COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING **39**, 186–201 (1987)

A Thinning Algorithm Based on Contours

M. PILAR MARTÍNEZ-PÉREZ*

Facultad de Ciencias Físicas, Universidad Complutense, 28040 Madrid, Spain

AND

JAVIER JIMÉNEZ AND JOSÉ L. NAVALÓN

IBM Scientific Centre, Paseo de la Castellana 4, 28046, Madrid, Spain

Received December 16, 1985

We present a thinning algorithm based on the manipulation of the polygons that represent the borders of thin structures in digital images. We pay special attention to the improvement of the algorithmic complexity of the method. We identify two steps of the algorithm whose "naive" implementations have a quadratic complexity, and that dominate the rest of the problem, and propose faster solutions for both of them. The theoretical estimates for the complexities are supported by simulation experiments. © 1987 Academic Press, Inc.

INTRODUCTION

Classical thinning algorithms work on binary images coded in raster form [1–5]. They scan the image, using a mask of a given size (usually 3 × 3), and proceed to erase all those pixels that belong to an edge, but not to the axis of elongated objects. The process is repeated until only axial points are left. For an image whose linear dimension is **n**, measured in pixels, these methods involve an **O(n**2)** search over all the pixels in the image, followed by **m*mask size** thinning comparisons, where **m** is the length of the axis. The whole procedure has to be repeated a number of times proportional to the maximum thickness of the object. Because both the linear size of the image and the maximum thickness of the objects, in pixels, increase with the resolution of the digitization, the effective complexity of the whole procedure is **O(n**3)**, and quickly becomes impractical for high quality images. Moreover, the result is again a raster image which, normally, has to be subject to a further line following step to reduce the thinned features to vector form [5, 8]. This extra step requires a new **O(n**2)** search, plus a set of comparisons which scales as **m*mask size**.

Recently, a new type of thinning algorithm has been proposed which uses the vectorized borders of the objects, coded as polygons, to calculate an approximate "median" line [7–10]. Initially, these algorithms find a pair of "opposite" points and, beginning at this first couple, they generate sequentially a list of pairs and projecting lines whose "median points" are added to the median line. In this process, each vertex of the border is checked only once and, therefore, the number of operations scales with **m**, which now is the number of sides of the polygon, and which increases only linearly with **n**. Since the initial contour following step is equivalent to the final line following of the classical algorithms, these new methods promise a reduction of complexity from **O(n**3)** to **O(n**2)**, with the thinning now reduced to a trivial linear postprocessing step to vectorization.

*Present address: IBM Scientific Centre, Madrid, Spain.

0734-189X/87 $3.00
Copyright © 1987 by Academic Press, Inc.
All rights of reproduction in any form reserved.

In practice, the computing times obtained are not as good as expected. To begin with, the choice of the first pair of points requires checking all the vertices of the object in order to choose, for a given vertex, its "best" matching point. Furthermore, the completion of a branch produces a loss of sequentiality and, when this happens, a new initial pair has to be detected. Thus, the number of operations needed for this initial point search is proportional to **m*number of branches** which, unfortunately, is usually **O(n**2)**.

Once the whole polygon has been examined, the algorithm produces a set of branches, separated (or not) by physical gaps. The final stage of the algorithm is dedicated to building a continuous median line without gaps, and to reconstruct the connections among different branches. This requires an exhaustive search of all the possible connections, and may need, in principle, up to **number of branches**2** operations, which is again **O(n**2)**.

These two problems decrease the theoretical advantage of this new class of thinning algorithms. We propose in this paper two possible ways of determining an initial point pair for a branch whose complexities scale with **m** and **number of branches**2**. We also show that, for simply connected objects, a proper ordering of the search operations establishes the final tree-like structure in linear time.

GENERATION OF THE MEDIAN LINE

Algorithm I gives a general idea of the processing of the median line for a single branch. The calculation involves two pointers to a "current" vertex, and to its "facing" segment, which move incrementally along the polygon in the course of the algorithm. Roughly speaking, we begin at an arbitrary vertex, and first identify its facing segment (**procedure** OPPOSITE). Then, we throw a "projecting" line from the vertex to the segment (Fig. 1) and check whether it intersects it, satisfying a set of conditions which are detailed below. If it does, the mid-point between the vertex and the intersection is added to the median line, and the vertex pointer is moved forward to the next vertex in the polygon. The whole procedure is repeated with the new point and the old segment and, if it is not successful, it is tried again after incrementing the segment pointer. The algorithm marches through the polygon by incrementing both pointers as needed and stops either when all the points in the

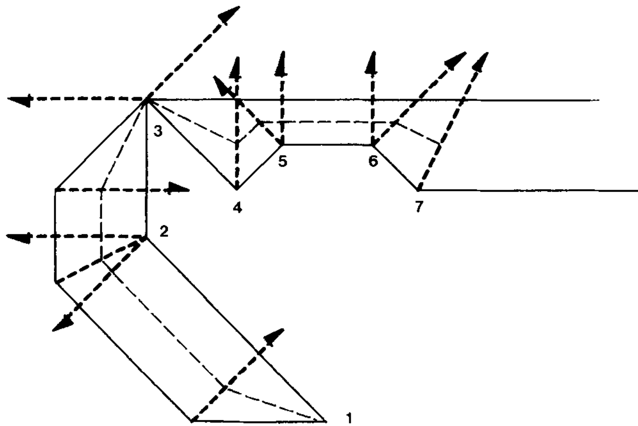

FIG. 1. Illustration of the calculation of the points of a branch.

polygon have been processed, or when sequentiality is unable to provide a valid intersection.

```
(**********************************************************)
procedure BRANCH;
(**********************************************************)
begin
      OPPOSITE;
      repeat
          INTERSECTION;
          next vertex
      until ('branch completed')|('no more vertexes')
                                                  Algorithm I
(**********************************************************)
```

Algorithm II shows the details of how intersections are computed. The directions of the projecting lines depend on whether the inner angle formed by the polygon at the vertex is smaller or greater than 180°. In the first case, the projecting line is the bisector of the angle (see vertex 4 in Fig. 1), while in the latter (vertex 5 in Fig. 1), we project two lines from each vertex, which correspond to the inner normals to the two segments meeting at it. We have found this latter procedure useful in smoothing sharp concave corners.

```
(**********************************************************)
procedure INTERSECTION;
(**********************************************************)
begin
      if ('angle < 180') then draw bisector
      else draw inner normal to incoming segment;
      if ('valid intersection') then
      begin
          add midpoint to axis;
          if ('angle > 180') then
          begin
              draw inner normal to outgoing segment;
              if ('valid intersection') then
                      add midpoint to axis
              else
              begin
                  FAN;
                if ('fan backtracks') then
                        'branch completed'
                else next segment
              end
          end;
      end
      else
      begin
          NORMAL;
          if NOT ('valid intersection') then
                  'branch completed'
          else next segment
      end;
      next vertex
end;
                                                  Algorithm II
(**********************************************************)
```

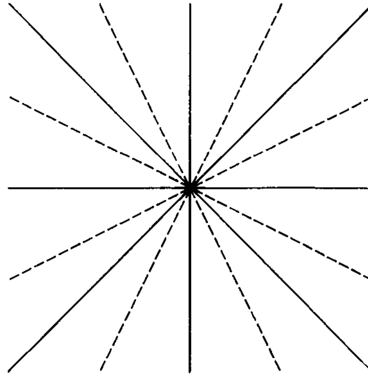

FIG. 2. From eight initial discrete directions (continuous lines) we obtain a set of 16 possible directions for the projecting lines (continuous and broken lines).

It is generally possible to code the original contour in such a way as to speed considerably the calculation of the intersections. Assume that the polygons derive directly from a contour in a digital image and that no approximation is involved. The directions of the sides can take only one out of eight possible values (Fig. 2), and the directions of the projecting lines, one out of 16, counting the possible bisectors. The directional parameters needed to compute the intersections can then be pre-stored as entries in an 8×8 table whose indices are the directions of the two segments meeting at the vertex. It is easy to arrange for the contour follower to provide those indices directly [8].

The thinning algorithm discussed here is intended for "elongated" objects, with a clearly defined longitudinal direction, and it usually makes sense to define an average thickness that can be considered reasonably constant along the object. An estimate for this thickness is

thick = 1 + (2*surface / perimeter),

where the 1 corrects for the finite size of the pixel. The idea is that an intersection is only considered valid if, in addition to everything else, the distance between the root vertex and the intersection with the facing segment is less than **THR*thick**, where **THR** is an adjustable parameter, **O(1)**, which allows for local irregularities in the thickness of the object. We have found that the result of the thinning is not very sensitive to the choice of this parameter, although small branches are smoothed out, but that the amount of computer time used by the algorithm increases considerably if the value chosen is too large. In all our examples, we have used values of **THR** between 2 and 3.

The presence of an "average" width and of a "slack" factor is a characteristic of most thinning algorithms which do not try to make a strict calculation of the "skeleton" [9] of the object. Skeletons are well defined mathematical structures that are unique to each object, whether elongated or not, and free of the choice of arbitrary parameters. Unfortunately, they are also hard to compute, include curved segments, and tend to introduce unwanted branches that have little to do with the structure of the external boundary of the object, but which are imposed by the requirement that they described its shape exactly [10].

Thinning transforms, on the other hand, are approximations that make sense only if used to describe median lines of objects which are already "thin." As a consequence, the result is not uniquely defined from the object, but errors are not very important since they are usually of the order of the original thickness, but small with respect to the dominant "length." As a consequence, big savings in computer time are possible, but a thickness has to be defined a priori, in essence to set the appropriate magnitude for the acceptable errors [6–8].

We are now ready to describe in detail the thinning algorithm. Assume that the current vertex corresponds to an angle smaller than 180°, and that the last vertex gave a valid intersection with the current segment. The inner bisector is drawn and intersected with the current segment. If the intersection exists and the distance from it to the vertex is smaller than the threshold distance, it is considered valid and its midpoint is added to the median line (see the exception below). Otherwise, the algorithm increments the segment pointer and checks whether the new segment would give a valid intersection (call to **procedure** NORMAL in Algorithm II, and Fig. 3a). Before actually testing the intersection, however, the algorithm tries to add a point to the median line representing the end of the segment that has just been abandoned. To do this, it projects the **first** vertex in the new segment, perpendicularly to the **incoming** segment to the current vertex. If this intersection is valid in the same sense as above and, moreover, if its projection over the incoming segments falls "forward" (in the sense of advance around the polygon) of the previous vertex and of any previous projection, its mid-point is added to the middle line, and the

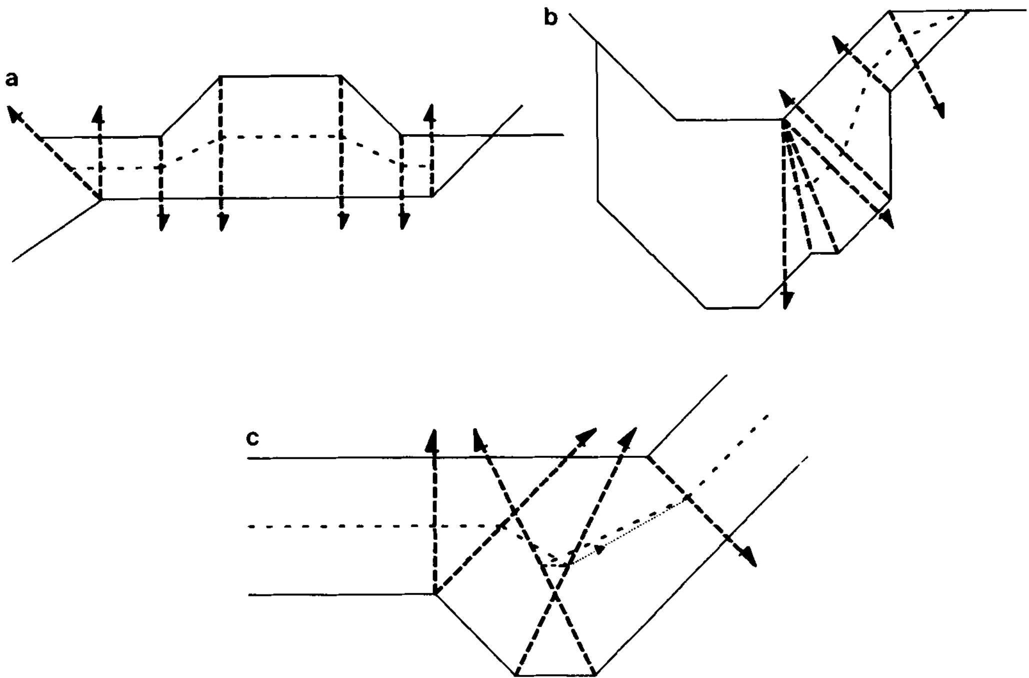

FIG. 3. Treatment of "non-fatal" invalid intersections: (a) **procedure** NORMAL in Algorithm II. (b) **procedure** FAN in Algorithm II. (c) Avoiding spurious loops in thick corners.

current vertex is tried against the new segment. The process is repeated until either a segment is reached that is intersected correctly by the projecting line from the current vertex, or until one of the "back projections" becomes invalid for any of the two reasons given above. In the first case, the mid-point of the intersection is added to the median line, the vertex pointer is incremented and the process is repeated. In the second, the branch is closed, all the points and segments used for it are deleted from the polygon, and the whole algorithm is re-started for the remaining points by searching for a new initial pair.

The treatment of the first normal to a vertex with an angle greater than 180° is similar to the one above but, in this case, the vertex pointer is not incremented at the end, and the intersection is repeated with the second normal. The procedure followed when this second intersection fails is very similar to the one used above for the failure of the bisector. The segment pointer is increased sequentially, and the vertices defined by the ends of the abandoned segments are joined to the current vertex, forming a "fan" that is required to never "backtrack" and to remain always shorter than the threshold distance (**procedure** FAN in Algorithm II, and Fig. 3b). The fan ends successfully when a segment is found that intersects correctly the second normal to the current vertex, or the branch is closed if a failure is detected.

We have found this strategy fairly easy to implement and are able to detect all the special cases in which loss of sequentiality is due to branching from the current median line. Each vertex is treated only once, but each vertex generates at least one point in the median line, so that little detail is lost in the thinning process. The only exception, referred to above, comes from the consideration of sharp corners in fairly thick objects. In those cases, the median line defined above can backtrack, giving rise to small loops which, even if strictly consistent with the approximation, look "wrong" in the final result. To avoid this effect, each valid intersection is checked with the position of the previous one along the current segment. If the position of the new intersection is not "forward" of the previous one, but is otherwise valid, it is still treated as valid, but it is not added to the median line (Fig. 3c). Other useful thinning strategies are presented in [6–8].

IDENTIFICATION OF THE INITIAL PAIR

As we have seen, the contour follower provides an ordered list of the vertices that define the borders of the objects in a binary image. We thus know originally the sequential relation between the vertices, which is used to march the pointers in the thinning algorithm, and is the basis for its linear complexity. Unfortunately, we do not know which segments are opposite to a given vertex, and this information is needed, at least for one point, to start the thinning procedure. In fact, this search of a starting pair has to be done again each time that sequentiality is lost and a new branch has to be started. The analysis of this operation is the purpose of this section.

Algorithm III describes a first intuitive approach to the search for matching segments. Essentially, we choose a vertex, draw its first projecting line, and test it against all the segments in the polygon. If some valid intersections are found, the segment whose intersection is closest to the current vertex is kept as the ideal pair. Otherwise, the same procedure is repeated with the next vertex. The number of intersections to be tested in this step is $\mathbf{O(n)}$ and, since it has to be repeated for

every new branch, and the number of branches is also **O(n)**, the total time spent in this step is **O(n**2)**, and dominates the complexity of the whole algorithm.

```
(************************************************************)
procedure OPPOSITE1;
(************************************************************)
begin
      first vertex and segment;
      repeat
      WINDOW;
   FOR1: for first segment to last do
      begin
         if ('segment touches window')
         then
         begin
            INTERSEC;
            if ('valid intersection') then
            add segment to list;
            next segment
         end
      end;
      next vertex
      until ('non empty list');
      select nearest segment from list
end;
                                        Algorithm III
(************************************************************)
```

A first remedy is to speed the test of each segment. Actual intersection of two segments is a relatively long process involving several floating point operations. On the other hand, many of the segments in the polygon are usually too far away from the vertex to even be considered as candidates. An easy way to "weed" out these segments without doing a full intersection is to bound each of them inside a rectangular window, with sides parallel to the coordinate axes (Fig. 4). The projecting line, bounded by the thresholding distance, also has an equivalent window, and a segment will not intersect the line if their windows do not touch. Two windows can be tested with just four comparisons, which are usually much faster than a full intersection, and only those segments passing this test, usually a small

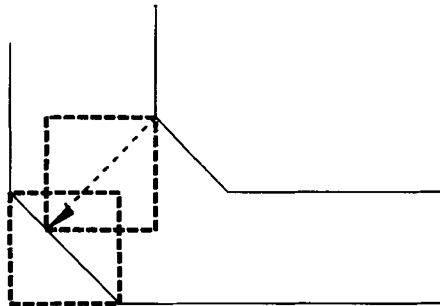

FIG. 4. Bounding "windows" are used to speed the search for opposite pairs.

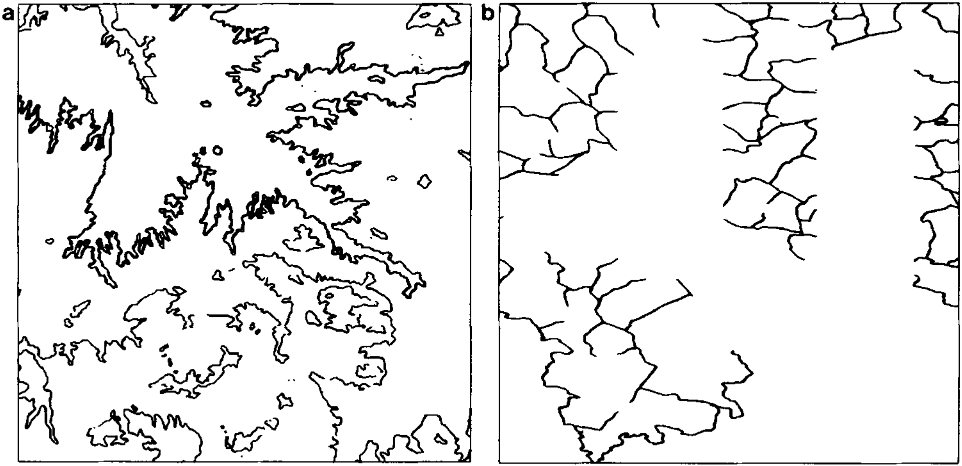

FIG. 5. Test images used to illustrate algorithmic complexity. Part (a) is a set of geographic contour lines, while (b) derives from land ownerships borders.

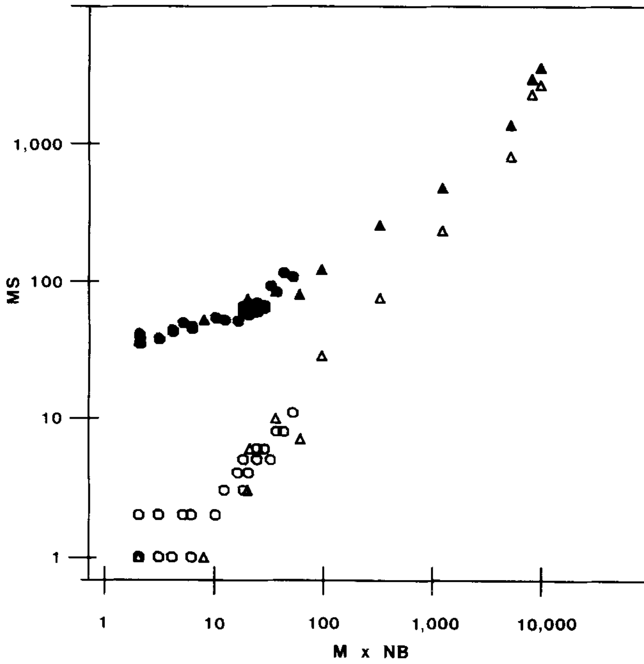

FIG. 6. Performance of "naive" search for opposite pairs. CPU time vs. **number of points*number of branches**. Open symbols are associated to the initial search, closed symbols to the whole thinning process.

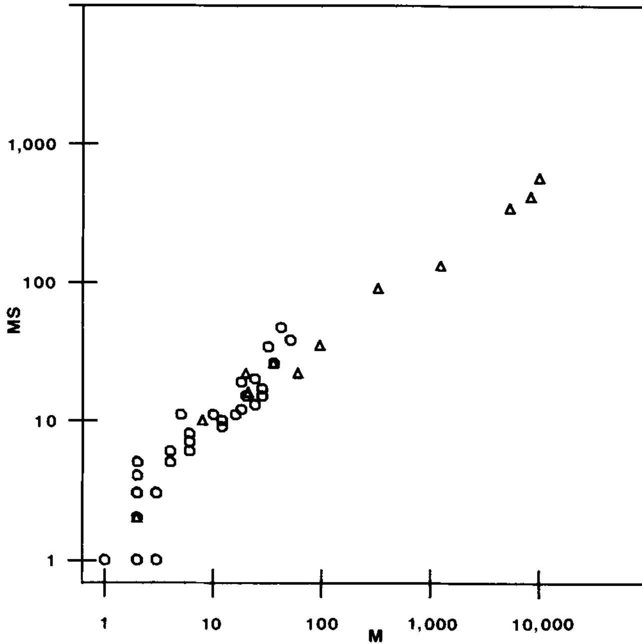

FIG. 7. Performance of thinning algorithm in the calculation of isolated branches. CPU time vs. number of points.

constant number, need be considered for full intersections. The final operation count is the **O(m*number of branches)** comparisons, plus **O(number of branches)** intersections, but the complexity of the first part is still **O(n**2)**.

We have tested Algorithm III with several different images, two of which are shown in Fig. 5. They are chosen to represent different types of geometries. The polygons obtained from the first image present quite complicated geometric structures (Fig. 5a), but relatively few branches, while those of the second one have a simpler geometry but many branches. The complete algorithm was implemented in PL/I and run in an IBM 4381-2 under CMS. In Fig. 6 we have plotted the CPU time used both by the initial segment searches (Algorithm III) and by the complete thinning algorithm, as a function of the **m*number of branches**. Figure 7 shows the time used by the isolated thinning step, as a function of the number of vertices, **m**. It is clear, both that the thinning step is linear in **m**, and that the complexity of the complete algorithm is very soon dominated by the initial search, suggesting that any effort to improve the algorithm must concentrate on that latter part.

IMPROVEMENTS

Most of the time used by the initial search algorithm is spent scanning all the segments of the polygon to check whether an intersection should be attempted. It is easy to see that an exhaustive search is indeed needed, since a polygon can have loops or folds, so that a vertex may face several segments, and it is not enough to

keep as the facing segment the first one that apparently satisfies all the conditions. The obvious way to speed the search is to use all the information available to reject candidates faster.

Algorithm IV shows a first approach to doing this. The idea is to group segments into contiguous substrings which are monotonic in one of the coordinates. Each string has a bounding "strip" whose extent, along the monotonic coordinate, is defined by its initial and its final point, and that can be used to accept or reject the string as a whole. Moreover, the search for candidate segments within a candidate string can be done with a fast binary search along the monotonic coordinate. This procedure is well known for many problems in the manipulation of convex polygons [11], in which the partition in monotonic strings is natural, and in which the number of monotonic substrings is constant (and always equal to 2), leading to search times which are **O(log m)**. For general polygons, on the other hand, the number of substrings is, in general, variable and, at worst, can be **O(m)**. In those cases, a general search would involve first a search over strings, which would have to be sequential and take **O(number of strings)** operations, and a search within the chosen string, which is **O(log[m / number of strings])**, and is usually trivial by comparison. In general we can assume that **number of strings** is **O(number of branches)**, so that the complexity of this search strategy is **number of branches**2**, which is generally lower than for Algorithm III, but still **O(n**2)**. Figure 8 represents the performance

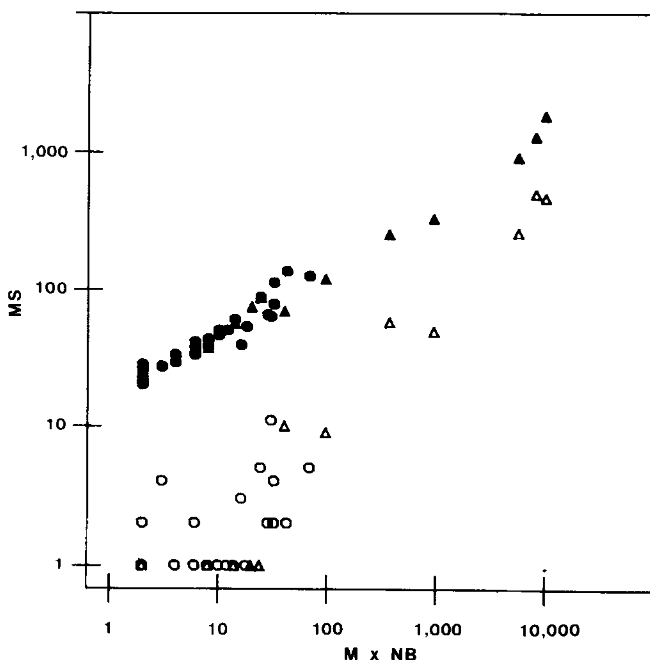

FIG. 8. Performance of Algorithm IV. CPU time vs. **number of points*number of branches**. Symbols as in Fig. 6.

of this new algorithm. The absolute times are faster than in Fig. 6, but the slope of the performance curve is still quadratic in the number of points.

```
(************************************************************)
procedure OPPOSITE2;
(************************************************************)
begin
      select coordinate;
      NEWSTRING;
      for first vertex to last do
      if monotony changes then NEWSTRING
      else add vertex to current substring;
      first vertex;
      repeat
      WINDOW;
   FOR3: for all substrings that touch window do
      begin
         select segments through binary search;
         for selected segments do
         begin
            INTERSEC;
            if ('valid intersection') then
            add segment to list
         end
      end;
      next vertex
      until ('non empty list');
      select nearest segment from list
end;
                                          Algorithm IV
(************************************************************)
```

A more drastic departure from the original strategy for the search of opposite pairs is the "raster" method in Algorithm V. For any object, the first step is to cover it with an imaginary grid (Fig. 9), formed by squares whose sides are proportional to the threshold distance, and to associate to each element of the grid a list with all the segments which touch it. We have coded the grid as a matrix that contains pointers to linked lists. Each element of this matrix represents a square of the grid and its associated linked list contains pointers to all the segments which touch it anywhere.

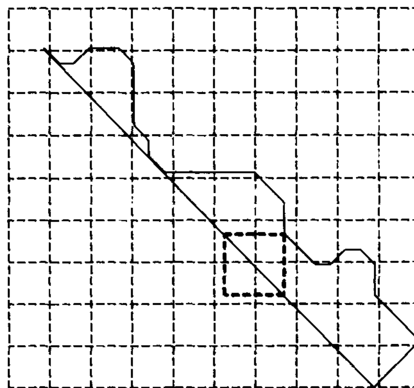

FIG. 9. Raster algorithm. Illustration of imaginary grid and search window.

The search follows a series of steps similar to those of Algorithm III. But in FOR2 we now consider only those segments touching the squares that touch the window associated to the projecting line being considered.

```
(**********************************************************)
procedure OPPOSITE3;
(**********************************************************)
begin
      form grid;
      for first vertex to last do
      associate vertex to element of the grid;
      first vertex;
      repeat
      WINDOW;
   FOR2: for all segments that touch window do
      begin
         INTERSEC;
         if ('valid intersection') then
         add segment to list
      end;
      next vertex
      until ('non empty list');
      select nearest segment from list
end;
                              Algorithm V
(**********************************************************)
```

Obviously, the formation of the grid requires a scan of all the vertices of the structure, but it has to be done only once for each object. The list of segments to be

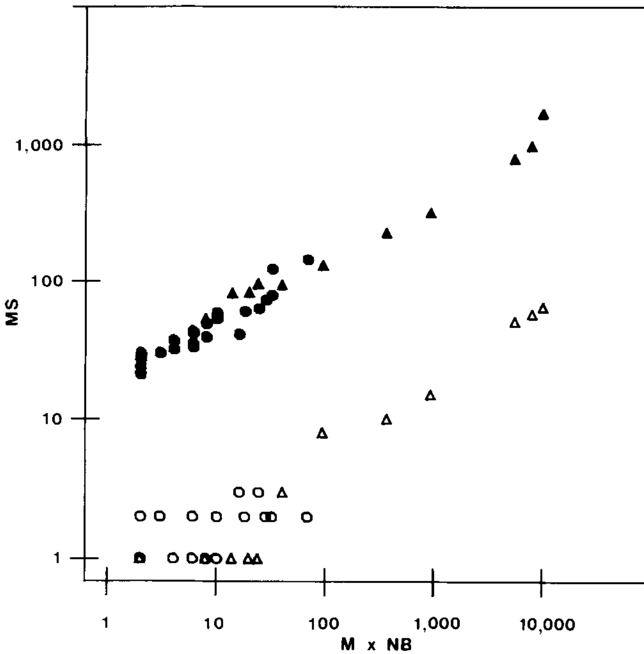

FIG. 10. Performance of the raster algorithm. Symbols as in Fig. 6.

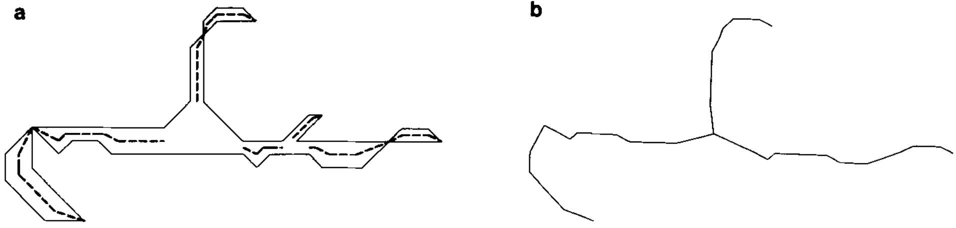

FIG. 11. (a) Set of branches generated by the thinning algorithm. (b) Final median line obtained after cross-linking and discarding of short branches.

searched for each branch is only that associated to the few adjacent grid cells, and is only **O(1)**. As a consequence, the operation is only **O(m + number of branches)**, which is significantly smaller than for the previous algorithms and increases only as **O(n)**. On the other hand, the storage needed for the pointers matrix is proportional to **O(n**2)**. The performance of this algorithm is presented in Fig. 10. Not only are the times shorter than any of the previous algorithms, but its asymptotic complexity is clearly lower.

POSTPROCESSING

The calculation of the median line of an object is done by means of the consecutive extractions of branches. Each branch is generated from a different subset of vertices, so that each vertex is examined only once. When all vertices have been treated, we are left with a set of branches, separated or not by physical gaps (Fig. 11a). We can represent mathematically this structure by a graph, whose arcs are the branches and whose nodes are defined either as regions of confluence of different branches or as terminal end points.

The purpose of postprocessing is to obtain this graph, which implies finding to which nodes do the ends of a given branch belong. This is difficult in general if the contiguity relations between branch ends are lost in the course of the algorithm. For each branch end, we know which vertex and which segment form the terminal pair on which the branch was closed (or opened). Two branch ends are contiguous if

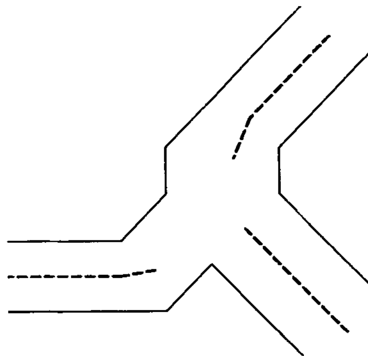

FIG. 12. Two branch ends are contiguous if they have no intervening branch separating them along at least one of their two sides.

FIG. 13. Computing order used by the recursive thinning algorithm.

either their terminal vertices or their terminal segments are contiguous, in the sense that there are no points between them that belong to a different branch (Fig. 12). Branch ends which are contiguous belong to the same node.

If no information on contiguity is available from the output of the thinner, the general graph building algorithm must check each branch end against all others to find whether they are contiguous, and build lists of contiguous ends which would eventually correspond to nodes. This is a **O(number of branches**2)** operation. In the case of simply connected objects, the graph is actually a tree, and contiguity can be established while the branches are being computed by a recursive organization of the algorithm.

The recursive algorithm begins by calculating the first branch, using all the vertices in the polygon (Fig. 13). Once this branch is closed (branch A in Fig. 13), there might still be points left in the polygon which have not been used. The algorithm picks the next unused vertex, finds it opposite segment and computes a new branch (branch B). It also creates a node associated with the end of the old branch and with the beginning of the new one (node 1). Any point still unused in this part of the polygon must be located either near the closing end of the new branch (node 2) or near the gap between the first segment used for branch B and the last segment used for branch A (node 1). The algorithm tries to compute new branches in those two spots, in that order, calling itself recursively and exiting only when it finds no points to construct a branch on a given gap (Algorithm VI). Since the routine always knows on which node it is trying to build a new branch, it can save this information to be used directly in organizing the final graph.

```
(*************************************************************)
procedure GENERAL(first, last);
(*************************************************************)
begin
      BRANCH;
      num_of_branches := num_of_branches + 1;
      associate ends of branch to nodes;
      if ('vertexes left') then
      begin
         firstv := first + 1;
         lastv := last segment of branch;
         GENERAL(firstv, lastv)
      end;
      if ('vertexes left between 'last'
                    and first segment of branch') then
```

```
        begin
            firstv := first segment of branch + 1;
            lastv := last;
            GENERAL(firstv, lastv)
        end
end;
                                        Algorithm VI
(************************************************************)
```

After the routine returns from the last recursive call, there may still be points near the opening end of the first branch. These are treated by exactly the same method, but changing the direction of motion around the polygon. For singly connected objects, this ordering would produce a complete list of the branch ends belonging to the nodes in the tree. In the multiply connected case, some nodes would be incomplete, because re-entrant branches cannot be identified as part of their target nodes, but the algorithm does provide a list of incomplete nodes that can then be cross-linked through a restricted quadratic search.

The result is a complete set of branches, with each branch end associated to a graph node. The final step of postprocessing is to build from this a continuous tree-like structure, and to discard those small branches that are considered irrelevant for the application involved. Algorithm VII shows how this is done.

```
(************************************************************)
procedure POSTPROCESSING1;
(************************************************************)
begin
    for first node to last do
    begin
        associate a single point to the node;
        if degree two node then
        join appropriate ends of branches
        else
        begin
            examine node's branches;
            as long as node doesn't disappear
                    eliminate short leaves;
            if degree two node generated then
            join appropriate ends of branches
        end
    end
end;
                                        Algorithm VII
(************************************************************)
```

The first step is to generate a point to represent each node. We chose for it the center of gravity of all the branch ends that define it. This choice is arbitrary, but consistent with the general degree of approximation used throughout the method. Next, we calculate the length of all the branches. The simplification of the median line requires a new parameter: the minimum length "allowed" for a branch. This parameter is essentially semantic, and depends on the application intended for the final result. Its purpose is to discard small branches that usually originate from irrelevant protrusions in the original object. Since which of those protrusions generate branches is governed by the threshold distance used in the thinning, the new "rejection length" is usually taken as a multiple of that distance.

To begin with, the algorithm eliminates all the nodes with degree two, joining the appropriate ends of their branches. Then, it studies all the other nodes. It discards all terminal branches shorter that the rejection length, but only as long as their disappearance does not produce the elimination of a node. Thus, if a node is formed by three short branches, only the shortest one is discarded; the two remaining ones will be joined in the next iteration of the cleaning loop and their combined length checked to test whether they should still be thrown away. The process is iterated until no change occurs. The result of connecting and simplifying the branches in Fig. 11a is shown in Fig. 11b.

The postprocessing algorithm examines each branch once, so that the number of operations is proportional to **number of branches**.

CONCLUSIONS

We have developed a thinning algorithm that uses as input data the vectorized borders of the objects of an image. Although this approach was already proposed by some authors, there were still problems to solve.

The first problem consisted of choosing a starting pair to initialize the computation of each branch. This problem had not been explicitly treated by the previous investigators, but the obvious solutions are not very efficient and have complexity **O(m**2)**. We propose two improved algorithms whose computing times scale with **m** and **number of branches**2**.

A second problem arises in the reconstruction and simplification of the median line. The thinning algorithm obtains, as an intermediate result, a set of branches, separated by gaps. The obvious way of cross-linking these separate branches into a single structure is to do an exhaustive search of all possible interconnections, which is again a **number of branches**2** operations. We show that a proper ordering of the algorithm establishes the final result in linear time, at least for singly connected objects.

REFERENCES

1. R. Stefanelli and A. Rosenfeld, Some parallel thinning algorithms for digital pictures, *J. Assoc. Comput. Mach.*, **18**, No. 2, 1971, 255–264.
2. A. Rosenfeld and L. S. Davis, A note on thinning. *IEEE Trans. Systems Man Cybernet.*, **6**, no. 3, 1978, 226–228.
3. T. Pavlidis, A thinning algorithm for discrete binary images, *Comput. Graphics Image Process.*, **13**, 1980, 142–157.
4. A. Bel-Lan and L. Montoto, A thinning transform, *Signal Process.*, **3**, 1981, 37–47.
5. C. Arcelli, Pattern thinning by contour tracing, *Comput. Graphics Image Process.*, **17**, 1981, 130–144.
6. B. B. Chaudhuri, A simple method of thinning, *J. Inst. Electron. Telecom. Engrs.* **24**, No. 6, 1978, 264–265.
7. B. Saphiro, J. Pisa, and J. Sklansky, Skeleton generation from x, y boundary sequences, *Comput. Graphics Image Process.*, **15**, 1981, 136–153.
8. J. Jimenez and J. L. Navalon, Some experiments in image vectorisation, *IBM J. Research Develop.*, **26**, No. 6, 1982, 724–734.
9. F. L. Bookstein, The line skeleton, *Comput. Graphics Image Process.*, **11**, 1979, 123–137.
10. D. T. Lee, Medial axis transformation of a planar shape, *IEEE Trans. Pattern Anal. Machine Intell.*, **PAMI-4**, 1982, 363–369.
11. M. I. Shamos, *Computational Geometry*, Ph.D. thesis, Yale University, 1978.