4. Results

In this section we quantitatively and qualitatively compare our vectorization results against several prior baselines [BS19, PNCB21, MSSG*21, YLA*24]. Note that since the code of Guo et al. [GZH*19] is unavailable, a comparison against it is not included.

Table 1: *Quantitative comparisons of line drawing vectorization methods. The metrics are given as averages across all samples on the benchmark presented in Sec. 4.1. Note that our method performs particularly well on the topological metrics "Intersections" and "r_{SD}", while staying on par with SOTA on the geometric metrics. Gold and silver indicate best and second best scores, respectively.*

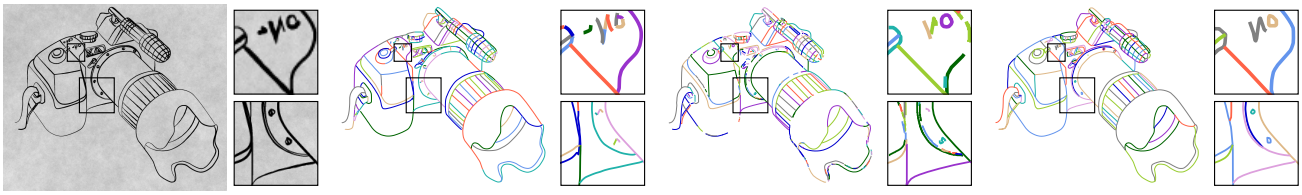
Method	Chamfer ↓	Stroke length ↓	Intersections ↑	r_{SD} ↓
[BS19]	0.82	591	22.1%	5.40
[PNCB21]	0.83	660	58.2%	5.92
[MSSG*21]	7.73	1379	8.9%	16.30
[YLA*24]	0.37	150	50.5%	1.71
Ours	0.52	175	73.7%	1.10

4.1. Quantitative comparison

Benchmark. To evaluate our results, we use the benchmark dataset of Yan et al. [YLA*24]. It comprises 369 artistic and technical line drawings collected in the wild, for which professional artists produced clean vector versions by hand. Each drawing is rasterized at a 1024×1024 resolution with a randomly sampled brush style, brush width and background texture, yielding visually diverse inputs.

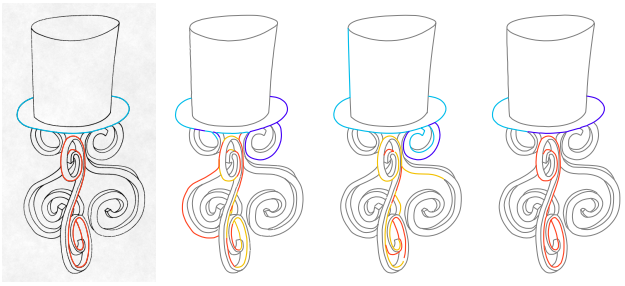
Metrics. Following Yan et al. [YLA*24], we measure the Chamfer distance and total stroke length difference (see supplementary material). While these metrics capture geometric faithfulness to the ground truth, they ignore the drawing topology: specifically which intersections are present and how strokes are routed around junctions (related to the number of strokes used). To assess how well intersections are captured, we follow Magne and Sorkine-Hornung [MSH25] and compare the intersections in the ground truth raster image, obtained from crowdsourced annotators, to those in the output vector representation, obtained by intersecting all line segments. An intersection is considered shared between the ground truth and output if the corresponding points lie within a 1.5-pixel threshold. We then report the ratio of correctly detected intersections over the total number of intersections in the ground truth input. In addition, we define the *Stroke Density Ratio* r_{SD}, where the stroke density of a vector representation is the number of its polyline strokes divided by their total arc length; r_{SD} is the stroke density of the prediction divided by that of the ground truth. A value close to 1 thus indicates a similar number of strokes per unit length, while values above or below 1 indicate over- or under-segmentation of the prediction, respectively.

Evaluation results. The evaluation results are presented in Table 1, with additional statistics in the supplementary material. Our method is competitive but performs slightly worse than DeepSketch [YLA*24] on the Chamfer distance and stroke length difference metrics. An important caveat should be noted: in this benchmark, Yan et al. [YLA*24] appear to use the same brushes for rasterizing the ground truth sketches as for their training data, granting them a slight competitive advantage. Our method achieves a stroke density ratio closest to 1, indicating it is less prone to the



Input Bessmeltsev and Solomon [BS19] Yan et al. [YLA*24] Ours

Figure 10: *Qualitative comparison of various vectorization methods. Each stroke is shown in a different color.*



Input [BS19] [YLA*24] Ours

Figure 11: *Qualitative comparison of various vectorization methods. Selected strokes are colored to highlight stroke routing.*

stroke over-segmentation observed in other methods. It also produces more correct intersections than all other methods, by a large margin. The method of Puhachov et al. [PNCB21] ranks second on this metric, likely because intersection keypoints are explicitly detected in the pipeline.

Runtime. On an M3 Pro laptop with 18GB of RAM, using PyTorch's MPS backend for model inference [PyT26], our method runs in 5s on average over the whole dataset, with a median of 2s and 90% of samples taking less than 9s. Note that model loading time is excluded, as it is a one-time cost. Our method is faster than the other baselines: DeepSketch [YLA*24] reports a mean runtime of 21s on a powerful cluster node with 32GB of RAM and an NVIDIA A100 GPU. As shown by Yan et al. [YLA*24], the method of Mo et al. [MSSG*21] is slower, likely due to its recurrent network. The remaining baselines [BS19, PNCB21] are frame field methods involving expensive optimization and are thus substantially slower.

4.2. Visual comparison

We visually compare our results against the methods of Yan et al. [YLA*24] and Bessmeltsev and Solomon [BS19] on samples from the benchmark dataset of Yan et al. [YLA*24] in Fig. 10 and Fig. 11. Overall, our method better preserves the topology and visual appearance of the original drawing. DeepSketch [YLA*24] suffers from over-segmented strokes and gaps, although this is highly drawing dependent. The method of Bessmeltsev and Solomon [BS19] produces clean lines but introduces noticeable topological modifications compared to the raster image. In addition, both methods yield unnatural stroke routing at junctions. Both the benchmark inputs and our training data are rasterized drawings. To test whether this biases our results, we additionally compare the same methods on real line

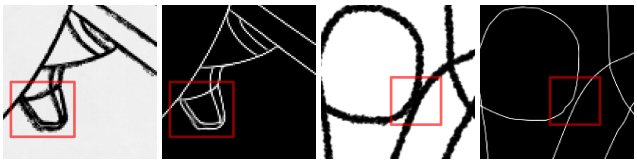


Figure 12: *Input drawings (left and middle-right) present some challenging regions. Centerlines (middle-left and right) predicted by our model accurately represent them.*

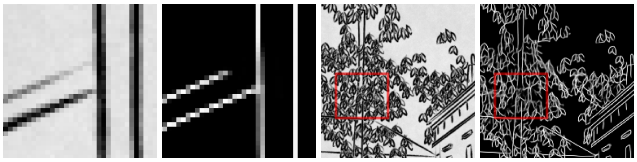


Figure 13: *Left: thin, washed-out lines are missed by the centerline model, leaving gaps. Right: the drawing presents high-frequency, messy details that our centerline model struggles to capture.*

drawings from public-domain books in Fig. 15. These inputs are not rasterized at all, and their stroke style was not seen by any models during training. Our method nevertheless produces longer strokes that follow the artist's intent and captures thin details, indicating that it does not overfit to its synthetic training data and generalizes to real drawings. Further visual comparisons are provided in the supplementary material.

4.3. Limitations and future work

Centerline model. As shown in Fig. 12, our centerline model disambiguates touching or overlapping strokes in busy regions, capturing the drawing's semantics better than image thinning or skeleton methods. However, it struggles with degenerate inputs: messy overlapping strokes yield blurry predictions (Fig. 13, right), thin washed-out lines may be ignored (Fig. 13, left), and very thick strokes, absent from training, degrade performance. Augmenting the training set with such cases should resolve these issues.

Graph extraction. This step is the main source of topology errors in regions of fine detail. Close parallel lines may merge because they become indistinguishable once the binary image is extracted (Fig. 14, top), and our greedy handling of intersections can collapse two expected junctions into one (Fig. 14, bottom). Favoring pixel intensity over thresholding, or using a learned component operating

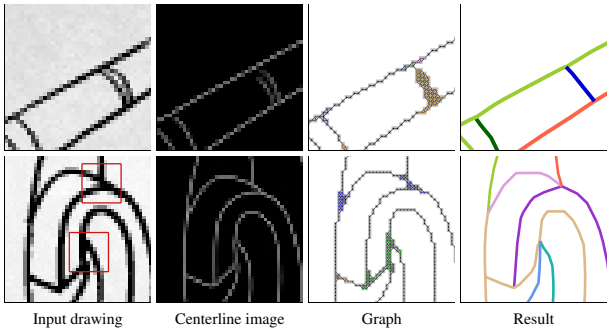


Figure 14: *Some limitations of our graph extraction method. Top: close, parallel lines are resolved to a single line. Bottom: the two highlighted regions show strokes that meet at 2 different junctions in both cases. Our method resolves them to a single junction.*

directly on the predicted image, could help recover the topology that the current geometric procedure loses.

Stroke routing. Although our routing improves on previous methods, failures remain: in Fig. 11, the light and dark blue strokes are left unconnected. This stems partly from training the model on synthetic data alone, which leaves a distribution gap with real sketches. Fine-tuning on annotated real-world sketches is a natural next step. The intersection model also reasons from stroke geometry alone and cannot recognize semantic object boundaries. Conditioning the routing on image embeddings is a promising research direction.

Runtime. While our method is generally fast, it can take longer on complex samples. Inference of the centerline and intersection models stays sub-second, owing to their small size (14.7 and 1.2 million parameters). The bottleneck is graph extraction, whose runtime scales with the number of clusters found. A parallel C++ implementation of this step would improve performance.

5. Conclusion

We presented an end-to-end pipeline for line drawing vectorization that separates the recovery of drawing geometry from the routing of strokes through junctions. A learned image-to-image model first produces a thin centerline representing the raster input. A graph-extraction step then converts this image into a planar straight-line graph, whose intersection clusters are collapsed to single vertices. A second model, trained on a synthetic dataset, predicts branch pairings at each junction. We formulated this decision globally as the prediction of an involution on half-branches, trained with a symmetric Gumbel-Sinkhorn relaxation. At inference, a discrete involution is recovered via a maximum-weight matching problem. On the benchmark of [YLA*24], our pipeline achieves the highest correct intersection rate and the best stroke density ratio, while remaining competitive on Chamfer distance and total stroke length. In addition, the correct connectivity at junctions makes our outputs easily editable. We released our code publicly at github.com/verven/InvolutionStrokeVectorization to foster further research in this field.

6. Acknowledgements

We thank the anonymous reviewers for their careful reading and valuable comments.

References

[BS19] BESSMELTSEV M., SOLOMON J.: Vectorization of line drawings via polyvector fields. *ACM Trans. Graph. 38*, 1 (Jan. 2019). doi:10.1145/3202661. 2, 6, 7, 9

[BSH24] BINNINGER A., SORKINE-HORNUNG O.: Sd-πxl: Generating low-resolution quantized imagery via score distillation. In *SIGGRAPH Asia 2024 Conference Papers* (2024). doi:10.1145/3680528.3687570. 5

[Cou23] COURTBOUILLON: Cairosvg, 2023. Version 2.7.1. URL: https://cairosvg.org/. 3

[DCP19] DONATI L., CESANO S., PRATI A.: A complete hand-drawn sketch vectorization framework. *Multimedia Tools and Applications 78*, 14 (Jul 2019), 19083–19113. doi:10.1007/s11042-019-7311-3. 2

[Edm65] EDMONDS J.: Paths, trees, and flowers. *Canadian Journal of Mathematics 17* (1965), 449–467. doi:10.4153/CJM-1965-045-4. 6

[EHA12] EITZ M., HAYS J., ALEXA M.: How do humans sketch objects? *ACM Trans. Graph. 31*, 4 (July 2012). doi:10.1145/2185520.2185540. 3

[FLB16] FAVREAU J.-D., LAFARGE F., BOUSSEAU A.: Fidelity vs. simplicity: a global approach to line drawing vectorization. *ACM Trans. Graph. 35*, 4 (July 2016). doi:10.1145/2897824.2925946. 2

[GGZP21] GE S., GOSWAMI V., ZITNICK C. L., PARIKH D.: Creative sketch generation, 2021. URL: https://arxiv.org/abs/2011.10039, arXiv:2011.10039. 3

[GHB*23] GUȚAN O., HEGDE S., BERUMEN E. J., BESSMELTSEV M., CHIEN E.: Singularity-free frame fields for line drawing vectorization. *Computer Graphics Forum 42*, 5 (2023). doi:10.1111/cgf.14901. 2

[Goo17] GOOGLE CREATIVE LAB: The quick, draw! dataset, 2017. URL: https://github.com/googlecreativelab/quickdraw-dataset. 3

[GZH*19] GUO Y., ZHANG Z., HAN C., HU W., LI C., WONG T.-T.: Deep line drawing vectorization via line subdivision and topology reconstruction. *Computer Graphics Forum 38*, 7 (2019), 81–90. doi:10.1111/cgf.13818. 2, 6

[HE18] HA D., ECK D.: A neural representation of sketch drawings. In *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=Hy6GHpkCW. 3

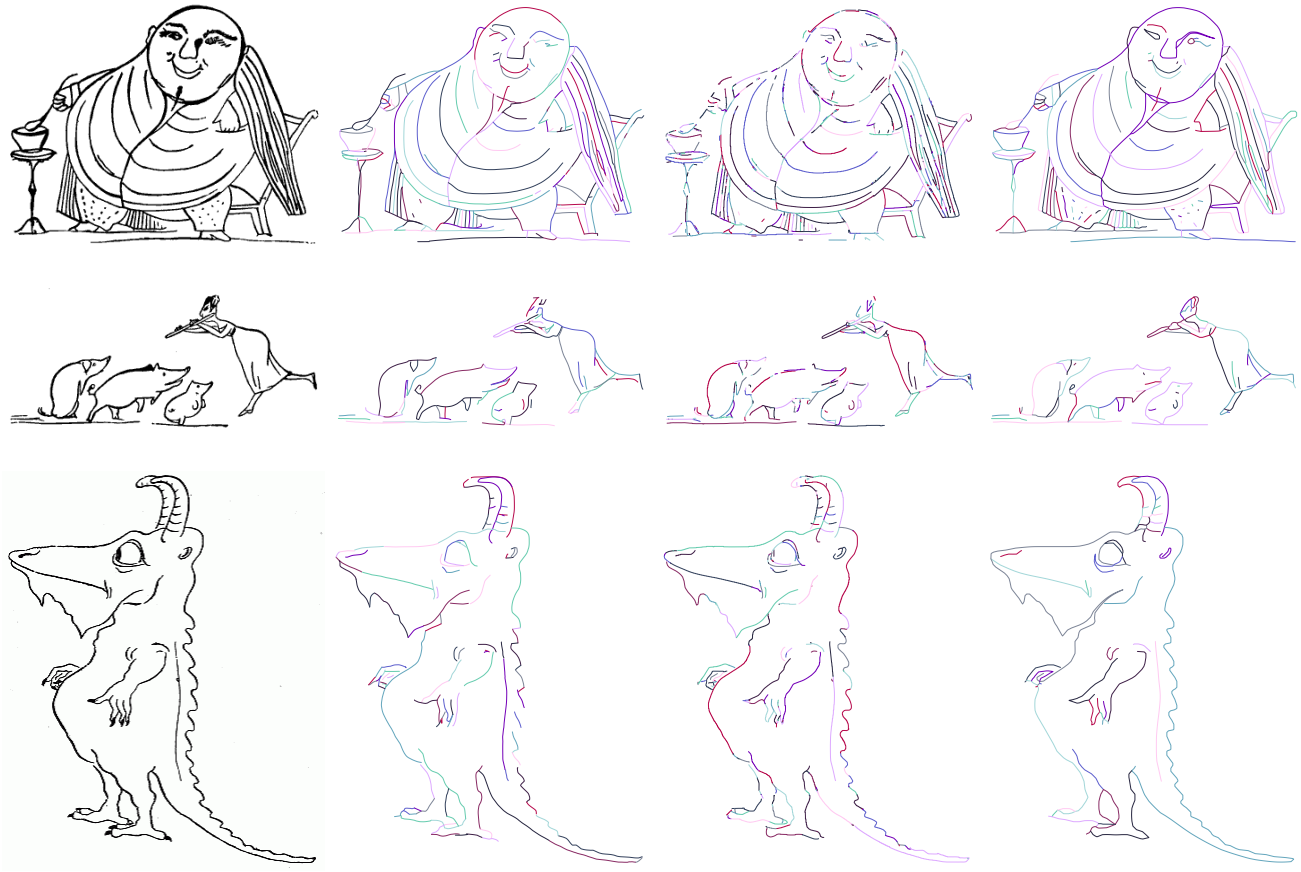
[HGW26] HUANG Z., GUAN Z., WANG Z.: Smarttracer: Interactive tracing-based stroke extraction for complex line art. *Computer Aided Geometric Design 128* (2026), 102568. doi:10.1016/j.cagd.2026.102568. 2

[Kuh55] KUHN H. W.: The hungarian method for the assignment problem. *Naval Research Logistics Quarterly 2*, 1-2 (1955), 83–97. doi:10.1002/nav.3800020109. 6

[LLLW22] LIU H., LI C., LIU X., WONG T.-T.: End-to-end line drawing vectorization. *Proceedings of the AAAI Conference on Artificial Intelligence 36*, 4 (Jun. 2022), 4559–4566. doi:10.1609/aaai.v36i4.20379. 2

[LLS92] LAM L., LEE S.-W., SUEN C.: Thinning methodologies-a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 14*, 9 (1992), 869–885. doi:10.1109/34.161346. 2

[MBLS18] MENA G., BELANGER D., LINDERMAN S., SNOEK J.: Learning latent permutations with gumbel-sinkhorn networks. In *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=Byt3oJ-0W. 5



Input Bessmeltsev and Solomon [BS19] Yan et al. [YLA*24] Ours

Figure 15: *Qualitative comparison of various vectorization methods on real line drawings taken from A Book of Nonsense by Edward Lear (first two rows) and The Bad Child's Book of Beasts by Hilaire Belloc (last row), both in the public domain. Each stroke is shown in a different color. In the last row, our method recovers the serrated back and tail as one stroke while both baselines split it.*

[MSH25] MAGNE T., SORKINE-HORNUNG O.: Single-line drawing vectorization. *Computer Graphics Forum 44*, 7 (2025). doi:10.1111/cgf.70228. 2, 6

[MSSG*21] MO H., SIMO-SERRA E., GAO C., ZOU C., WANG R.: General virtual sketching framework for vector line art. *ACM Trans. Graph. 40*, 4 (July 2021). doi:10.1145/3450626.3459833. 2, 6, 7

[NHS*13] NORIS G., HORNUNG A., SUMNER R. W., SIMMONS M., GROSS M.: Topology-driven vectorization of clean line drawings. *ACM Trans. Graph. 32*, 1 (Feb. 2013). doi:10.1145/2421636.2421640. 2

[PNCB21] PUHACHOV I., NEVEU W., CHIEN E., BESSMELTSEV M.: Keypoint-driven line drawing vectorization via polyvector flow. *ACM Trans. Graph. 40*, 6 (Dec. 2021). doi:10.1145/3478513.3480529. 1, 2, 3, 6, 7

[PyT26] PYTORCH CONTRIBUTORS: Mps backend, 2026. URL: https://docs.pytorch.org/docs/2.12/notes/mps.html. 7

[RFB15] RONNEBERGER O., FISCHER P., BROX T.: U-net: Convolutional networks for biomedical image segmentation. In *Proc. MICCAI* (2015), pp. 234–241. 3

[Sav26] SAVAGE INTERACTIVE: Procreate, 2026. URL: https://procreate.com/. 1

[TSM*20] TANCIK M., SRINIVASAN P., MILDENHALL B., FRIDOVICH-KEIL S., RAGHAVAN N., SINGHAL U., RAMAMOORTHI R., BARRON J., NG R.: Fourier features let networks learn high frequency functions in low dimensional domains. In *Advances in Neural Information Processing Systems* (2020), vol. 33, pp. 7537–7547. URL: proceedings.neurips.cc/paper_files/paper/2020/file/55053683268957697aa39fba6f231c68-Paper.pdf. 5

[YLA*24] YAN C., LI Y., ANEJA D., FISHER M., SIMO-SERRA E., GINGOLD Y.: Deep sketch vectorization via implicit surface extraction. *ACM Trans. Graph. 43*, 4 (July 2024). doi:10.1145/3658197. 1, 2, 6, 7, 8, 9

[YPAW26] YOUSEFI S., PLESNER A., ACZEL T., WATTENHOFER R.: Mind the gap: Removing the discretization gap in differentiable logic gate networks. In *Proc. Neural Information Processing Systems* (2026). URL: https://openreview.net/forum?id=chYXaetMmz. 5

[ZS84] ZHANG T. Y., SUEN C. Y.: A fast parallel algorithm for thinning digital patterns. *Commun. ACM 27*, 3 (Mar. 1984), 236–239. doi:10.1145/357994.358023. 3

[ZY01] ZOU J. J., YAN H.: Cartoon image vectorization based on shape subdivision. In *Proc. Computer Graphics International* (2001), pp. 225–231. doi:10.1109/CGI.2001.934678. 2