Chapter 6

Application of Principal Curves to Hand-Written Character Skeletonization

The main subject of this chapter is an application of an extended version of the polygonal line algorithm to hand-written character skeletonization. Skeletonization is one of the important areas in image processing. It is most often, although not exclusively, used for images of hand-written or printed characters so we describe it here in this context. When we look at the image of a letter, we see it as a collection of curves rather than a raster of pixels. Since the earliest days of computers, it has been one of the challenges for researchers working in the area of pattern recognition to imitate this ability of the human mind [Din55, KCRU57]. Approaching skeletonization from a practical point of view, representing a character by a set of thin curves rather than by a raster of pixels is useful for reducing the storage space and processing time of the character image. It was found that this representation is particularly effective in finding relevant features of the character for optical character recognition [Deu68, AH69].

The objective of skeletonization is to find the medial axis of a character. Ideally, the medial axis is defined as a smooth curve (or set of curves) that follows the shape of a character equidistantly from its contours. In case of hand-written characters, one can also define the medial axis as the trajectory of the penstroke that created the letter. Most skeletonization algorithms approximate the medial axis by a unit-width binary image obtained from the original character by iteratively peeling its contour pixels until there remains no more removable pixel [Pav80, NS84, SA86]. The process is called the *thinning* of the character template, and the result is the *skeleton* of the character. The different thinning methods are characterized by the rules that govern the deletion of black pixels.

In this chapter we propose another approach to skeletonization. The development of the method

was inspired by the apparent similarity between the definition of principal curves and the medial axis. A principal curve is a smooth curve that goes through the "middle" of a data set, whereas the medial axis is a set of smooth curves that go equidistantly from the contours of a character. Therefore, by representing the black pixels of a character by a two-dimensional data set, one can use the principal curve of the data set to approximate the medial axis of the character. Other methods using this "analogue" approach for skeletonization are described in Section 6.1. In this section we also summarize existing applications of the HS principal curve algorithm.

Since the medial axis can be a *set* of connected curves rather then only *one* curve, in Section 6.2 we extend the polygonal line algorithm to find a *principal graph* of a data set. The extended algorithm also contains two elements specific to the task of skeletonization, an initialization method to capture the approximate topology of the character, and a collection of restructuring operations to improve the structural quality of the skeleton produced by the initialization method. To avoid confusion, in what follows we use the term skeleton for the unit-width binary image approximating the medial axis, and we refer to the set of connected curves produced by the polygonal line algorithm as the *skeleton graph* of the character template.

In Section 6.3 test results of the extended polygonal line algorithm are presented. In Section 6.3.1 we apply the algorithm to isolated hand-written digits from the NIST Special Database 19 [Gro95]. The results indicate that the proposed algorithm finds a smooth medial axis of the great majority of a wide variety of character templates, and substantially improves the pixelwise skeleton obtained by traditional thinning methods. In Section 6.3.2 we present results of experiments with images of continuous handwriting. These experiments demonstrate that the skeleton graph produced by the algorithm can be used for representing hand-written text efficiently.

6.1 Related Work

6.1.1 Applications and Extensions of the HS Algorithm

Hastie [Has84] presented several experiments on real data. In the first example, computer chip waste is sampled and analyzed by two laboratories to estimate the gold content of the lot. It is in the interest of the owner of the lot to know which laboratory produces on average lower gold content estimates for a given sample. The principal curve method was used to point out that at higher levels of gold content one of the laboratories produced higher assays than the other. The difference was reversed at lower levels. [Has84] argues that these results could not have been obtained by using standard regression techniques. In another example, a principal curve was used for non-linear factor analysis on a data set of three-dimensional points representing measurements of mineral content of core samples.

The first real application of principal curves was part of the Stanford Linear Collider project [HS89]. The collider consists of a linear accelerator used to accelerate two particle beams, and two arcs that bend these beams to bring them to collision. The particle beams are guided by roughly 475 magnets that lie on a smooth curve with a circumference of about three kilometers. Measurement errors in the range of ± 1 millimeters in placing the magnets resulted that the beam could not be adequately focused. Engineers realized that it was not necessary to move the magnets to the ideal curve, but rather to a curve through the existing positions that was smooth enough to allow focused bending of the beam. The HS principal curve procedure was used to find this curve.

Banfield and Raftery [BR92] described an almost fully automatic method for identifying ice floes and their outlines in satellite images. The core procedure of the method uses a closed principal curve to estimate the floe outlines. Besides eliminating the estimation bias of the HS algorithm (see Section 3.1.3), [BR92] also replaced the initialization step of the HS algorithm by a more sophisticated routine that produced a rough estimate of the floe outlines. Furthermore, [BR92] extended existing clustering methods by allowing groups of data points to be centered about principal curves rather than points or lines.

Principal curve clustering was further extended and analyzed by Stanford and Raftery [SR00]. Here, fitting a principal curve is combined with the Classification EM algorithm [CG92] to iteratively refine clusters of data centered about principal curves. The number of clusters and the smoothness parameters of the principal curves are chosen automatically by comparing approximate Bayes factors [KR95] of different models. Combining the clustering algorithm with a denoising procedure and an initialization procedure, [SR00] proposed an automatic method for extracting curvilinear features of simulated and real data.

Chang and Ghosh [CG98b, CG98a] used principal curves for nonlinear feature extraction and pattern classification. [CG98b] pointed out experimentally that a combination of the HS and BR algorithms (the BR algorithm is run after the HS algorithm) reduces the estimation bias of the HS algorithm and also decreases the variance of the BR algorithm that was demonstrated in Section 5.2. [CG98b] and [CG98a] demonstrated on several examples that the improved algorithm can be used effectively for feature extraction and classification.

Reinhard and Niranjan [RN98] applied principal curves to model the short time spectrum of speech signals. First, high-dimensional data points representing diphones (pairs of consecutive phones) are projected to a two-dimensional subspace. Each diphone is than modeled by a principal curve. In the recognition phase, test data is compared to the principal curves representing the different diphones, and classified as the diphone represented by the nearest principal curve. [RN98] demonstrated in experiments that the diphone recognition accuracy of can can be comparable to the accuracy of the state-of-the-art hidden Markov models.

6.1.2 Piecewise Linear Approach to Skeletonization

[SWP98] used the HS principal curve algorithm for character skeletonization. The initial curve is produced by a variant of the SOM algorithm where the neighborhood relationships are defined by a minimum spanning tree of the pixels of the character template. The HS algorithm is then used to fit the curve to the character template. In the expectation step a weighted kernel smoother is used which, in this case, is equivalent to the update rule of the SOM algorithm. [SWP98] demonstrated that principal curves can be successfully used for skeletonizing characters in fading or noisy texts where traditional skeletonization techniques are either inapplicable or perform poorly.

Similar skeletonization methods were proposed by Mahmoud et al. [MAG91] and Datta and Parui [DP97]. Similarly to [SWP98], [DP97] uses the SOM algorithm to optimize the positions of vertices of a piecewise linear skeleton. The algorithm follows a "bottom-up" strategy in building the skeletal structure: the approximation starts from a linear topology and later adds forks and loops to the skeleton based on local geometric patterns formed during the SOM optimization. [MAG91] proposed an algorithm to obtain piecewise linear skeletons of Arabic characters. The method is based on fuzzy clustering and the fuzzy ISODATA algorithm [BD75] that uses a similar optimization to the batch version of the SOM algorithm.

Although, similarly to the principal graph algorithm, [MAG91, DP97, SWP98] also use a piecewise linear approximation of the skeleton of the character, their approaches substantially differ from our approach in that smoothness of the skeleton is not a primary issue in these works. Although the SOM algorithm implicitly ensures smoothness of the skeleton to a certain extent, it lacks a clear and intuitive formulation of the two competing criteria, smoothness of the skeleton and closeness of the fit, which is explicitly present in our method. In this sense our algorithm complements these methods rather then competes with them. For example, the method of [SWP98] could be used as an alternative initialization step for the principal graph algorithm if the input is too noisy for our thinning-based initialization step. One could also use the restructuring operations described in [DP97] combined with the fitting-and-smoothing optimization step of the principal graph algorithm in a "bottom-up" approach of building the skeleton graph.

6.2 The Principal Graph Algorithm

In this section we describe the principal graph algorithm, an extension of the polygonal line algorithm for finding smooth skeletons of hand-written character templates. To transform binary (black-and-white) character templates into two-dimensional data sets, we place the midpoint of the bottom-most left-most pixel of the template to the center of a coordinate system. The unit length of the coordinate system is set to the width (and height) of a pixel, so the midpoint of each pixel

has integer coordinates. Then we add the midpoint of each black pixel to the data set. Figure 26 illustrates the data representation model.

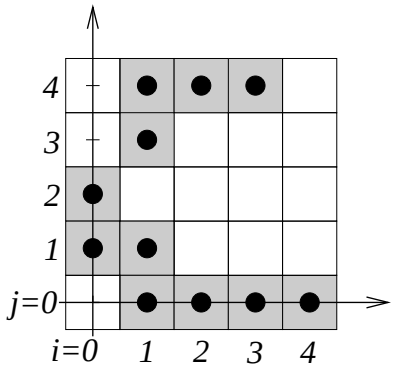


Figure 26: Representing a binary image by the integer coordinates of its black pixels. The 5×5 image is transformed into the set $x = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \begin{bmatrix} 3 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 4 \\ 0 \end{bmatrix} \right\}$.

The polygonal line algorithm was tested on images of isolated handwritten digits from the NIST Special Database 19 [Gro95]. We found that the polygonal line algorithm can be used effectively to find smooth medial axes of simple digits which contain no loops or crossings of strokes. Figure 27 shows some of these results.

To find smooth skeletons of more complex characters we extend and modify the polygonal line algorithm. In Section 6.2.1 we extend the optimization and the projection steps so that in the inner loop of the polygonal line algorithm we can optimize Euclidean graphs rather than only polygonal curves. To capture the approximate topology of the character, we replace the initialization step by a more sophisticated routine based on a traditional thinning method. The new initialization procedure is described in Section 6.2.2. Since the initial graph contains enough vertices for a smooth approximation, we no longer need to use the outer loop of the polygonal line algorithm to add vertices to the graph one by one. Instead, we use the inner loop of the algorithm only twice. Between the two fitting-and-smoothing steps, we "clean" the skeleton graph from spurious branches and loops that are created by the initial thinning procedure. Section 6.2.3 describes the restructuring operations used in this step. The flow chart of the extended polygonal line algorithm is given in Figure 28. Figure 29 illustrates the evolution of the skeleton graph on an example.

6.2.1 Principal Graphs

In this section we introduce the notion of a *Euclidean graph* as a natural extension of polygonal curves. The principal curve algorithm is then extended to optimize a Euclidean graph rather than a single curve. We introduce new vertex types to accommodate junction points of a graph. The new vertex types are tailored to the task of finding a smooth skeleton of a character template. In a

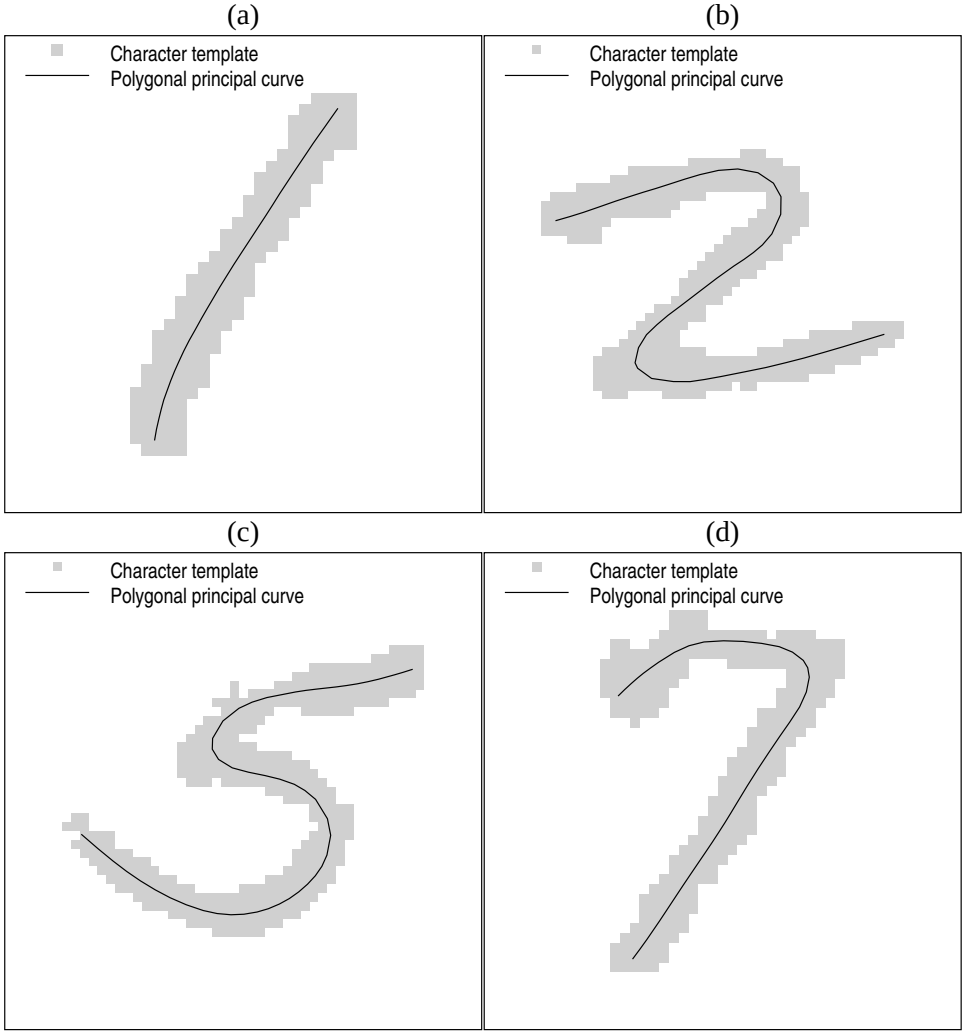


Figure 27: The polygonal line algorithm can be used effectively to find smooth medial axes of simple digits which contain no loops or crossings of strokes.

different application, other vertex types can be introduced along the same lines.

Once the local distance function and the local penalty term are formulated for the new vertex types, the vertex optimization step (Section 5.1.5) is completely defined for Euclidean graphs. The projection step (Section 5.1.4) can be used without modification. As another indication of the robustness of the polygonal line algorithm, the penalty factor λ, which was developed using the data generating model (85), remains as defined in (73).

Euclidean Graphs

A Euclidean graph $G_{\mathcal{V},\S}$ in the d-dimensional Euclidean space is defined by two sets, $\mathcal{V}$ and §, where $\mathcal{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\} \subset \mathbb{R}^d$ is a set of *vertices,* and $\S = \{(\mathbf{v}_{i_1}, \mathbf{v}_{j_1}), \ldots, (\mathbf{v}_{i_k}, \mathbf{v}_{j_k})\} = \{\mathbf{s}_{i_1 j_1}, \ldots, \mathbf{s}_{i_k, j_k}\}$,

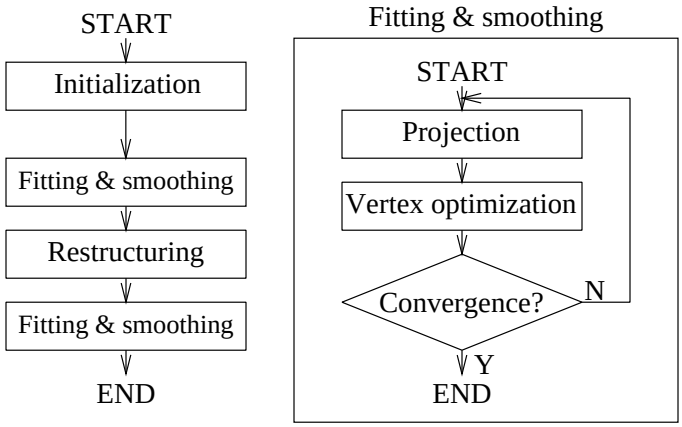


Figure 28: The flow chart of the extended polygonal line algorithm.

$1 \leq i_1, j_1, \ldots, i_k, j_k \leq m$ is a set of *edges*, such that $\mathbf{s}_{ij}$ is a line segment that connects $\mathbf{v}_i$ and $\mathbf{v}_j$. We say that two vertices are *adjacent* or *neighbors* if there is an edge connecting them. The edge $\mathbf{s}_{ij} = (\mathbf{v}_i, \mathbf{v}_j)$ is said to be *incident* with the vertices $\mathbf{v}_i$ and $\mathbf{v}_j$. The vertices $\mathbf{v}_i$ and $\mathbf{v}_j$ are also called the *endpoints* of $\mathbf{s}_{ij}$. The *degree* of a vertex is the number of edges incident with it.

Let $\mathbf{x} \in \mathbb{R}^d$ be an arbitrary data point. The squared Euclidean distance between $\mathbf{x}$ and a graph $G_{\mathcal{V},\S}$ is the squared distance between $\mathbf{x}$ and the nearest edge of $G_{\mathcal{V},\S}$ to $\mathbf{x}$, i.e.,

$$\Delta(\mathbf{x}, G_{\mathcal{V},\S}) = \min_{\mathbf{s}\in\S} \Delta(\mathbf{x}, \mathbf{s}).$$

Then, given a data set $\mathcal{X}_n = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, the empirical distance function of $G_{\mathcal{V},\S}$ is defined as usual,

$$\Delta_n(G_{\mathcal{V},\S}) = \frac{1}{n} \sum_{i=1}^{n} \Delta(\mathbf{x}_i, G_{\mathcal{V},\S}).$$

Note that a polygonal curve $\mathbf{f}$ is a special Euclidean graph with the property that the vertices of $\mathbf{f}$, $\mathbf{v}_1, \ldots, \mathbf{v}_m$, can be indexed so that $\mathbf{s}_{ij} = (\mathbf{v}_i, \mathbf{v}_j)$ is an edge if and only if $j = i+1$.

In what follows we will use the term *graph* as an abbreviation for Euclidean graph. We will also omit the indices of $G_{\mathcal{V},\S}$ if it does not cause confusion.

New Vertex Types

The definition of the local distance function (74) in Section 5.1.5 differentiates between vertices at the end and in the middle of the polygonal curve. We call these vertices *end-vertices* and *line-vertices*, respectively. In this section we introduce new vertex types to accommodate intersecting curves that occur in handwritten characters. Vertices of different types are characterized by their degrees and the types of the local curvature penalty imposed at them (see Table 3).

The only vertex type of degree one is the end-vertex. Here we penalize the squared length of the incident edge as defined in (75). If two edges are joined by a vertex, the vertex is either a

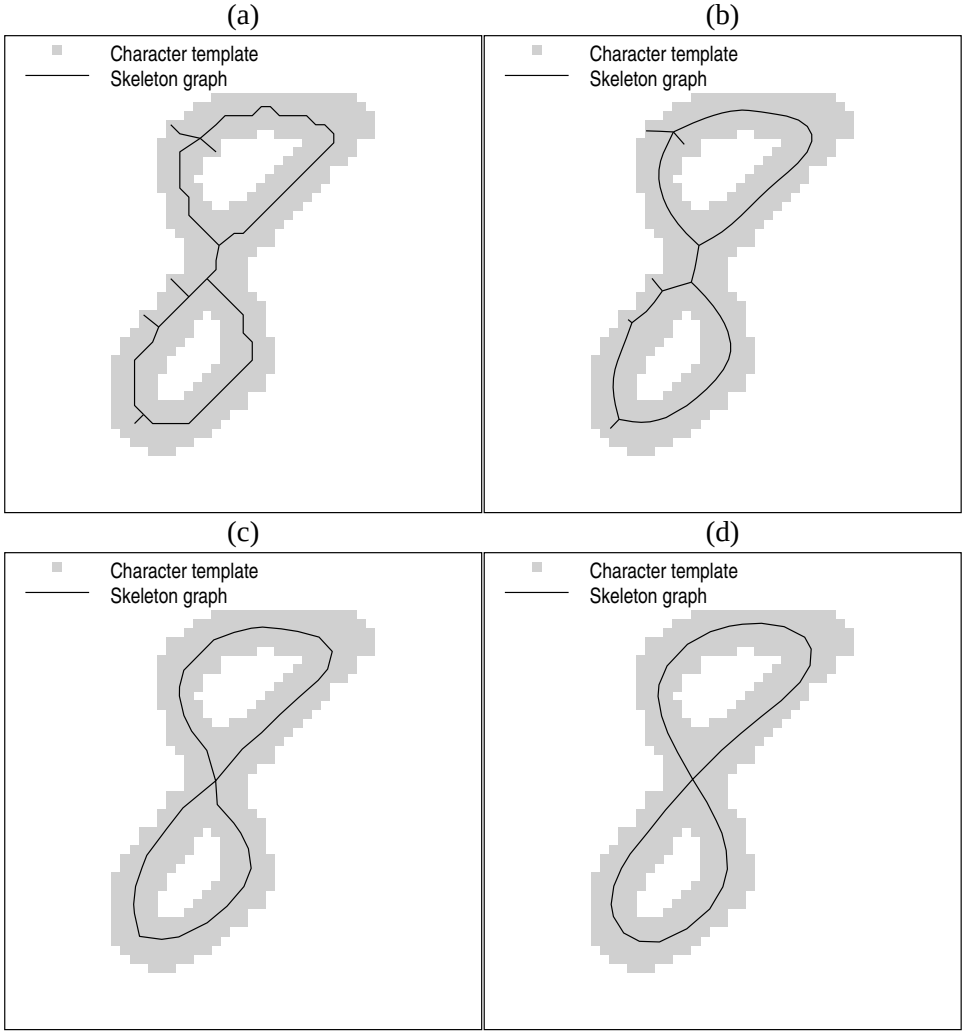


Figure 29: Evolution of the skeleton graph. The skeleton graph produced by the extended polygonal line algorithm (a) after the initialization step, (b) after the first fitting-and-smoothing step, (c) after the restructuring step, and (d) after the second fitting-and-smoothing step (the output of the algorithm).

line-vertex or a *corner-vertex*. The angle at a line-vertex is penalized as in (75), while at a corner vertex we penalize the angle for its deviation from right angle. We introduce three different vertex types of degree three. At a *star3-vertex*, no penalty is imposed. At a *T-vertex*, we penalize one of the three angles for its deviation from straight angle. The remaining two angles are penalized for their deviations from right angle. At a *Y-vertex*, two of the possible angles are penalized for their deviations from straight angle. We use only two of the several possible configurations at a vertex of degree four. At a *star4-vertex* no penalty is imposed, while at an *X-vertex* we penalize sharp angles on the two crossing curves. Vertices of degree three or more are called *junction-vertices*.

In principle, several other types of vertices can be considered. However, in practice we found that these types are sufficient to represent hand-written characters from the Latin alphabet and of the

ten digits. Vertices at the end and in the middle of a curve are represented by end-vertices and line-vertices, respectively. Two curves can be joined at their endpoints by a corner-vertex (Figure 30(a)). The role of a Y-vertex is to "merge" two smooth curves into one (Figure 30(b)). A T-vertex is used to join the end of a curve to the middle of another curve (Figure 30(c)). An X-vertex represents the crossing point of two smooth curves (Figure 30(d)). Star3 and star4-vertices are used in the first fitting-and-smoothing step, before we make the decision on the penalty configuration at a particular junction-vertex.

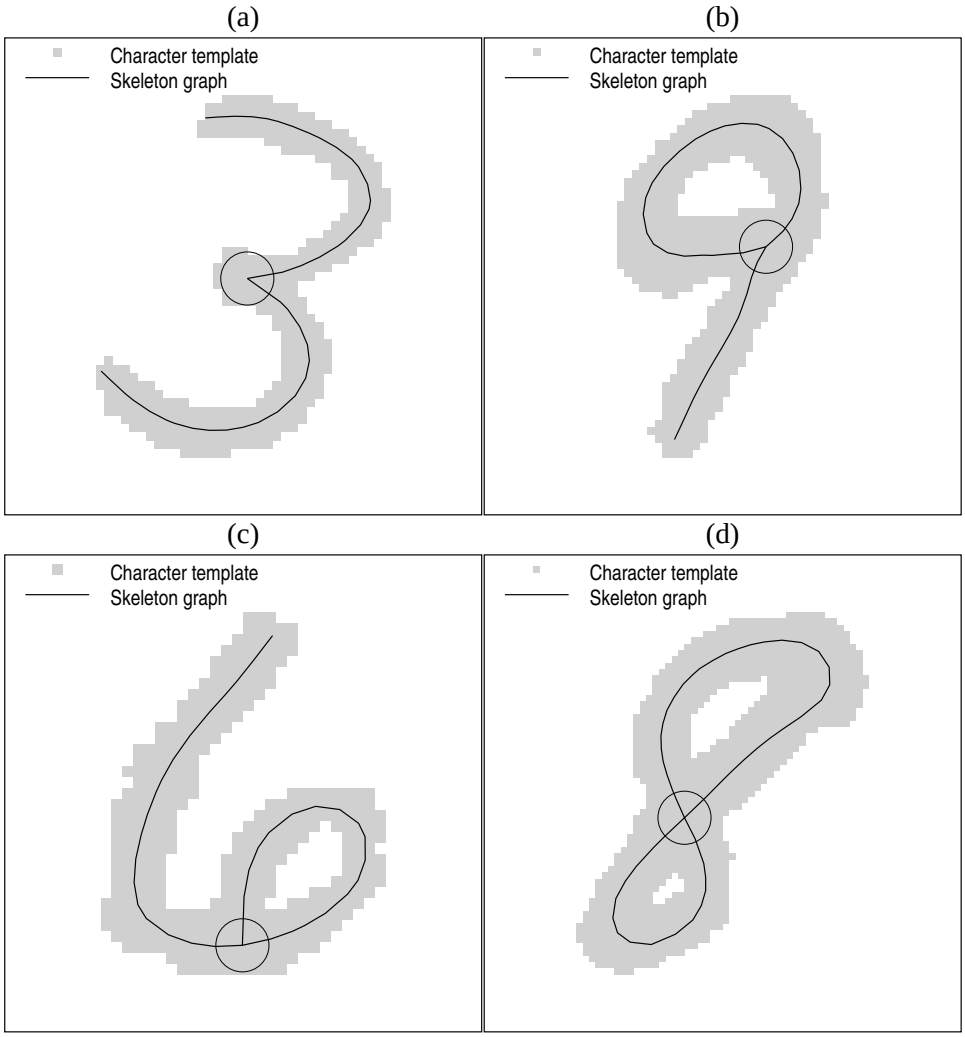


Figure 30: Roles of vertices of different types. (a) A corner-vertex joins two curves at their endpoints. (b) A Y-vertex merges two smooth curves into one. (c) A T-vertex joins the end of a curve to the middle of another curve. (d) An X-vertex represents the crossing point of two smooth curves.

The Local Distance Function

Since the edges can no longer be naturally ordered as in the case of curves, we revise our notation used in Chapter 5 in the formal definition of the formulas of the local distance function and the local penalty. Let $\mathbf{v}_1, \ldots \mathbf{v}_m$ denote the vertices of the graph and let $\mathbf{s}_{ij}$ denote the edge that connects vertices $\mathbf{v}_i$ and $\mathbf{v}_j$. Let V_i and S_{ij} be the nearest neighbor sets of vertex $\mathbf{v}_i$ and edge $\mathbf{s}_{ij}$, respectively, as defined in Section 5.1.4, let $\mathbf{s}'_{ij}$ be the line obtained by the infinite extension of the line segment $\mathbf{s}_{ij}$, and let

$$\begin{aligned} \sigma(\mathbf{s}_{ij}) &= \sum_{\mathbf{x} \in S_{ij}} \Delta(\mathbf{x}, \mathbf{s}'_{ij}), \\ \nu(\mathbf{v}_i) &= \sum_{\mathbf{x} \in V_i} \Delta(\mathbf{x}, \mathbf{v}_i). \end{aligned}$$

(Note that the new notation is a simple generalization of the notation used in Section 5.1.5 as $\nu(\mathbf{v}_i)$ is the same as before, $\sigma_+(\mathbf{v}_i) = \sigma(\mathbf{s}_{i,i+1})$, and $\sigma_-(\mathbf{v}_i) = \sigma(\mathbf{s}_{i,i-1})$.) Then the local distance function of $\mathbf{v}_i$ is defined by

$$\Delta_n(\mathbf{v}_i) = \frac{1}{n} \left(\nu(\mathbf{v}_i) + \sum_{j=1}^{\phi_i} \sigma(\mathbf{s}_{i,i_j}) \right) \tag{86}$$

where ϕ_i is the degree of $\mathbf{v}_i$ ($1 \le \phi_i \le 4$), and $i_1, \ldots, i_{\phi_i}$ are the indices of the adjacent vertices to $\mathbf{v}_i$. Note that (86) is an extension of (74) where $\Delta_n(\mathbf{v}_i)$ is defined for $\phi_i = 1, 2$.

The Local Penalty

For the definition of the local penalty, let $\pi_{ij\ell} = r^2(1 + \cos\gamma_{ij\ell})$ be the angle penalty at $\mathbf{v}_j$ where $\gamma_{ij\ell}$ denotes the angle of line segments $\mathbf{s}_{ji}$ and $\mathbf{s}_{j\ell}$, and let $\mu_{ij} = \|\mathbf{v}_i - \mathbf{v}_j\|^2$ be the length penalty at edge $\mathbf{s}_{ij}$. At corner and T-vertices we introduce $\omega_{ij\ell} = 2r^2 \cos^2\gamma_{ij\ell}$ to penalize the deviation of $\gamma_{ij\ell}$ from right angle. The penalty $P_{\mathbf{v}}(\mathbf{v}_i)$ at vertex $\mathbf{v}_i$ is defined in Table 3 for the different vertex types. (Note that for end and line-vertices, $P_{\mathbf{v}}(\mathbf{v}_i)$ remains as defined in (72).) When the vertex $\mathbf{v}_i$ is moved, only angles at $\mathbf{v}_i$ and at neighbors of $\mathbf{v}_i$ can change. Therefore, the total penalty at $\mathbf{v}_i$ is defined as

$$P(\mathbf{v}_i) = P_{\mathbf{v}}(\mathbf{v}_i) + \sum_{j=1}^{\phi_i} P_{\mathbf{v}}(\mathbf{v}_{i_j}) \tag{87}$$

where ϕ_i is the degree of $\mathbf{v}_i$ ($1 \le \phi_i \le 4$), and $i_1, \ldots, i_{\phi_i}$ are the indices of the adjacent vertices to $\mathbf{v}_i$. Note that (87) is an extension of (75) where $P(\mathbf{v}_i)$ is defined for line and end-vertices. Also note that the definition of the global penalized distance function (68), and hence the discussion in Section 5.1.6, remains valid if $\Delta'_n(\mathbf{f})$ is redefined for a graph $G_{\mathcal{V},\S}$ as

$$\Delta'_n(G_{\mathcal{V},\S}) = \sum_{\mathbf{v} \in \mathcal{V}} \nu(\mathbf{v}) + \sum_{\mathbf{s} \in \S} \sigma(\mathbf{s}).$$

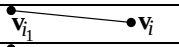
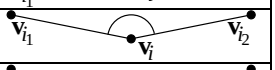
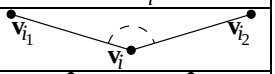
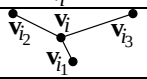
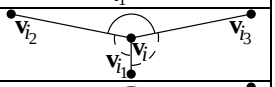
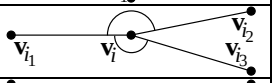
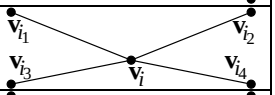
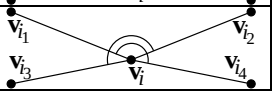
Type of $\mathbf{v}_i$	$\phi(\mathbf{v}_i)$	Penalty at $\mathbf{v}_i$	Configuration
end	1	$P_{\mathbf{v}}(\mathbf{v}_i) = \mu_{i,i_1}$	
line	2	$P_{\mathbf{v}}(\mathbf{v}_i) = \pi_{i_1,i,i_2}$	
corner	2	$P_{\mathbf{v}}(\mathbf{v}_i) = \omega_{i_1,i,i_2}$	
star3	3	$P_{\mathbf{v}}(\mathbf{v}_i) = 0$	
T	3	$P_{\mathbf{v}}(\mathbf{v}_i) = \pi_{i_2,i,i_3} + \omega_{i_1,i,i_2} + \omega_{i_1,i,i_3}$	
Y	3	$P_{\mathbf{v}}(\mathbf{v}_i) = \pi_{i_1,i,i_2} + \pi_{i_1,i,i_3}$	
star4	4	$P_{\mathbf{v}}(\mathbf{v}_i) = 0$	
X	4	$P_{\mathbf{v}}(\mathbf{v}_i) = \pi_{i_1,i,i_4} + \pi_{i_2,i,i_3}$	

Table 3: Vertex types and their attributes. The third column defines the penalties applied at each vertex type. The arcs in the fourth column indicate the penalized angles. The dashed arc indicates that the angle is penalized for its deviation from right angle (rather than for its deviation from from straight angle).

Degrading Vertices

Most of the reconstructing operations described in Section 6.2.3 proceed by deleting noisy edges and vertices from the skeleton graph. When an edge is deleted from the graph, the degrees of the two incident vertices decrease by one. Since there exist more than one vertex types for a given degree, the new types of the degraded vertices must be explicitly specified by degradation rules. When an edge incident to an end-vertex is deleted, we delete the vertex to avoid singular points in the skeleton graph. Line and corner-vertices are degraded to end-vertices, while star4-vertices are degraded to star3-vertices. Any vertex of degree three is degraded to a line-vertex if the remaining angle was penalized for its deviation from straight angle before the degradation, or if the angle is larger than 100 degrees. Otherwise, it is degraded to a corner-vertex. An X-vertex is degraded to a T-vertex if both of the two unpenalized angles are between 80 and 100 degrees, otherwise it is degraded to a Y-vertex. The explicit degradation rules are given in Table 4.

6.2.2 The Initialization Step

The most important requirement for the initial graph is that it approximately capture the topology of the original character template. We use a traditional connectedness-preserving thinning technique that works well for moderately noisy images. If the task is to recover characters from noisy or faded images, this initialization procedure can be replaced by a more sophisticated routine (e.g., the method based on the minimum spanning tree algorithm presented in [SWP98]) without modifying

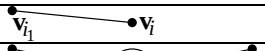
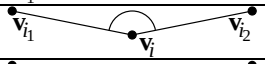
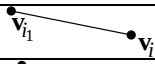
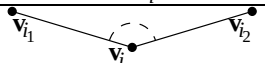
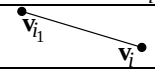
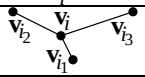
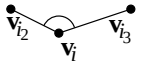
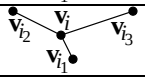
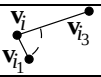
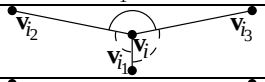
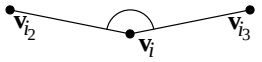
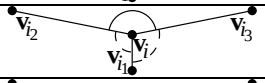
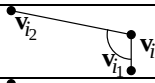
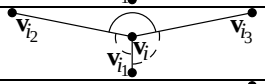
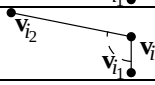
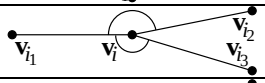
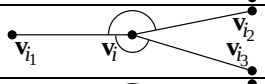
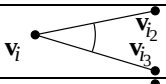
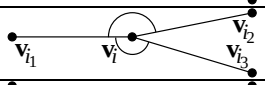
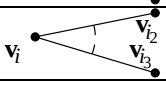
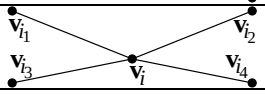
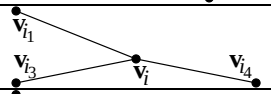
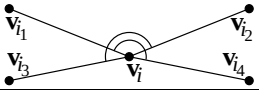
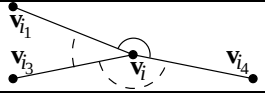
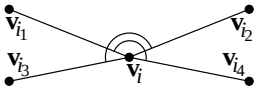
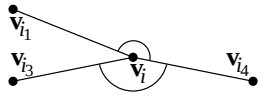
Type (before)	Configuration (before)	Deleted edge	Type (after)	Configuration (after)	Conditions
end		$\mathbf{s}_{i,i_1}$	*deleted*	–	–
line		$\mathbf{s}_{i,i_2}$	end		–
corner		$\mathbf{s}_{i,i_2}$	end		–
star3		$\mathbf{s}_{i,i_1}$	line		$\gamma_{i_2,i,i_3} > 100^\circ$
star3		$\mathbf{s}_{i,i_2}$	corner		$\gamma_{i_1,i,i_3} \leq 100^\circ$
T		$\mathbf{s}_{i,i_1}$	line		–
T		$\mathbf{s}_{i,i_3}$	line		$\gamma_{i_1,i,i_2} > 100^\circ$
T		$\mathbf{s}_{i,i_3}$	corner		$\gamma_{i_1,i,i_2} \leq 100^\circ$
Y		$\mathbf{s}_{i,i_2}$	line		–
Y		$\mathbf{s}_{i,i_1}$	line		$\gamma_{i_2,i,i_3} > 100^\circ$
Y		$\mathbf{s}_{i,i_1}$	corner		$\gamma_{i_2,i,i_3} \leq 100^\circ$
star4		$\mathbf{s}_{i,i_2}$	star3		–
X		$\mathbf{s}_{i,i_2}$	T		$80^\circ \leq \gamma_{i_1,i,i_3} \leq 100^\circ$, $80^\circ \leq \gamma_{i_3,i,i_4} \leq 100^\circ$
X		$\mathbf{s}_{i,i_2}$	Y		not as above, $\gamma_{i_1,i,i_4} > \gamma_{i_1,i,i_3}$, $\gamma_{i_3,i,i_4} > \gamma_{i_1,i,i_3}$

Table 4: Vertex degradation rules.

other modules of the algorithm.

We selected the particular thinning algorithm based on a survey [LLS93] which used several criteria to systematically compare twenty skeletonization algorithms. From among the algorithms that preserve connectedness, we chose the Suzuki-Abe algorithm [SA86] due to its high speed an simplicity. Other properties, such as reconstructability, quality of the skeleton (spurious branches, elongation or shrinkage at the end points), or similarity to a reference skeleton, were less important at this initial phase. Some of the imperfections are corrected by the fitting-and-smoothing operation, while others are treated in the restructuring step. The Suzuki-Abe algorithm starts by computing and storing the distance of each black pixel from the nearest white pixel (distance transformation). In the second step, layers of border pixels are iteratively deleted until pixels with locally maximal

distance values are reached. Finally, some of the remaining pixels are deleted so that connectedness is preserved and the skeleton is of width one.

After thinning the template, an initial skeleton graph is computed (Figure 31). In general, midpoints of pixels of the skeleton are used as vertices of the graph, and two vertices are connected by an edge if the corresponding pixels are eight-neighbors, i.e., if they have at least one common corner. To avoid short circles and crossing edges near junctions of the skeleton, neighboring junction pixels (pixels having more than two eight-neighbors) are recursively placed into pixel-sets. For such a set, only one vertex is created in the center of gravity of the pixels' midpoints. This junction-vertex is then connected to vertices representing pixels neighboring to any of the junction pixels in the set. In this initial phase, only end, line, star3, and star4-vertices are used depending on whether the degree of the vertex is one, two, three, or four, respectively. In the rare case when a vertex representing a set of junction pixels has more than four neighbors, the neighbors are split into two or more sets of two or three vertices. Each neighbor in a set is then connected to a mutual junction-vertex, and the created junction-vertices are connected to each other. The circled vertices in Figures 31(c) and (d) demonstrate this case.

6.2.3 The Restructuring Step

The restructuring step complements the two fitting-and-smoothing steps. In the fitting-and-smoothing step we relocate vertices and edges of the skeleton graph based on their positions relative to the template, but we do not modify the skeleton graph in a graph theoretical sense. In the restructuring step we use geometric properties of the skeleton graph to modify the configuration of vertices and edges. We do not explicitly use the template, and we do not move vertices and edges of the skeleton graph in this step.

The double purpose of the restructuring step is to eliminate or rectify imperfections of the initial skeleton graph, and to simplify the skeletal description of the template. Below we define operations that can be used to modify the configuration of the skeleton graph. Since the types of the imperfections depend on properties of both the input data and the initialization method, one should carefully select the particular operations and set their parameters according to the specific application. At the description of the operations, we give approximate values for each parameter based on our experiments with a wide variety of real data. Specific settings will be given in Section 6.3 where we present the results of two particular experiments.

For the formal description of the restructuring operations, we define some simple concepts. We call a list of vertices $p_{i_1,\ldots,i_\ell} = (\mathbf{v}_{i_1},\ldots,\mathbf{v}_{i_\ell}), \ell > 1$ a *path* if each pair of consecutive vertices $(\mathbf{v}_{i_j},\mathbf{v}_{i_{j+1}}), j = 1,\ldots,\ell-1$ is connected by an edge. A *loop* is a path $p_{i_1,\ldots,i_\ell}$ such that $i_1 = i_\ell$ and none of the inner vertices $\mathbf{v}_{i_2},\ldots,\mathbf{v}_{i_{\ell-1}}$ are equal to each other or to $\mathbf{v}_{i_1}$. The *length* of a path is

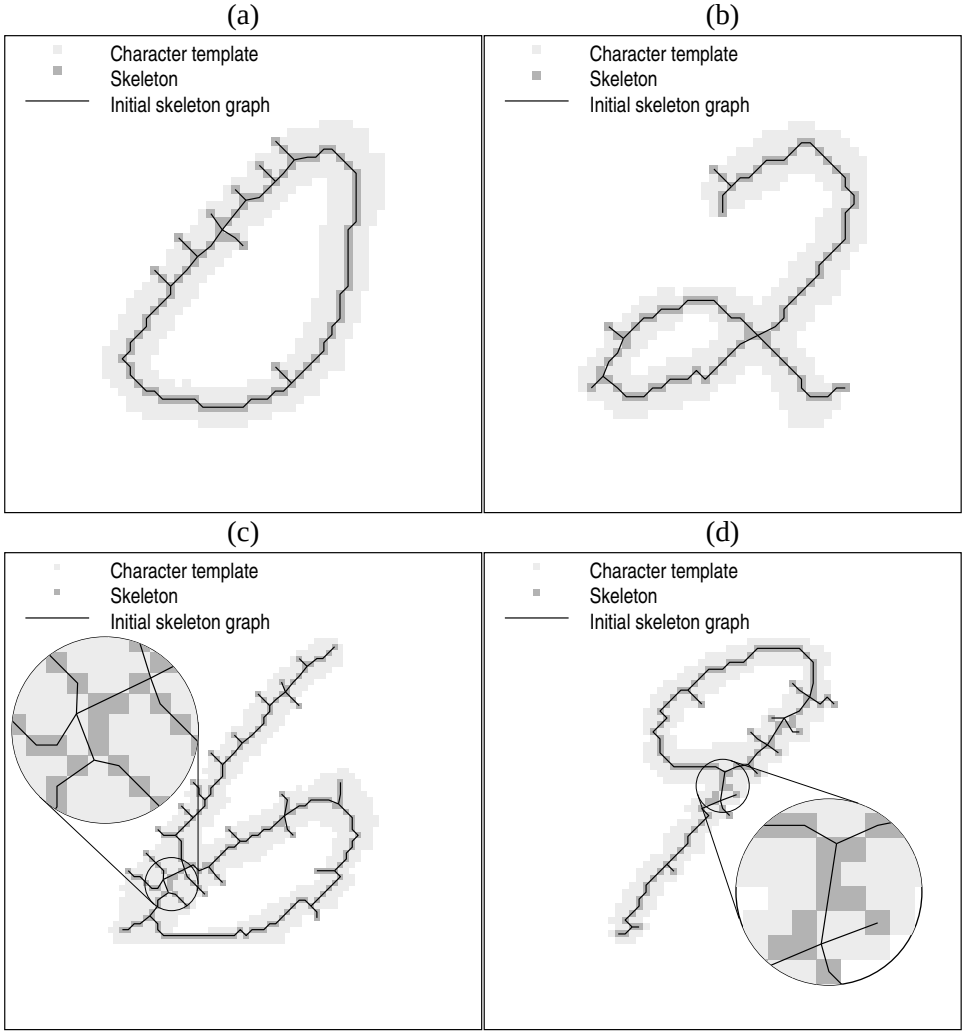


Figure 31: Examples of transforming the skeleton into an initial skeleton graph.

defined by

$$l(p_{i_1,\ldots,i_\ell}) = \sum_{j=1}^{\ell-1} l(\mathbf{s}_{i_j,i_{j+1}}) = \sum_{j=1}^{\ell-1} \|\mathbf{v}_{i_{j+1}} - \mathbf{v}_{i_j}\|.$$

A path $p_{i_1,\ldots,i_\ell}$ is *simple* if its endpoints $\mathbf{v}_{i_1}$ and $\mathbf{v}_{i_\ell}$ are not line-vertices, while all its inner vertices $\mathbf{v}_{i_2},\ldots,\mathbf{v}_{i_{\ell-1}}$ are line-vertices. A simple path is called a *branch* if at least one of its endpoints is an end-vertex. When we *delete* a simple path $p_{i_1,\ldots,i_\ell}$, we remove all inner vertices $\mathbf{v}_{i_2},\ldots,\mathbf{v}_{i_{\ell-1}}$ and all edges $\mathbf{s}_{i_1,i_2},\ldots,\mathbf{s}_{i_{\ell-1},i_\ell}$. Endpoints of the path $\mathbf{v}_{i_1}$ and $\mathbf{v}_{i_\ell}$ are degraded as specified by Table 4. Figure 32 illustrates these concepts.

Most of the reconstructing operations simplify the skeleton graph by eliminating certain simple paths that are shorter then a threshold. To achieve scale and resolution independence, we use the *thickness* of the character as the yardstick in length measurements. We estimate the thickness of the

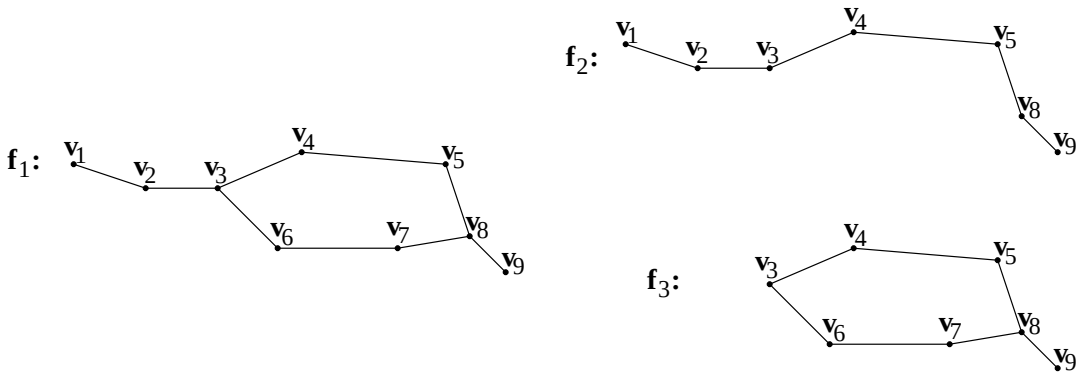


Figure 32: Paths, loops, simple paths, branches, and deletion. A loop in $\mathbf{f}_1$ is $p_{3458763}$. Simple paths of $\mathbf{f}_1$ are p_{123}, p_{3458}, p_{3678}, and p_{89}. p_{123} and p_{89} are branches of $\mathbf{f}_1$. $\mathbf{f}_2$ and $\mathbf{f}_3$ were obtained by deleting p_{3678} and p_{123}, respectively, from $\mathbf{f}_1$.

data set $\mathcal{X}_n = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ by

$$\tau = 4 \sum_{i=1}^{n} \sqrt{\Delta_n(\mathbf{x}_i, G)}.$$

Deleting Short Branches

Small protrusions on the contours of the character template tend to result in short branches in the initial skeleton graph. We first approached this problem by deleting any branch that is shorter than a threshold, τ_{branch}. Unfortunately, this approach proved to be too simple in practice. By setting τ_{branch} to a relatively large value, we eliminated a lot of short branches that represented "real" parts of the character, whereas by setting τ_{branch} to a relatively small value, a lot of "noisy" branches remained in the graph. We found that after the first fitting-and-smoothing step, if the size of the protrusion is comparable to the thickness of the skeleton, i.e., the protrusion is likely to be a "real" part of the skeleton, the angles of the short branch and the connecting paths tend to be close to right angle (Figure 33(a)). On the other hand, if the short branch has been created by the noisy contour of the character, the angle of the short branch and one of the connecting path tends to be very sharp (Figure 33(b)). So, in the decision of deleting the short branch $p_{i,i_3,\ldots}$ (Figure 33), we weight the length of the branch by

$$w_i = 1 - \cos^2 \gamma$$

where

$$\gamma = \min(\gamma_{i_1,i,i_3}, \gamma_{i_2,i,i_3}),$$

and we delete $p_{i,i_3,\ldots}$ if $w_i l(p_{i,i_3,\ldots}) < \tau_{branch}$. Experiments showed that to delete most of the noisy branches without removing essential parts of the skeleton, τ_{branch} should be set between τ and 2τ. Figure 34 shows three skeleton graphs before and after the deletions. To avoid recursively pruning longer branches, we found it useful to sort the short branches in increasing order by their length, and

deleting them in that order. This was especially important in the case of extremely noisy skeleton graphs such as depicted by Figures 34(a) and (b).

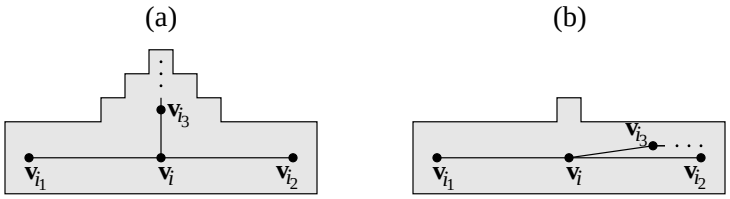


Figure 33: If the protrusion is a "real" part of the skeleton, the angles of the short branch and the connecting paths tend to be close to right angle (a), whereas if the short branch has been created by a few noisy pixels on the contour of the character, the short branch tends to slant to one of the connecting paths during the first fitting-and-smoothing step (b).

Removing Short Loops

Short loops created by thinning algorithms usually indicate isolated islands of white pixels in the template. We remove any loop from the skeleton graph if its length is below a threshold τ_{loop}. A loop is removed by deleting the longest simple path it contains. Experiments showed that to remove most of the noisy loops without removing essential parts of the skeleton, τ_{loop} should be set between 2τ and 3τ. Figure 35 shows three skeleton graphs before and after the operation.

Merging Star3-Vertices

In experiments we found that if two penstrokes cross each other at a sharp angle, the thinning procedure tends to create two star3-vertices connected by a short simple path rather then a star4-vertex. The first approach to correct this imperfection was to merge any two star3-vertices that are connected by a simple path shorter than a threshold, τ_{star3}. Unfortunately, this approach proved to be too simple in practice. By setting τ_{star3} to a relatively large value, we eliminated a lot of short paths that were not created by crossing penstrokes, whereas by setting τ_{star3} to a relatively small value, a lot of paths created by crossing penstrokes remained in the graph. We found that it is more likely that the short path $p_{i,\ldots,j}$ (Figure 36) is created by two crossing penstrokes if the angles γ_{i_1,i,i_2} and γ_{j_1,j,j_2} are small. To avoid merging $\mathbf{v}_i$ and $\mathbf{v}_j$ when these two angles are large, we weight the length of the path $p_{i,\ldots,j}$ by an experimentally developed factor

$$w_{ij} = \left[\frac{(1-\cos\gamma_{i_1,i,i_2}) + (1-\cos\gamma_{j_1,j,j_2})}{4}\right]^3,$$

and we merge $\mathbf{v}_i$ and $\mathbf{v}_j$ if $w_{ij}l(p_{i,\ldots,j}) < \tau_{star3}$. When merging two star3-vertices $\mathbf{v}_i$ and $\mathbf{v}_j$, we first delete the path $p_{i,\ldots,j}$, and remove $\mathbf{v}_i$ and $\mathbf{v}_j$. Then we add a new vertex $\mathbf{v}_{new}$ and connect the four

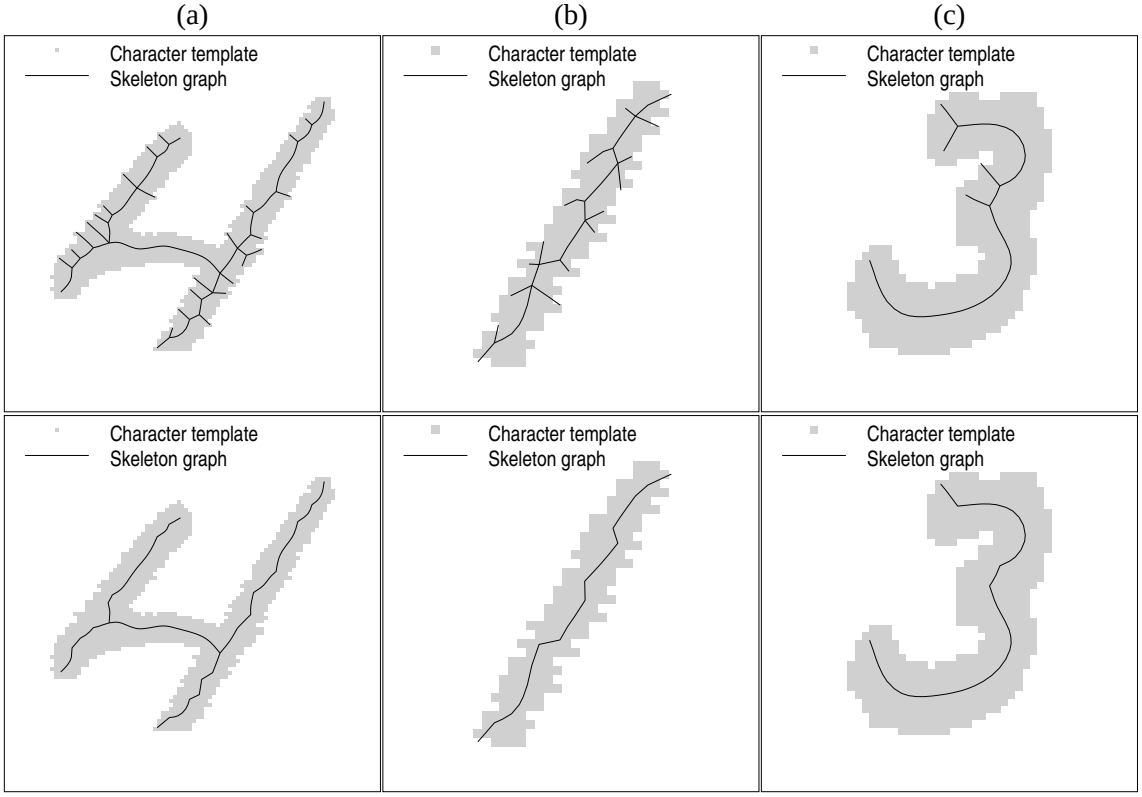


Figure 34: Deleting short branches. Skeleton graphs before (top row) and after (bottom row) the deletion.

remaining neighbors of $\mathbf{v}_i$ and $\mathbf{v}_j$ to $\mathbf{v}_{new}$ (Figure 36). Experiments indicated that for the best results τ_{star3} should be set between 0.5τ and τ. Figure 37 shows three skeleton graphs before and after the merge.

Updating Star3 and Star4-Vertices

Initially, all the junction-vertices of the skeleton are either star3 or star4-vertices. After the skeleton has been smoothed by the first fitting-and-smoothing step and cleaned by the restructuring operations described above, we update the junction-vertices of the skeleton to Y, T and X-vertices depending on the local geometry of the junction-vertices and their neighbors. A star4-vertex is always updated to an X-vertex. When updating a star3-vertex, we face the same situation as when degrading an X-vertex, so a star3-vertex is updated to a T-vertex if two of the angles at the vertex are between 80 and 100 degrees, otherwise it is updated to a Y-vertex. The formal rules are given in the last two rows of Table 4.

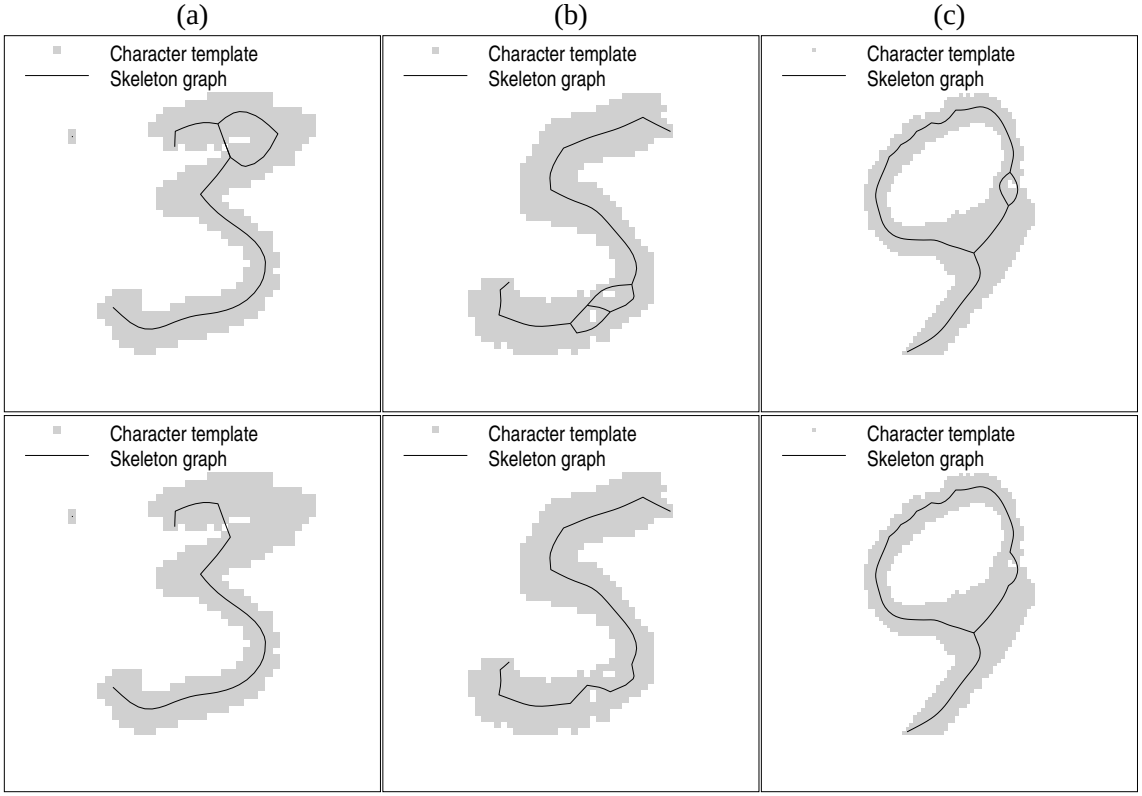


Figure 35: Removing short loops. Skeleton graphs before (top row) and after (bottom row) the removal.

Filtering Vertices

In this step, we iteratively remove every line-vertex whose two incident edges are shorter than a threshold τ_{filter}. Formally, any line-vertex $\mathbf{v}_j$ is removed from the graph if

$$\|\mathbf{v}_j - \mathbf{v}_i\| < \tau_{filter} \quad \text{and} \quad \|\mathbf{v}_j - \mathbf{v}_\ell\| < \tau_{filter}$$

where $\mathbf{v}_i$ and $\mathbf{v}_\ell$ are the neighbors of $\mathbf{v}_j$. When a line-vertex $\mathbf{v}_j$ is removed, the two edges incident to $\mathbf{v}_j$, $\mathbf{s}_{ij}$ and $\mathbf{s}_{j\ell}$, are also removed. Then the two former neighbors of $\mathbf{v}_j$ are connected by a new edge (Figure 38).

Filtering vertices is an optional operation. It can be used to speed up the optimization if the character template has a high resolution since in this case the initial skeleton graph has much more vertices than it is needed for reasonably smooth approximation. It can be also used after the optimization to improve the compression rate if the objective is to compress the image by storing the character skeleton instead of the template. In this case the filtering operation can be coupled with a smoothing operation at the other end where the character is recovered based on the skeleton graph (see Section 6.3.2). Figure 39 shows an example of a skeleton graph before and after the filtering operation.

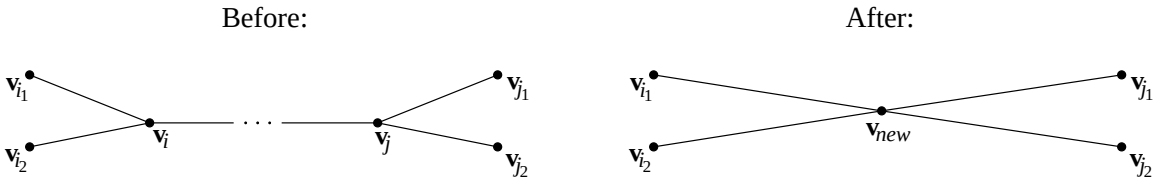


Figure 36: When merging two star3-vertices, we remove the vertices and the path connecting them. Then we add a new vertex and connect the former neighbors of the two star3-vertices to the new vertex.

6.3 Experimental Results

6.3.1 Skeletonizing Isolated Digits

In this section we report results on isolated hand-written digits from the NIST Special Database 19 [Gro95]. To set the parameters and to tune the algorithm, we chose 100 characters per digit randomly. Figures 40 through 49 display eight templates for each digit. These examples were chosen so that they roughly represent the 100 characters both in terms of the variety of the input data and in terms of the success rate of the algorithm. To illuminate the contrast between the pixelwise skeleton of the character and the skeleton graph produced by the principal graph algorithm, we show the initial graph (upper row in each figure) and the final graph (lower row in each figure) for each chosen character template. The length thresholds of the restructuring operations were set to the values indicated by Table 5.

τ_{branch}	τ_{loop}	τ_{star3}	τ_{filter}
1.2τ	3τ	τ	τ

Table 5: Length thresholds of the restructuring operations in experiments with isolated digits.

The results indicate that the principal graph algorithm finds a smooth medial axis of the great majority of the characters. In the few cases when the skeleton graph is imperfect, we could identify two sources of errors. The first cause is that, obviously, the restructuring operations do not work perfectly for all the characters. For instance, short branches can be cut (Figure 46(a)), short loops can be eliminated (Figure 42(c)), or star3-vertices can be merged mistakenly (Figure 44(h)). To correct these errors, one has to include some a-priori information in the process, such as a collection of possible configurations of skeleton graphs that can occur in hand-written digits. The other source of errors is that at this phase, we do not have restructuring operations that *add* components to the skeleton graph. For instance, skeleton graphs in Figures 42(e) and 48(d) could be improved by connecting broken lines based on the closeness of their endpoints. One could also add short paths to create branches or loops that were missing from the initial graph (Figures 42(b) and 48(f)). This operation could be based on local thickness measurements along the graph that could point out protrusions caused by overlapping lines in the character. The exact definitions and implementations

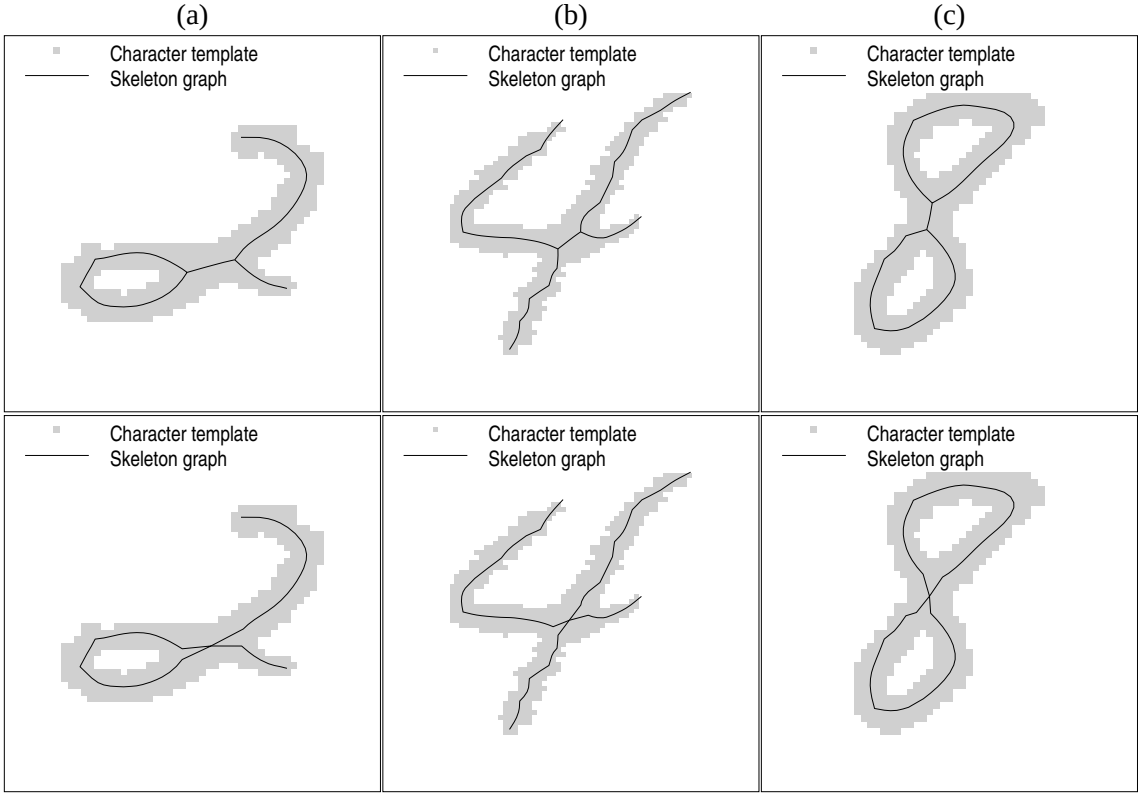


Figure 37: Merging star3-vertices. Skeleton graphs before (top row) and after (bottom row) the merge.

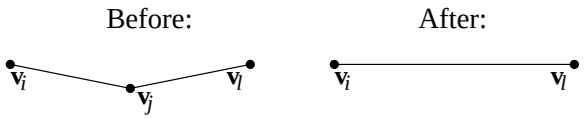


Figure 38: Removing the line-vertex $\mathbf{v}_j$ in the filtering operation.

of these operations are subjects of future research.

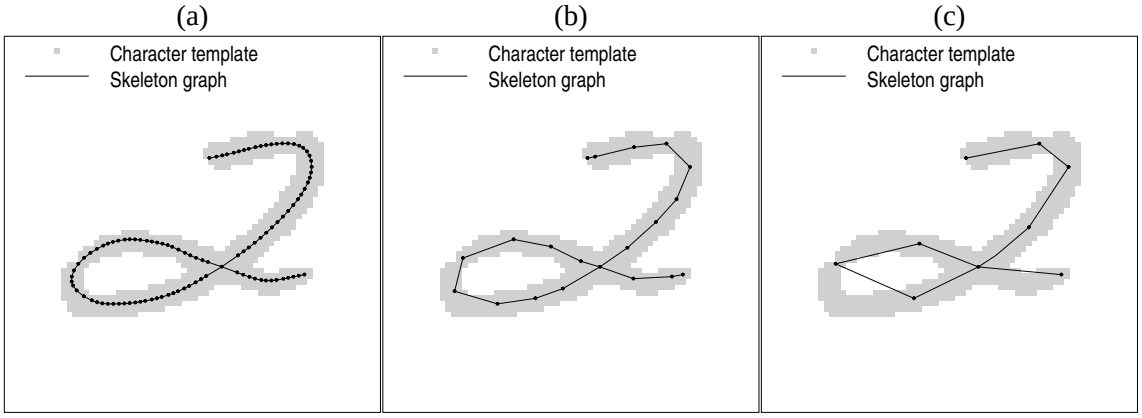


Figure 39: Filtering vertices. A skeleton graph (a) before filtering, (b) after filtering with $\tau_{filter} = 1$, and (b) after filtering with $\tau_{filter} = 2$.

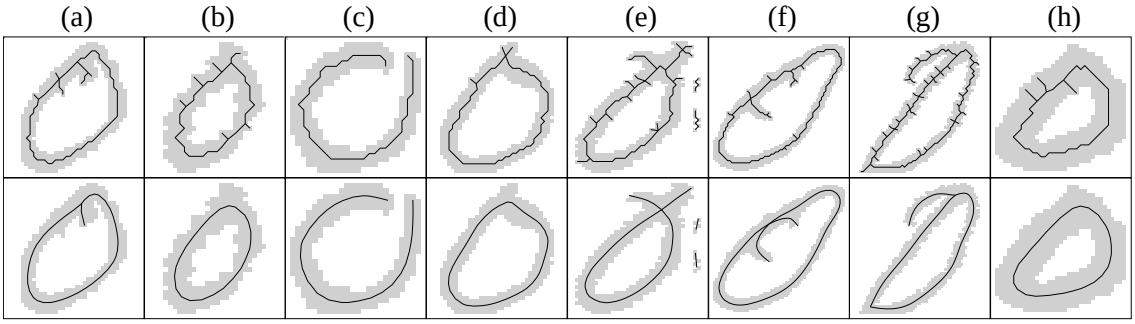


Figure 40: Skeleton graphs of isolated 0's. Initial (upper row) and final (lower row) skeletons.

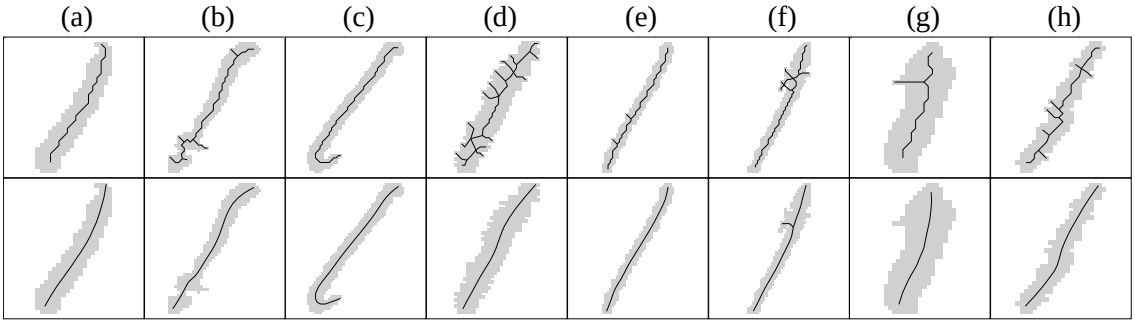


Figure 41: Skeleton graphs of isolated 1's. Initial (upper row) and final (lower row) skeletons.

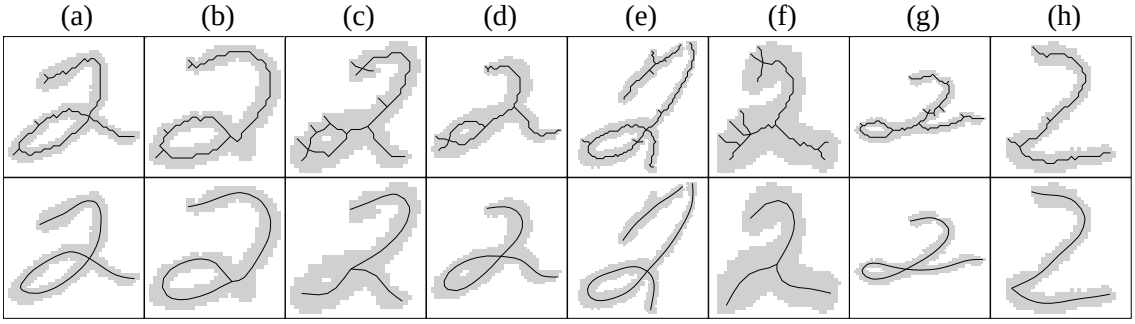


Figure 42: Skeleton graphs of isolated 2's. Initial (upper row) and final (lower row) skeletons.

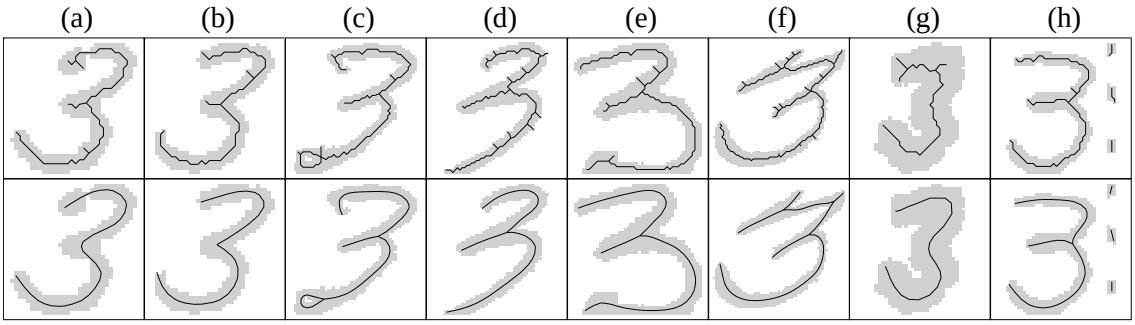


Figure 43: Skeleton graphs of isolated 3's. Initial (upper row) and final (lower row) skeletons.

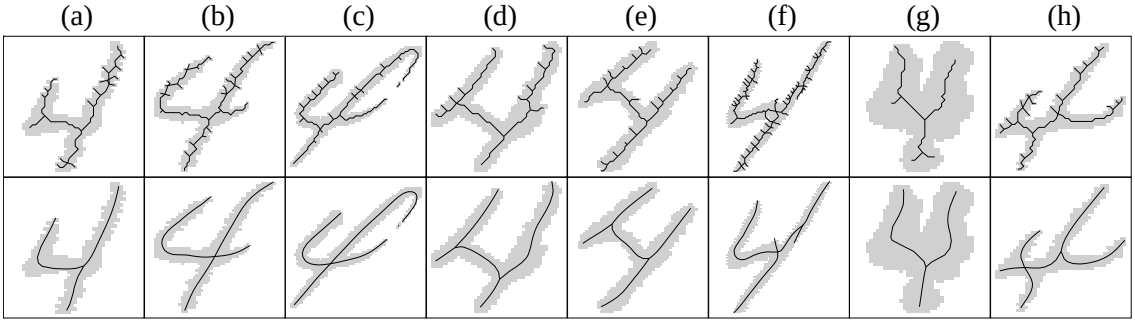


Figure 44: Skeleton graphs of isolated 4's. Initial (upper row) and final (lower row) skeletons.

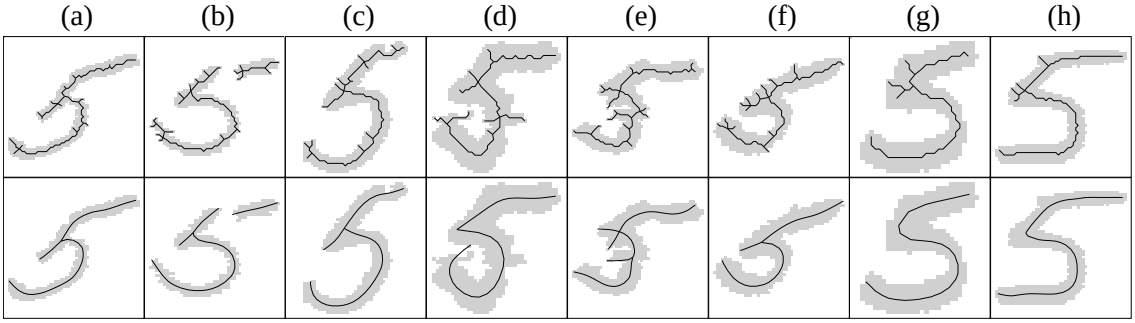


Figure 45: Skeleton graphs of isolated 5's. Initial (upper row) and final (lower row) skeletons.

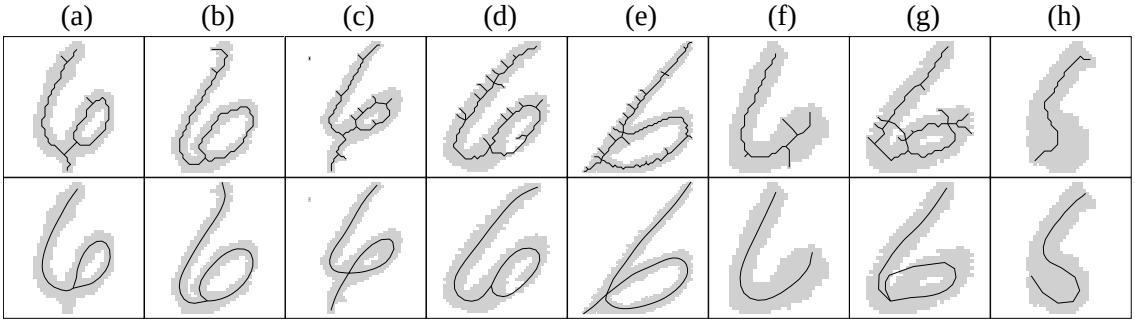


Figure 46: Skeleton graphs of isolated 6's. Initial (upper row) and final (lower row) skeletons.

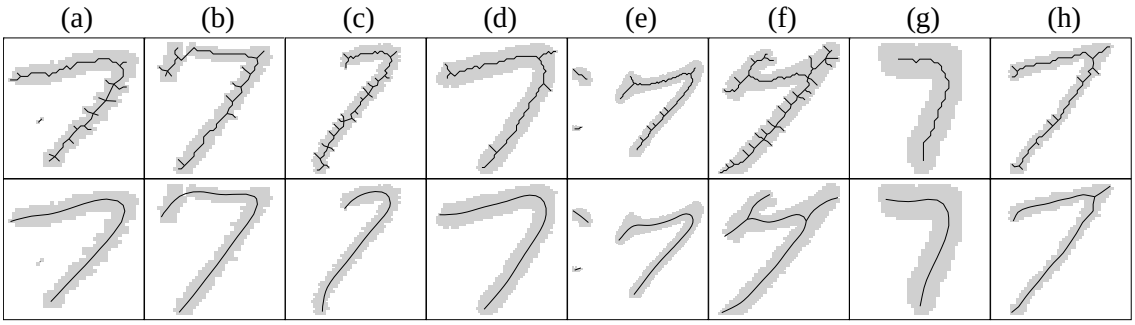


Figure 47: Skeleton graphs of isolated 7's. Initial (upper row) and final (lower row) skeletons.

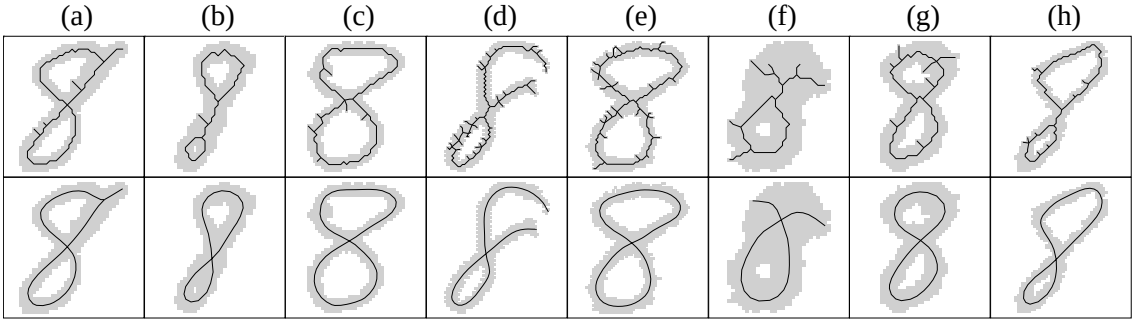


Figure 48: Skeleton graphs of isolated 8's. Initial (upper row) and final (lower row) skeletons.

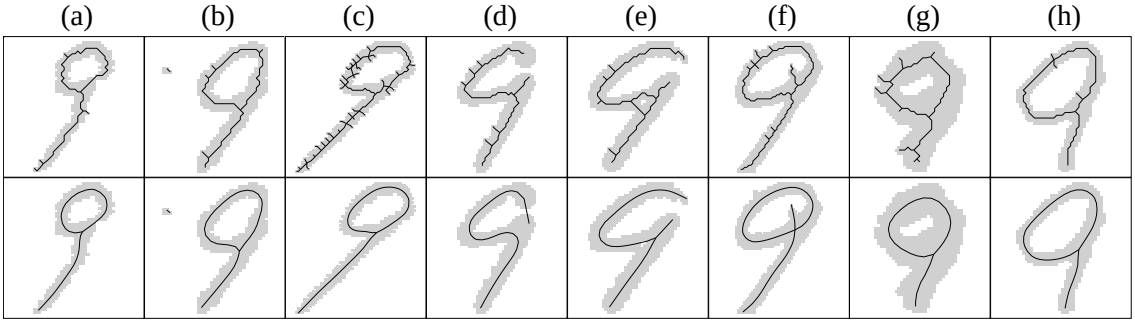


Figure 49: Skeleton graphs of isolated 9's. Initial (upper row) and final (lower row) skeletons.

6.3.2 Skeletonizing and Compressing Continuous Handwriting

In this section we present results of experiments with images of continuous handwriting. We used the principal graph algorithm to skeletonize short pangrams (sentences that contain all the letters of the alphabet) written by different individuals. The emphasis in these experiments was on using the skeleton graph for representing hand-written text efficiently.

Figure 50 shows the images of two pangrams written by two individuals. For the sake of easy referencing, hereafter we will call them Alice (Figure 50(a)) and Bob (Figure 50(b)). After scanning the images, the principal graph algorithm was used to produce the skeleton graphs depicted by Figure 51. Since the images were much cleaner than the images of isolated digits used in the previous section, τ_{branch} and τ_{loop} were set slightly lower than in the previous experiments. We also found that the incorrect merge of two star3-vertices has a much worse visual effect than not merging two star3-vertices when they should be merged, so we set τ_{star3} to half of the value that was used in the experiments with isolated digits. Finally, we did not use filtering vertices in the restructuring step. The length thresholds of the restructuring operations were set to the values indicated by Table 6. The thickness of each curve in Figure 51 was set to the estimated thickness τ of the template.

τ_{branch}	τ_{loop}	τ_{star3}	τ_{filter}
τ	2τ	0.5τ	0

Table 6: Length thresholds of the restructuring operations in experiments with continuous handwriting. $\tau_{filter} = 0$ indicates that we did not filter vertices in the reconstruction step.

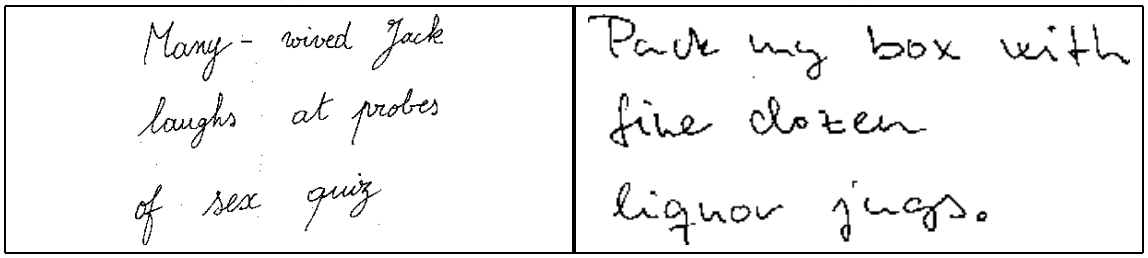


Figure 50: Original images of continuous handwritings. (a) Alice, (b) Bob.

To demonstrate the efficiency of representing the texts by their skeleton graphs, we applied the vertex filtering operation *after* the skeleton graphs were produced. For achieving high compression rate, τ_{filter} should be set to a relatively large value to remove most of the line-vertices from the skeleton graph. Since filtering with a large threshold has an unfortunate visual effect of producing sharp-angled polygonal curves (see Figure 39(c)), we fit cubic splines through the vertices of each path of the skeleton graph. Tables 7 and 8 show the results.

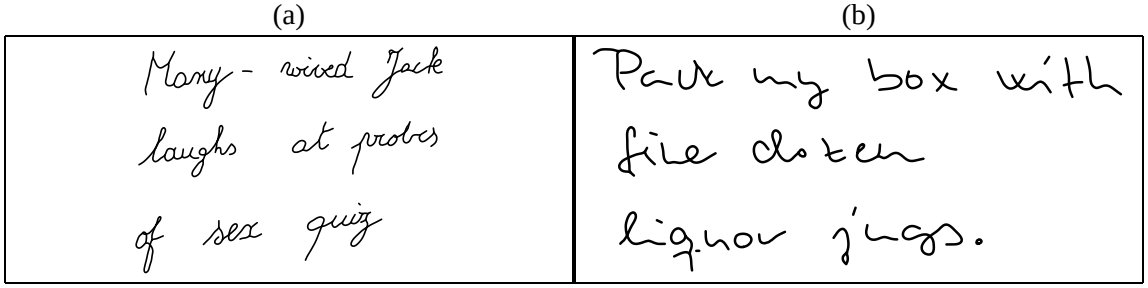


Figure 51: Skeleton graphs of continuous handwritings. (a) Alice, (b) Bob.

To be able to compute the number of bytes needed for storing the images, in the compression routine we also set the number of bits n_b used to store each coordinate of a vertex. The vertices are stored consecutively with one bit sequence of length n_b marking the end of a path. So, for example, when n_b is set to 8, the vertices of the skeleton graph are rounded to the points of a 255×255 rectangular grid, and the remaining byte is used to mark the end of a path. By using this scheme, the skeleton graph can be stored by using

$$N = \lceil (n_p + 2m) n_b / 8 \rceil$$

bytes where n_p is the number of paths and m is the number of vertices. Tables 7 and 8 show the skeleton graphs and the number of bytes needed to store the images. The numbers of paths in Alice's and Bob's texts are 148 and 74, respectively. As a comparison, the size of the raw bitmap compressed by using the Lempel-Ziv algorithm (gzip under UNIX) is 2322 bytes in Alice's case, and 1184 bytes in Bob's case. So, for instance, if the filter threshold is set to 6τ, and 8 bits are used to store the coordinates of the vertices, the algorithm produces a skeleton that approximates the original text quite well while compressing the image to less than half of the size of the gzipped raw bitmap. Note that the bit sequence representing the skeleton graph can be further compressed by using traditional compression methods.

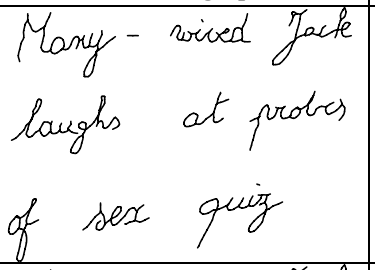
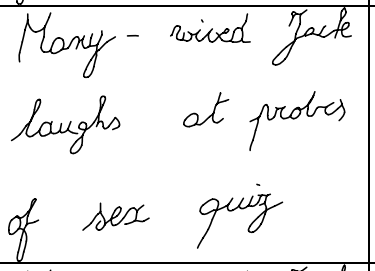
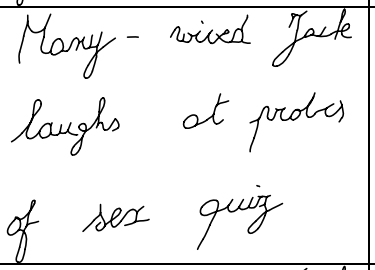
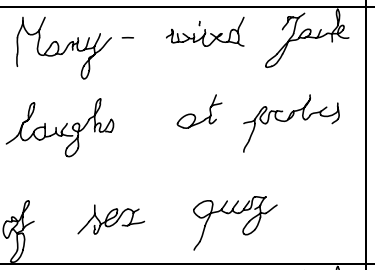
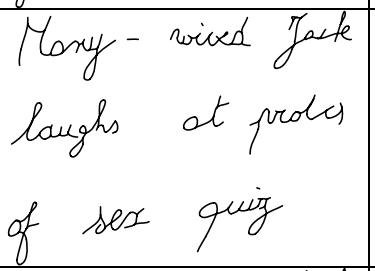
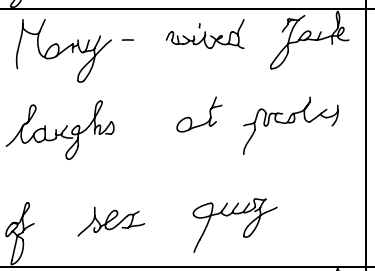
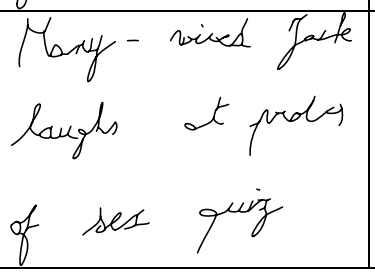
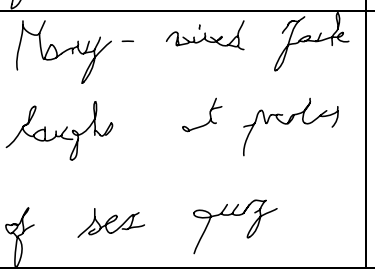
τ_{filter}	m	$n_b = 8$		$n_b = 6$	
		Skeleton graph	N	Skeleton graph	N
2τ	743		1634	–	–
4τ	492		1132	–	–
6τ	408		964		723
10τ	361		870		653
20τ	324		796		597

Table 7: Compression of Alice's handwriting. m is the number of vertices, n_b is the number of bits used to store each coordinate of a vertex, and N is the total number of bytes needed to store the skeleton graph of the image.

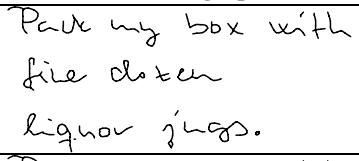
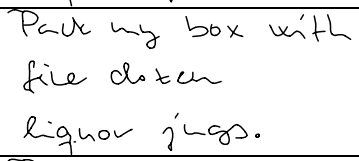
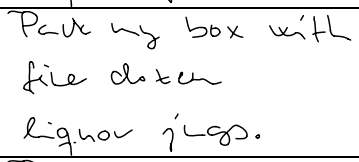
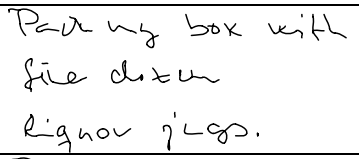
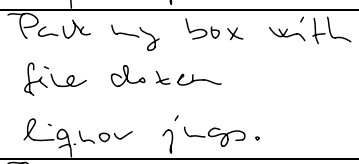
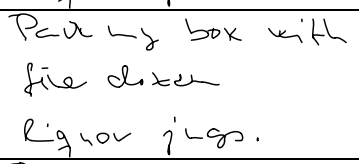
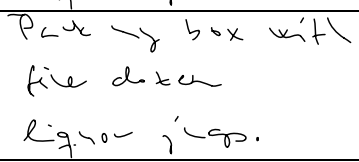
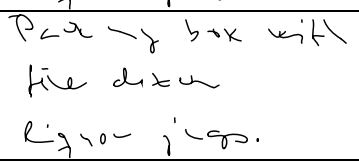
τ_{filter}	m	$n_b = 8$		$n_b = 6$	
		Skeleton graph	N	Skeleton graph	N
2τ	435		944	–	–
4τ	280		634	–	–
6τ	225		524		393
10τ	195		464		348
20τ	173		420		315

Table 8: Compression of Bob's handwriting. m is the number of vertices, n_b is the number of bits used to store each coordinate of a vertex, and N is the total number of bytes needed to store the skeleton graph of the image.