SKELETONIZING BINARY PATTERNS BY USING LINE ADJACENCY GRAPHS

Serban Iliescu
Speedware Corporation
3300 Côte Vertu, Suite 303, St. Laurent, Québec, Canada H4R 2B8
email: siliescu@speedware.com

Rajjan Shinghal
Department of Computer Science, Concordia University
1455 de Maisonneuve West Blvd., Montréal, Québec, Canada H3G 1M8
email: shinghal@cs.concordia.ca

Abstract

The skeletonization algorithm of [5] is extended in decomposing patterns into nodes, each node depending on the local configuration of the pattern at that node. Rather than use a single skeletonization procedure over the entire pattern, we employ specialized procedures for each node, that is for each local configuration found in the pattern. Experiments demonstrate that our skeletonization procedures are robust and fast. Moreover, they produce very good skeletons.

1 Introduction

Skeletonization of a binary pattern consists of deleting some of the black pixels (that is, changing them to white) so that the pattern is thinned to a line drawing, called the *skeleton*, which must preserve the connectedness and shape of the pattern. Ideally, the skeleton should (1) be unitary, that is, its strokes should be one pixel thick, and (2) it should lie along the medial axis.

There are two principle approaches to skeletonization: *peeling* and *shelling*. In peeling, the pixels along the edges of the pattern are deleted iteratively until the pattern is thinned to a line drawing. Most of the skeletonization algorithms described in the literature adopt a peeling approach [3], [4], [6]. In shelling, all pixels to be deleted for obtaining the skeleton are first designated. The designated pixels are then deleted all at once, for example, the algorithm of [1].

In this paper, we present a shelling skeletonization algorithm built around the concept of a Line Adjacency Graph (LAG), that is, the representation of the binary pattern as a list of segments of black pixels. LAGs can be employed for noise removal procedures as well [2]. In this paper, we shall assume that our skeletonization procedures are applied to noise-free patterns.

Pavlidis [5] showed that skeletonization using LAGs is sensitive to minor variations in the pattern. He gave an example of two patterns of the letter K that were nearly alike, but not identical. But the skeletons produced from these patterns were considerably different (Fig. 1).

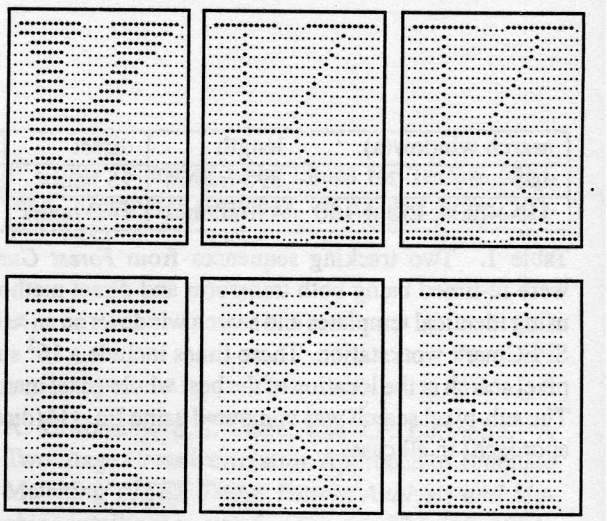

Fig. 1. For two similar character patterns (leftmost column), the skeletons produced by Pavlidis's algorithm [5] are not similar (center column), whereas they are so with our algorithm (right-most column).

In this paper, we start by revising the definition of LAGs and their procedure for condensation. By doing so, we have experimentally observed that we reduce substantially the sensitivity of the skeleton produced to minor variations in the pattern (Fig. 1).

Section 2 explains how LAGs are constructed from a given pattern. Then Section 3 discusses how LAGs can be decomposed to take into consideration local configurations in the pattern. Skeletonization procedures are in Section 4, the description of our experiments in Section 5, and concluding remarks in Section 6. We assume that the reader is familiar with standard pattern recognition terminology.

2 Line Adjacency Graphs

The LAG is the encoding of a binary pattern as a list of segments. A *segment* is defined to be a sequence of consecutive black pixels in a row. It can be denoted by the triple $<r, L, R>$, which indicates that, in row r of the pattern, all pixels from column L to column R, both inclusive, are black. L and R thus denote the *limits* of the segment, and the length of the segment is $R - L + 1$. The *mid-pixel* of the segment is defined to be in the column $\lceil \frac{L + R}{2} \rceil$.

Two black segments, $b_1 = <r_1, L_1, R_1>$ and $b_2 = <r_2, L_2, R_2>$, are said to be *friends* if, and only if, (1) they lie on adjacent rows ($r_1 = r_2 \pm 1$), and (2) a pixel in b_1 is one of the 8-neighbours of a pixel in b_2.

Our definition of friendly segments is slightly different from that given of "neighbours" in [5]. Our definition allows the pattern to retain its 8-connectivity, which helps in developing good quality skeletons.

If a segment b in row r has N friends in row $r - 1$, then b is said to have N *north friends*. Similarly, if b has S friends in row $r + 1$, then b is said to have S *south friends*. From now on, we shall delineate any black segment b by its five-tuple $<r, L, R, N, S>$, where N is the number of *north friends* of the segment, and S is the number of *south friends* of the segment. Representing a pattern by its set of five-tuples of segments gives us the LAG of the pattern.

To obtain the skeleton from the pattern we have next to condense the LAG into *nodes*, which are groups of one or more segments, each group containing the essential shape information of a part of the pattern. Our condensation is different from that in [5], and it is our condensation approach that makes skeletons less sensitive to minor pattern variations.

We begin by putting each segment of the LAG into a separate node, and then we attempt to iteratively merge those nodes pair-wise, such that a pair of nodes becomes one node. In general, two nodes, $B_i = \{b_1, b_2, ..., b_m\}$ and $B_j = \{b_1', b_2', ..., b_k'\}$, can be merged into a single node, if b_1' is the only south friend of b_m, and b_m is the only north friend of b_1'. When merging occurs, the two nodes are replaced by the single node $B_i = \{b_1, b_2, ..., b_m, b_1', b_2', ..., b_k'\}$ while B_j becomes an empty node.

Let $b_1, b_2, ..., b_n$ be the n segments in a LAG. We create n nodes $B_1, B_2, ..., B_n$, where $B_i = \{b_i\}$, for $i = 1, 2, ..., n$. The procedure to condense a LAG is then the following:

(1) Initialize variable i to 1, that is, $i \leftarrow 1$.
(2) If $i \geq n$, then delete all empty nodes and stop.
(3) $j \leftarrow i + 1$
(4) If $j = n$, then set $i \leftarrow i + 1$, and go to step (2).
(5) If B_i and B_j can be merged as explained above, then do so.
(6) $j \leftarrow j + 1$
(7) Go to step (4).

For ease in notation, after condensation, the remaining nodes are renamed as $C_1, C_2, C_3, ...$ Thus, the set of C's denotes the *condensed LAG*. Please note that, our condensation procedure includes all the nodes of the initial LAG, while the procedure of [5] skips those nodes with more than one neighbour either above or below. As examples, the top two rows of Fig. 3 show two character patterns, and their corresponding C nodes (middle column).

3 Splitting the Nodes

The nodes obtained after condensation are next split at segments where there is either a sharp change in stroke width, or a sharp change in stroke direction. No node is ever split if it has fewer than three segments. So, after the split, each node has at least one segment. Let W be the width of the characteristic rectangle enclosing the pattern, and *len(b)* the length of segment b.

3.1 Splitting Nodes on Stroke Width Change

Let us consider a node $C = \{b_1, b_2, ..., b_m\}$ where

$m > 2$, and $b_i = \langle r_i, L_i, R_i, N_i, S_i \rangle$ for $i = 1, 2, ..., m$. We scan the segments of C starting from b_1 onward. The node is split up into two smaller nodes, C' and C'', where $C' = \{b_1, b_2, ..., b_i\}$ and $C'' = \{b_{i+1}, b_{i+2}, ..., b_m\}$ provided the following two conditions hold:

(1)

$$|len(b_{i+1}) - len(b_i)| \geq \max(2, \lceil 0.05W \rceil)$$

(This occurs when a stroke width changes by at least two pixels between two consecutive segments in C, the values *2* and *0.05* having been developed heuristically.)

(2)

$$\left[\left(\frac{len(b_i)}{len(b_{i+1})} \leq \min(0.75, (0.365 + 0.045 \times len(b_{i+1}))\right) \wedge \left(\frac{len(b_i)}{len(b_{i+1})} < \min\left(\frac{len(b_{i-1})}{len(b_i)}, \frac{len(b_{i+1})}{len(b_{i+2})}\right)\right)\right]$$

$$\vee$$

$$\left[\left(\frac{len(b_{i+1})}{len(b_i)} \leq \min(0.75, (0.365 + 0.045 \times len(b_i))\right) \wedge \left(\frac{len(b_{i+1})}{len(b_i)} < \min\left(\frac{len(b_i)}{len(b_{i-1})}, \frac{len(b_{i+2})}{len(b_{i+1})}\right)\right)\right]$$

where, if $N_1 = 1$, then b_0 is defined to be the north friend of b_1; otherwise, $len(b_0)$ is assumed to be equal to $len(b_1)$. Similarly, if $S_m = 1$, then b_{m+1} is defined to be the south friend of b_m; otherwise, $len(b_{m+1})$ is assumed to be equal to $len(b_m)$. To split a node between b_i and b_{i+1}, the relative change in stroke width between b_i and b_{i+1} should exceed a bounded heuristic value proportional to the longer of the two b's, and this relative change should be more than the relative change from b_{i-1} to b_i and from b_{i+1} and b_{i+2}; thus, the node is to be split at the point of greatest change in stroke width, provided this change exceeds a threshold value. The values *0.75*, *0.365* and *0.045* are heuristic. The above condition is substantially different from the one given by Pavlidis [5].

Having obtained nodes C' and C'' as mentioned above, we split C'' further, if it can be so done. All nodes in the condensed LAG are split as much as possible under the above criteria. For ease in notation, the nodes obtained after splitting are renamed as D_1, D_2, D_3, ... The segments in a D node represent a stroke with no sharp changes in its width.

3.2 Splitting Nodes on Stroke Direction Change

The D nodes obtained above are next split on sharp change in direction. Given a node $D = \{b_1, b_2, ..., b_m\}$, the straight line joining the mid-pixel of b_1 with the mid-pixel of b_m is called the *backbone* of the node. For $1 < i < m$, we compute the distance between the mid-pixel of b_i and the backbone. Let the largest of these distances be d, and let it be for segment b_k, where $1 < k < m$. If

$$(d \geq 3) \vee (d > \lceil 0.5 \times len(b_k) \rceil)$$

then node D is split into $D' = \{b_1, b_2, ..., b_k\}$ and $D'' = \{b_{k+1}, b_{k+2}, ..., b_m\}$. Intuitively, we can say that segments along a stroke are split into separate nodes when the mid-pixel of the segments are not collinear within the heuristic threshold of *3*, or the backbone lies outside a segment's limits. This happens when the strokes are curved such as in C, O, Q, etc. We can carry out such splitting of nodes until they cannot be split any further. Nonetheless we have seen that, in practice, a node is rarely split more than once. For ease in notation, the nodes obtained after splitting are renamed as E_1, E_2, E_3, ... The segments in an E node represent a stroke that, has no sharp change in width and direction. As examples, the right-most columns of the two top rows of Fig. 3 show the E nodes.

4 Skeletonization of Nodes

The nodes obtained above are next skeletonized in the sequence E_1, E_2, E_3, ... The procedure to skeletonize a given node E depends, however, on whether it is a *path* or a *non-path* node. A node $E = \{b_1, b_2, ..., b_m\}$ is a path node if, and only if, the number of north friends of b_1 is less than or equal to one, and the number of south friends of b_m is less than or equal to one. Informally, a *path node* is a set of segments along a stroke. A node E is a *non-path* node if, and only if, it is not a path node. We discuss first the skeletonization of path nodes, and then of non-path nodes. In our notation for these procedures, *row(q)* and *col(q)* denote pixel q's row and column respectively.

In all our procedures below, certain black pixels in a given node $E = \{b_1, b_2, ..., b_m\}$ are flagged. The unflagged pixels are the ones to be deleted, first from b_1, then from b_2, and so on. To retain the 8-connectivity of the skeleton, two further tests are done before the actual deletion. Suppose that all pixels except a pixel q are to be deleted in segment b_k. Let L and R be the left and right limits of a friend of b_k. Then:

(1) If $col(q) < L - 1$, then all pixels in b_k from column $col(q) + 1$ to column $L - 1$, both inclusive, are not deleted.

(2) If $R + 1 < col(q)$, then all pixels in b_k from column $R + 1$ to column $col(q) - 1$, both inclusive, are not deleted.

The above tests are performed for all friends of segment b_k.

4.1 Procedure to Skeletonize Path Nodes

The skeletonization procedure of a path node depends on whether it is an *upraised* (lying along a vertical or slanting stroke) or *horizontal*. For instance, in the pattern of letter '*A*' of Fig. 3, nodes E_6 and E_8 are horizontal path nodes, while E_2, E_3, E_5, and E_7 are upraised path nodes. For a given path node $E = \{b_1, b_2, ..., b_m\}$ we proceed as follows:

(1) Locate b_k, where b_k is the longest segment in E.

(2) If $len(b_k) < 0.2W$, where *0.2* is a heuristic value, and W is the width of the characteristic rectangle, then identify E as an upraised node, and go to step 5.

(3) Calculate the values of three variables β_1, β_2 and β_3, as follows:

(3.1) If segment b_1 has no north friend, then $\beta_1 \leftarrow 0$; otherwise

$$\beta_1 \leftarrow \frac{len(north\ friend\ of\ b_1)}{len(b_1)}$$

(β_1 measures how the stroke width changes from the north of the node E to E.)

(3.2) $\beta_2 \leftarrow \frac{m}{len(b_k)}$

(β_2 measures the ratio of height to width for the stroke contained in the node E.)

(3.3) If segment b_m has no south friend, then $\beta_3 \leftarrow 0$; otherwise

$$\beta_3 \leftarrow \frac{len(south\ friend\ of\ b_m)}{len(b_m)}$$

(β_3 measures how the stroke width changes from the south of the node E to E.)

(4) If

$$[(\beta_1 \geq 1) \vee (\beta_2 \geq 1) \vee (\beta_3 \geq 1)] \vee [(\beta_1 > 0) \wedge (\beta_2 > 0.75) \wedge (\beta_3 > 0)]$$

(where 0.75 is a heuristic value), is true, then identify E as an upraised node, and go to step (5); otherwise, identify E as a horizontal node, and go to step (6).

(5) For $1 \leq i \leq m$, flag the pixel in segment b_i that is the closest to the backbone of node E. Go to step (8).

(6) If m is odd, then $t \leftarrow \frac{m + 1}{2}$; otherwise, do the following:

(6.1) $u \leftarrow \frac{m}{2}$, $v \leftarrow u + 1$

(6.2) If $len(b_u) > len(b_v)$, then $t \leftarrow u$; go to step (7).

(6.3) If $len(b_u) < len(b_v)$, then $t \leftarrow v$; go to step (7).

(6.4) If the row r_u of segment b_u is less than $H - r_v$ (where r_v is the row of segment b_v), then $t \leftarrow u$; otherwise $t \leftarrow v$ (in other words, when length of b_u is equal to length of b_v, then we select as b_t, the one closer to the boundary of the image).

(7) In $b_t = <r_t, L_t, R_t, N_t, S_t>$, considered to be the medial axis of the stroke, flag the pixel q that is the closest to the backbone.

(7.1) If $(col(q) - L_t) \geq len(b_t)/3$ then flag pixel L_t and all pixels that lie between L_t and q. (Note that *3* is a heuristic value.)

(7.2) If $(R_t - col(q)) \geq len(b_t)/3$ then flag pixel R_t and all pixels that lie between R_t and q.

(7.3) If $\beta_1 > 0$, then, for $1 \leq i < t$, flag the pixel in b_i that is the closest to the backbone.

(7.4) If $\beta_3 > 0$, then, for $t < i \leq n$, flag the pixel in b_i that is the closest to the backbone.

(8) Change all unflagged pixels in E from black to white taking care that the 8-connectivity is retained as described in the beginning of the section.

It should be apparent that pixels to be retained are flagged in step 5 for an upraised path node, and in step 7 for a horizontal path node. The procedure to skeletonize path nodes, that we have proposed above, is a considerable refinement of the procedure given in [5], from which only step 3 has been borrowed unchanged.

4.2 Types of Non-Path Nodes

To skeletonize a non-path node, we need to first check for the type of the node. There are three types of non-path nodes. Consider a non-path node $E = \{b_1, b_2, ..., b_m\}$, and $b_i = <r_i, L_i, R_i, N_i, S_i>$ for $i = 1, 2, ..., m$. The type of node E is then identified as follows:

(1) If $N_1 > 1$ and $S_m \leq 1$, then E is called a *valley* node (for instance node E_8 in the pattern of letter '*e*' of Fig. 3).

(2) If $N_1 \leq 1$ and $S_m > 1$, then E is called a *crest* node (for instance node E_1 in the patterns of letters '*e*' and '*A*' of Fig. 3).

(3) If $N_1 > 1$ and $S_m > 1$, then E is called a *star* node (for instance node E_4 in the patterns of letters '*e*' and '*A*' of Fig. 3).

It should be apparent that every non-path node is one of the above three types. The name of the non-path nodes have been so chosen so that the reader may roughly visualize what the node looks like pictorially. The procedures to skeletonize the three types of non-path nodes are given below.

4.3 Skeletonizing a Valley Node

Our proposed procedure to skeletonize a valley node $E = \{b_1, b_2, ..., b_m\}$ is the following:

(1) For the leftmost north friend b_f of b_1, do the following:

(1.1) If b_f and b_1 do not have any overlapping columns, then set pixel q to be equal to the pixel in b_1 in column L_1.

(1.2) If b_f and b_1 have overlapping columns, let q be the pixel in b_1 such that q is the mid-pixel in the overlapping columns.

(1.3) $g_1 \leftarrow col(q)$

(1.4) For $i = 2, 3, ..., m$, do the following:

$d_i \leftarrow \max(1, g_{i-1} - L_{i-1})$

$g_i \leftarrow \min(L_i + d_i, \lfloor \frac{L_i + R_i}{2} \rfloor)$

(1.5) $g_m \leftarrow \min(R_m, \max(g_1, g_2, ..., g_m))$

(1.6) Let q' be the pixel in b_m in column g_m. Join q and q' by a straight line called the *left arm* of the node.

(1.7) $L' \leftarrow L_f$ (where L_f is the leftmost pixel of b_f).

(2) For the rightmost north friend b_f of b_1, do the following:

(2.1) If b_f and b_1 do not have any overlapping columns, than set q to be equal to the pixel in b_1 in column R_1.

(2.2) If b_f and b_1 have overlapping columns, let q be the pixel in b_1 such that q is the mid-pixel in the overlapping columns.

(2.3) $h_1 \leftarrow col(q)$

(2.4) For $i = 2, 3, ..., m$, do the following:

$d_i \leftarrow \max(1, R_{i-1} - h_{i-1})$

$h_i \leftarrow \max(R_i - d_i, \lceil \frac{L_i + R_i}{2} \rceil)$

(2.5) $h_m \leftarrow \max(L_m, \min(h_1, h_2, ..., h_m))$

(2.6) Let q' be the pixel in b_m in column h_m. Join q and q' by a straight line called the *right arm* of the node.

(2.7) $R' \leftarrow R_f$ (where R_f is the rightmost pixel of b_f).

(3) If b_m has a south friend (that is, $S_m = 1$), then set $k \leftarrow m$; otherwise, set $k \leftarrow \lceil \frac{m+1}{2} \rceil$.

(4) If $(h_1 - g_1) = (h_m - g_m)$, then go to step (8). (This happens when the left arm is parallel to the right arm.)

(5) Let p be the pixel nearest to the point at which the left and the right arms intersect. (For this, the arms may have to be extended, and p may lie outside the characteristic rectangle.)

(6) Reset g_m and h_m as follows:

$g_m \leftarrow col(p)$, $h_m \leftarrow col(p)$

(7) If $k < m$, then set

$k \leftarrow \min(m, \max(k, row(p)))$.

(8) For $i = 1, 2, ..., k$, flag those two pixels in segment b_i that are the closest to the left and the right arms of node E (since they will be retained).

(9) For every north friend b_f of b_1 that is neither the leftmost nor the rightmost north friend of b_1, do the following:

(9.1) Let q be the mid-pixel in the overlapping columns of b_f and b_1.

(9.2) Let q' be the pixel in b_m in the column that is the closest to $\lceil \frac{g_m + h_m}{2} \rceil$.

(9.3) Join q to q' by a straight line called the *ligament*.

(9.4) For $i = 1, 2, ..., k - 1$, flag the pixel in segment b_i that is the closest to the ligament.

(10) If $(S_m = 0) \wedge (R'-L'+1 < len(b_m))$ then flag all pixels in b_k; otherwise, flag the pixels in b_k that lie between the left and the right arms of the node E (pixels on the arms were already flagged in step 8).

(11) Change all unflagged pixels in the node E from black to white taking care that the 8-connectivity is retained as described in the beginning of the section.

4.4 Skeletonizing a Crest Node

As defined above, a valley node $E = \{b_1, b_2, ..., b_m\}$ has $N_1 > 1$ and $S_m \leq 1$. A crest node is its inverse since it has $N_1 \leq 1$ and $S_m > 1$. Space restriction in this paper does not permit us to present the pseudo code of the procedure to skeletonize a crest node. It, however, can be developed easily after making minor appropriate changes in the procedure to

skeletonize valley nodes.

4.5 Skeletonizing a Star Node

A star node can be viewed as a valley node above a crest node. Hence to skeletonize a star node $E = \{b_1, b_2, ..., b_m\}$, proceed as follows:

(1) $k \leftarrow \lceil \frac{m+1}{2} \rceil$

(2) Skeletonize node $E' = \{b_1, b_2, ..., b_k\}$ as a valley node.

(3) If $k < m$, then skeletonize node $E'' = \{b_k, b_{k+1}, ..., b_m\}$ as a crest node.

When $m < 3$, then $k = m$, and we need not to skeletonize E'' since it was already skeletonized in step (2).

As our experiments in the next section show, the above procedures produce good quality skeletons. In rare occasions, however, parts of the skeleton could be non unitary. To ensure that the skeleton is one pixel thick, it can be matched against the 16 templates of Fig. 2. In any matching template, the central pixel is deleted.

5 Experiments and Discussion

As can be seen from above, skeletonizing a character pattern requires quite a few procedures. To recapitulate, the sequence in which the procedures should be applied to a given pattern is the following:

(1) Construct the pattern's condensed LAG (Section 2).

(2) Split the nodes in the condensed LAG to obtain nodes E_1, E_2, E_3, ... (Section 3).

(3) Proceed sequentially in the nodes E_1, E_2, E_3, ... to skeletonize each node, the procedure to skeletonize a given node depending on whether it is a path or a non-path node (Section 4).

We carried out the above steps on a set of over 710 binarily digitized unconstrained handwritten characters (letters A to Z, a to z, and numerals 0 to 9) of various sizes, written by different students at Concordia University, Montreal. The procedures were coded in Borland C++ and implemented on an IBM PC compatible 80486/50 running under the MS-DOS 6.2 operating system. The average cpu-time per pattern, rounded off to the nearest tenth of a millisecond, is the following:

a) Building the condensed LAG (step 1): 8.0 ms

b) Skeletonizing the pattern (steps 2 and 3): 2.3 ms

Above, we have not included the time for input and output, that is the time to read the pattern and print the skeleton. Moreover, we have considered that all patterns were noise free. We had, however, used LAGs to remove noise from the patterns according to the procedure we described in [2].

One should note that in our skeletonization, a pattern is decomposed into nodes. The skeletonization procedure of a given node is then specialized to the configuration at the node: the two types (horizontal and upraised) of path nodes, and the three types (valley, crest and star) of non-path nodes. Intuitively, it is this specialization of the skeletonization procedure that gives us skeletons of good quality. Using specific skeletonization procedures for each node in the pattern increases the flexibility of the skeletonization algorithm and allows it to better adjust to complex binary patterns.

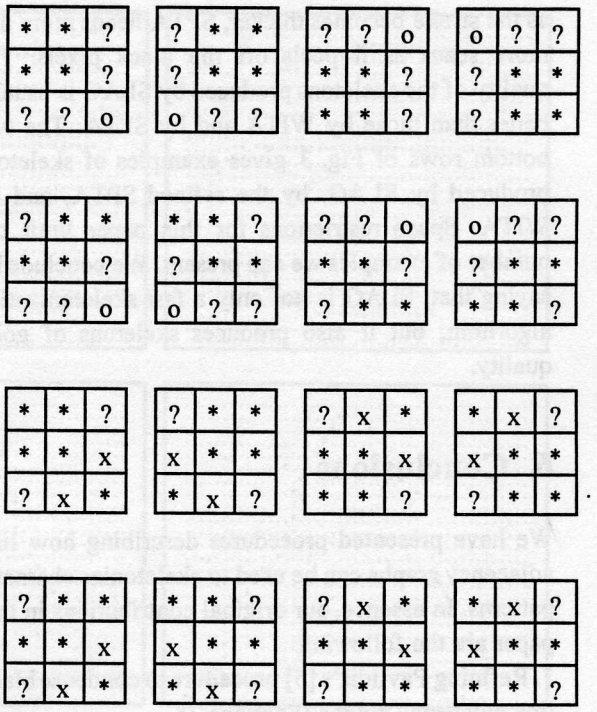

Fig. 2. The templates used to ensure that the skeleton is unitary. If any part of the skeleton matches one of the above templates, the central pixel is deleted. Pixels designated by '*' and 'o' are black and white, respectively. At least one of the pixels marked by 'x' is black, while the type of pixels marked by '?' is irrelevant.

For the purpose of further discussion we shall refer to our set of skeletonization procedures as *SLAG (Skeletonization by Line Adjacency Graph)*. We wanted to compare the performance of SLAG with two other well-known skeletonization algorithms. The first algorithm we chose for comparison uses a peeling approach. It was first proposed as the Safe Point Thinning Algorithm (SPTA) in [4], and it was later refined by speeding up in [6]. To our knowledge, it is among the fastest skeletonization algorithms, perhaps the fastest. The second reference algorithm is the Width-Independent Thinning Algorithm (WITA) [1], which employs a shelling approach.

We found that, on average, SLAG is about three times faster than WITA. If the width of the stroke is less than four, then SLAG is as fast as the refined SPTA of [6]. If, however, the stroke width is four pixels or more, then SLAG becomes faster than SPTA. In fact, the thicker the stroke becomes, the faster is SLAG compared with SPTA. This is because, as the stroke becomes thicker, SPTA needs more and more scans as it peels off the black pixels. The quality of the skeletons produced by SLAG is usually better than those by WITA and by SPTA. The two bottom rows of Fig. 3 gives examples of skeletons produced by SLAG, by the refined SPTA, and by WITA. Space restrictions for this paper limit the number of examples we can present. We conclude by saying that, SLAG is not only a fast skeletonization algorithm, but it also produces skeletons of good quality.

6 Conclusions

We have presented procedures describing how line adjacency graphs can be used to skeletonize character patterns. In essence, our original contributions in this paper are the following:

1. Refining Pavlidis's [5] procedure to condense black line adjacency graphs (Sections 2).
2. Proposing procedures to skeletonize each type of path or non-path nodes (Section 4).

Experiments (Section 5) reveal our procedures to be fast and effective.

References

[1] C. Arcelli and G.S. Di Baja, "A Width-Independent Fast Thinning Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 4, pp. 463-474, July 1985.

[2] S. Iliescu and R. Shinghal, "Noise Removal from Binary Patterns by Using Line Adjacency Graphs", *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pp. 79-84, San Antonio, Texas, October 1994.

[3] B.K. Jang and R.T. Chin, "One-Pass Parallel Thinning: Analysis, Properties, and Quantitative Evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 11, pp. 1129-1140, November 1992.

[4] N.J. Naccache and R. Shinghal, "SPTA: A Proposed Algorithm for Thinning Binary Patterns," *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-14, no. 3, pp. 409-418, May/June 1984.

[5] T. Pavlidis, "A Vectorizer and Feature Extractor for Document Recognition," *Computer Vision, Graphics and Image Processing*, vol. 35, pp. 111-127, 1986.

[6] P.K. Saha, B. Chanda, and D. D. Majumder, "A Single Scan Boundary Removal Thinning Algorithm for 2-D Binary Object," *Pattern Recognition Letters*, vol. 14, no. 3, pp. 173-179, March 1993.

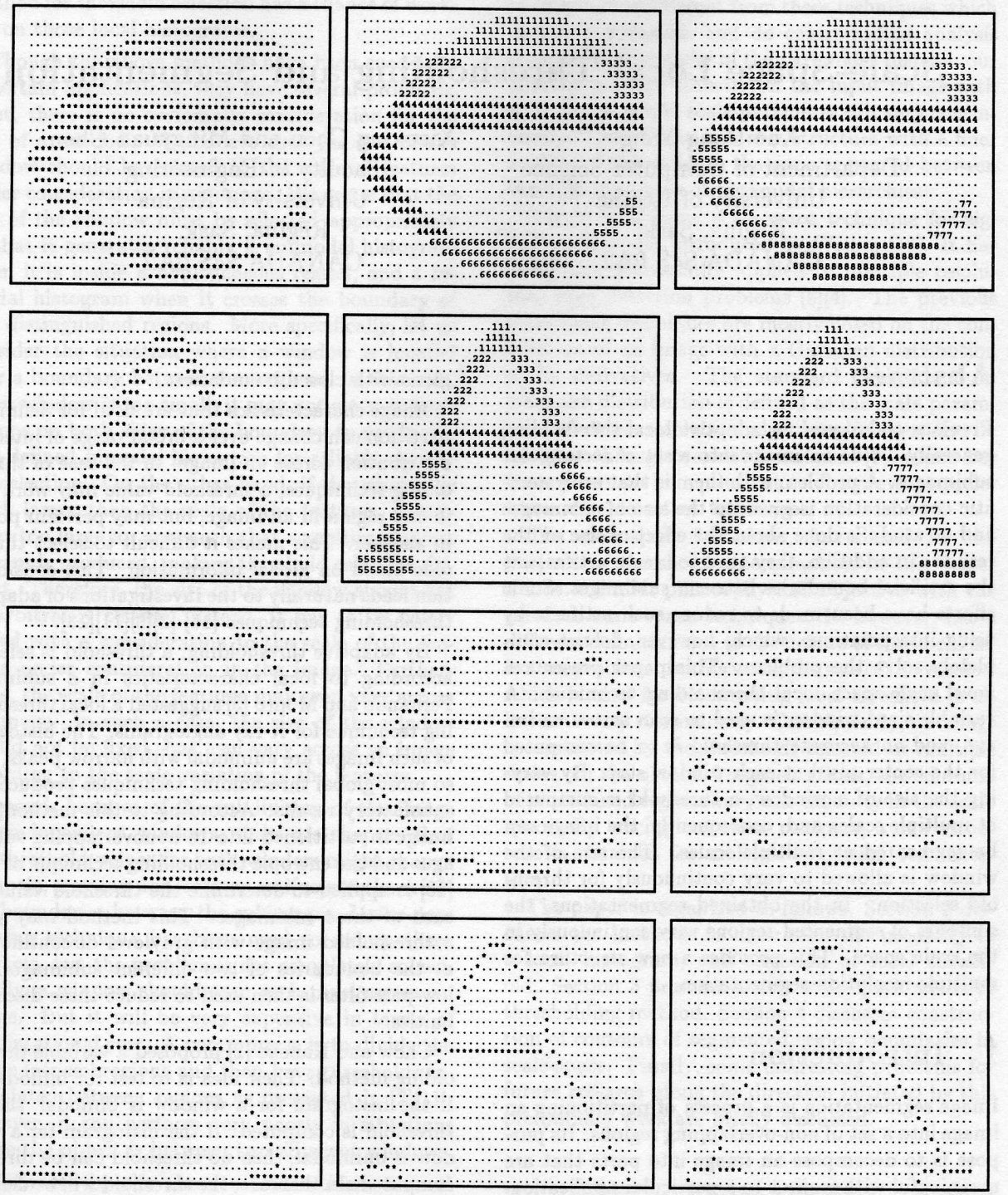

Fig. 3. The top two rows show two character patterns, and their corresponding *C* and *E* nodes. In the middle column, pixels shown by 1, 2, 3, ... belong to the corresponding pattern's nodes *C1*, *C2*, *C3*, ... In the rightmost column, pixels shown by 1, 2, 3, ... belong to the corresponding pattern's nodes *E1*, *E2*, *E3*, ... The two bottom rows show a comparison between the skeletons produced by our algorithm (leftmost column), the skeletons produced by the algorithm of [6] (middle column), and those produced by the algorithm of [1] (rightmost column).