

Breadth-First Search and Its Application to Image Processing Problems

Jaime Silvela and Javier Portillo

Abstract—This paper explores the use of breadth-first graph traversal for the processing of digital images. It presents efficient algorithms for eroding, dilating, skeletonizing, and distance-transforming regions. These algorithms work by traversing regions in a breadth-first manner, using a queue for storage of unprocessed pixels. They use memory efficiently—pixels are removed from the queue as soon as their processing has been completed—and they process only pixels in the region (and their neighbors), rather than requiring a complete scan of the image. The image is still represented as a pixel matrix in memory; the graph is just a convenient framework for thinking about the algorithms.

Index Terms—Morphological processing.

I. INTRODUCTION

A REGION is a group of pixels of the same color (or grey-level), any two of which can be connected by a continuous path of neighbors belonging to the group. The neighbors of a pixel are either those above it, below it, to its left, and to its right (4-neighbors), or the ones mentioned plus those to its upper left, upper right, lower left, and lower right (8-neighbors). The algorithms presented in this paper work for both kinds of neighborhood definition.

Section II presents a simple flooding algorithm to introduce the concepts used throughout the article. It is what Levoy [1] calls a pixel-oriented queue algorithm. Section IV presents algorithms for erosion and dilation of regions that handle subsequent iterations efficiently by using queues. A similar approach was suggested in Vincent [2] in the context of morphological reconstruction. Section V presents a simple algorithm to perform the distance transform of a region with just one pass over the pixels of the region, in contrast with usual methods requiring two or more scans of the image, as discussed in [3]. Section VI presents a fast skeletonization algorithm based on that of Zhang-Suen [4]. Of course, as discussed in [3], the skeletal image can be computed more accurately by working on the distance-transformed image.

All the algorithms presented are based on viewing a raster image not as a pixel matrix, but as a *graph*, whose vertices represent pixels, and whose edges represent neighborhood between pixels, as shown in Fig. 1.

Manuscript received July 25, 2000; revised April 5, 2001. This paper is based on J. Silvela's M.S. thesis, written while he was a full-time student at the Universidad Politécnica de Madrid, Madrid, Spain. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Scott T. Acton.

J. Silvela is with Nortel Networks, 28692 Madrid, Spain (e-mail: jsilvela@nortelnetworks.com).

J. Portillo is with the Polytechnic University of Madrid, 28692 Madrid, Spain (e-mail: javierp@grpss.ssr.upm.es).

Publisher Item Identifier S 1057-7149(01)06041-9.

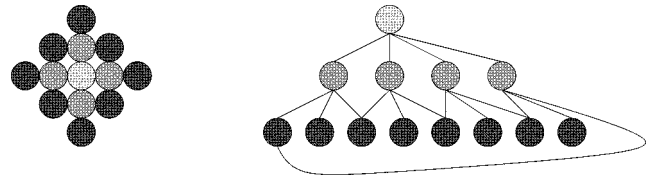


Fig. 1. Raster image seen as a graph.

This suggests that pixels in the image can be traversed in a breadth-first manner, rather than with the traditional raster scan. Because pixels to be processed are stored in a *queue*, breadth-first traversal yields propagation patterns that grow or shrink uniformly across their boundary, like wave fronts. This makes it well suited to problems that—like distance-transformation—require operating on layer after layer of pixels in the region.

We do not, however, need to represent images as graphs; we can still use the pixel matrix representation. All that is required is that, given a pixel in the matrix, we know the locations in memory of all its neighbors. A procedure that finds the i th neighbor of a pixel¹ allows us to implement the algorithms below, irrespective of the underlying representation in memory.

II. FLOOD-FILLING (OR OBJECT LABELLING)

In its simplest form, a flood-filling algorithm is one that changes the color of a region, given an initial pixel in that region. This is similar to the problem called “object labeling” in computer vision, which consists of assigning one same number to all the pixels in a region. The simplest algorithm that solves this problem is

```
FLOOD-FILL-1 (initial-pixel,
initial-color, final-color)
  color(initial-pixel) ← final-color
  for each n ∈ Neighbors (initial-pixel)
    if color(n) = initial-color
      FLOOD-FILL-1 (n,
initial-color, final-color).
```

As is widely known, this highly recursive algorithm uses a great amount of memory, and may cause stack overflow, so other algorithms, based on coloring scan-lines and using a stack, have been developed in Levoy [1], Smith [5], and Fishkin and Barsky [6].

The algorithm below uses breadth-first traversal, as discussed in Cormen *et al.* [7], as its propagation method. As in Cormen's

¹For a pixel matrix representation, this can be done in $O(c)$, that is, constant time.

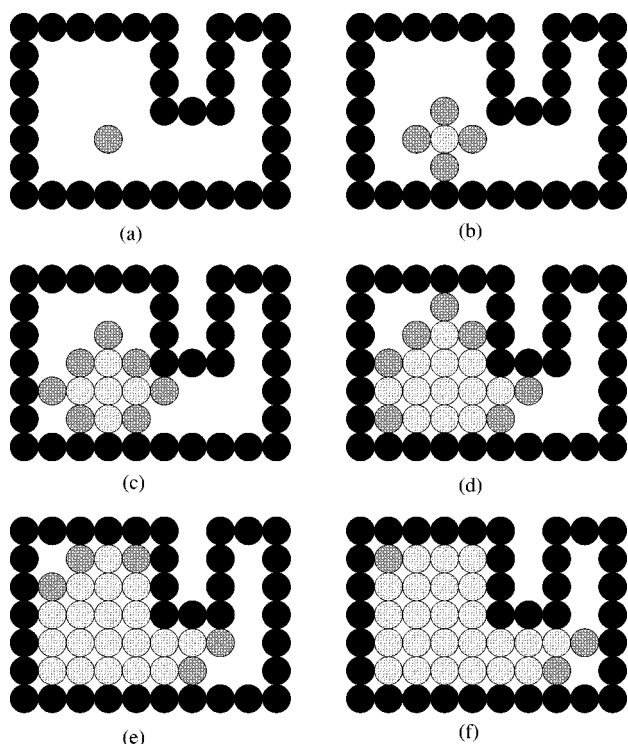


Fig. 2. Breadth-first flooding, from (a) to (f). Enqueued pixels are dark grey.

book, a queue, called Q , is used to store elements. Only those pixels that have *unexplored* neighbors are kept in the queue, so the algorithm uses memory efficiently. This is the algorithm Levoy [1] calls “pixel-based queue algorithm.”

```
FLOOD-FILL-2(initial-pixel,
initial-color,final-color)
color(initial-pixel) ← final-color
enqueue(Q, initial-pixel)
while Q not empty
  h ← head(Q)
  for each n ∈ Neighbors(h)
    if color(n) = initial-color
      color(n) ← final-color
      enqueue(Q, n)
  dequeue(Q).
```

This algorithm explores all pixels at a distance d from the initial pixel before exploring pixels at distance $d + 1$. It only keeps the pixels at the edges of the “explored” part of the region in memory. The others are taken off the queue. Its behavior is shown in Fig. 2.

The time complexity of this algorithm is easy to see: each pixel in the region is enqueued once, colored once, and dequeued once, therefore, complexity is $O(n)$, where n is the number of pixels in the region. The `while` loop is executed exactly n times, and the `for` loop is executed at most four times or eight times, depending on the neighborhood definition used.

The amount of memory used is less clearly defined. At any point, all the pixels on the border of the “expanding” region are in the queue. In the worst case, the boundary of the expanding region eventually fits the boundary of the original region. Thus, we can expect memory usage to be $O(b)$ where b is the number

of pixels in the boundary of the region. Since the area of figures generally grows quadratically with respect to their perimeter, we can expect $b \propto \sqrt{n}$.

III. STORING BOUNDARY PIXELS

In the flooding algorithm above, *propagation* started at a seed pixel in the region, and proceeded until reaching the region boundary. In the algorithms below, however, propagation starts at the boundary of the region and proceeds inwards (outwards in dilation). To accomplish this, the boundary pixels of the region must be stored in a queue.

Once one pixel in the boundary of the region is found, the procedure below can find all others in $O(b)$ steps², where b is the number of boundary pixels. Assume all foreground pixels are white, all background pixels black. We define a boundary pixel as a white pixel with at least one black 8-neighbor. In the algorithm below we check if a pixel belongs to the boundary with the procedure `is-boundary-pixel`.

```
STORE-BOUNDARY(initial-pixel)
color(initial-pixel) ← grey
enqueue(Q, initial-pixel)
enqueue(Aux, initial-pixel)
while Aux not empty
  h ← head(Aux)
  for each n ∈ Neighbors(h)
    if is-boundary-pixel(n)
      color(n) ← grey
      enqueue(Q, n)
      enqueue(Aux, n)
  dequeue(Aux).
```

This algorithm colors boundary pixels grey so that they are explored only once. On exit from the algorithm, Q holds all boundary pixels colored grey. As we will see, keeping boundary pixels colored in a distinctive way is useful.

Of course, in order to find the first boundary pixel we need to scan the image—perhaps completely. If the image contains more than one foreground region, a seed pixel must be found for each region, then `STORE-BOUNDARY` must be applied to each seed pixel. Therefore, search for seed points requires $O(N)$ steps, N being the number of pixels in the image, and storage of boundary pixels requires $O(\sum b_i)$, where b_i is the number of boundary pixels in the i th region.

IV. REPEATED EROSIONS

In computer vision, the erosion of a region is the deletion of its boundary pixels. This operation is useful in many instances, for example the elimination of noise in the boundary. It’s easy to get erosion wrong, by scanning the image row by row, and removing boundary pixels from it. If this is done, the unexplored neighbor of a boundary pixel becomes a new boundary pixel, liable to be removed when explored in the same scan. That is, this method may delete all the pixels in the region!

²This is valid only for simply connected regions. A multiply connected region has several disjoint sub-boundaries, and we must store each of them by applying `STORE-BOUNDARY` to a seed pixel in the sub-boundary.

This is an example of *sequential* algorithm, that is, one that assumes each pixel can be modified at any time. Erosion seems to require that all pixels be processed simultaneously; thus, a *parallel* algorithm is needed.

Implementing erosion as a parallel algorithm is usually done in one of two ways. One: making a copy of the image, scanning the *original* image, and deleting pixels (coloring them black) as necessary from the *copy* image. Once the original image has been fully scanned, the copy image holds the eroded region. Two: scanning the image, and coloring boundary pixels using a special color or grey-level. On a second scan of the image, the “special” pixels are deleted.

Both algorithms solve the problem for one erosion, but if we want to perform a new erosion, we must again scan the complete image, which is wasteful. This is the method usually presented in textbooks [8], [9], and it requires $O(iN)$ steps, where i is the number of iterations desired, and N is the number of pixels in the image.

The ideas in the filling algorithm above provide grounds for the next algorithm, which is based on breadth-first traversal, only this time, *from the boundary inwards*. The algorithm assumes all pixels should be previously stored in a queue, as shown in Section III.

Assume the pixels in the region of interest are white, and all other pixels black. We start with all boundary pixels colored grey and stored in a queue. Then, we put a special symbol, NULL, at the rear of the queue. Take the head of the queue. If it is not the NULL symbol, it is a pixel. Scan its neighbors, and for each that is white, color it grey and put it in the queue (after NULL). Then color the head pixel black and remove it from the queue. When the NULL symbol is reached, one erosion has been completed, and the boundary pixels of the new, eroded region, are stored in the queue, following NULL, and colored grey. If a new erosion is desired, remove NULL from the head of the queue, put it at the rear, and repeat.

The procedure below takes as parameters the number of erosions desired, and the boundary queue, with its pixels colored grey, as obtained by applying STORE-BOUNDARY.

```

ERODE (num-erosions, Q)
while num-erosions > 0
  enqueue(Q, NULL)
  h ← head(Q)
  while h ≠ NULL
    for each n ∈ Neighbors(h)
      if color(n) = white
        color(n) ← grey,
        enqueue(Q, n)
    color(h) ← black
    dequeue(Q)
    h ← head(Q)
  dequeue(Q) ;; eliminate NULL
  num-erosions ← num-erosions-1.

```

In each iteration, we start with the boundary pixels stored in Q , followed by NULL. We then add the “new” boundary pixels after NULL, and remove “old” boundary pixels from the head of the queue. That is, after NULL we store the foreground neighbors of the pixels before NULL. This means successive

layers of pixels are separated by NULL, like an onion. Fig. 3 shows how it works.

This algorithm has several advantages. In the first place, only those pixels to be removed, and their neighbors, are scanned. This is much quicker than re-scanning the complete image for each erosion. In the second place, as was noted for filling, the queue holds only those pixels whose neighbors haven’t been explored. Since the boundary of the image tends to shrink with each erosion, we can expect to store at most as many pixels as there are boundary pixels in the original image. As in the filling algorithm, time complexity is $O(d)$, where d is the number of pixels to be deleted by the erosions, or equivalently, $O(\sum_{j=1}^i b_j)$, where b_j is the number of boundary pixels before the j th iteration and i is the total number of erosions. Memory usage is $O(b)$, where b is the number of boundary pixels in the initial region.

In contrast, multiple erosions as implemented in textbooks take $O(iN)$ time, that is, i scans of the full image. Memory usage would be $O(c)$, since only the “current” pixel is stored at any time.

Also, as we will see, this algorithm can be easily adapted for *dilation*, *thinning* or *skeletonization*, and *distance transformation*.

The adaptation to dilation is trivial. Dilation consists of adding a new layer of pixels to the boundary of the region. It can be thought of as the inverse of erosion, though it cannot always recreate the exact pre-eroded image. To adapt the algorithm above to dilation, enqueue those neighbors of the head of the queue that belong to the *background*, and color the head of the queue *white* before dequeuing it. The same efficiency computations apply as for erosion, only the memory used is $O(B)$, where B is the number of pixels in the boundary after the dilations have been performed.

V. DISTANCE TRANSFORMATION

Distance transformation consists of assigning to each pixel in a region its distance to the boundary. It bears strong resemblance to erosion, in that one erosion deletes boundary pixels, whose distance to the boundary is, of course, 0, and each iterated erosion deletes pixels at a distance 1 greater than the previous iteration. It is convenient, then, to adapt erosion to this problem. In the following procedure, we start counting distances from 1 to avoid collisions with black pixels, whose grey-level is 0. Again, it takes the boundary queue as its parameter.

```

DISTANCE-TRANSFORM (Q)
distance ← 1
for each b in Q
  color (b) ← distance
while Q not empty
  distance ← distance+1
  enqueue(Q, NULL)
  h ← head(Q)
  while h ≠ NULL
    for each n ∈ Neighbors(h)
      if color (n) = white
        color(n) ← distance
        enqueue(Q, n)
  dequeue (Q)
  h ← head(Q)
dequeue(Q) ;; remove NULL

```

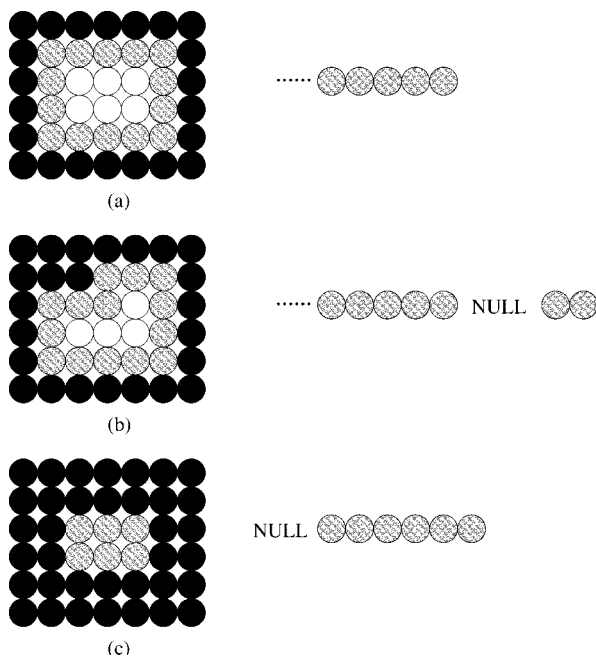


Fig. 3. Erosion. Boundary pixels are grey. At the right we see the boundary queue, with the tail at the right. (a) Shows the beginning of the erosion; in (c) one erosion has been completed, and the NULL symbol reaches the head of the queue.

Fig. 4 shows how it works.

This procedure requires only one pass through the image, and, better than that, scans only pixels in the region. Efficiency considerations are identical to those of iterated erosions, that is, $O(n)$ steps, where n is the number of pixels in the region, and $O(b)$ memory used, where b is the number of boundary pixels. This is in contrast with the *chamfer algorithm* [3], which requires two passes over the image. The algorithm above is actually a fast way of computing the basic distance transform. It can be easily extended to use the hexagonal grid, to support chamfer distance transforms [3], or even to propagate distance vectors, rather than distances (as in Danielsson’s algorithm [10]) in order to get more accurate results.

To use chamfering, pixels in the region yet “unexplored” by the procedure should not collide with explored ones in the determination of the minimal distance. This means that if distances are written on the image itself (convenient since this doesn’t require extra storage) the greylevels of pixels in the region should be distinctive or very high.

VI. SKELETONIZATION

Skeletonization or thinning is a technique widely used in pattern recognition, which consists of removing pixels from the region until only a *skeleton*, one pixel wide, is left. Ideally, the skeleton contains all the relevant information from the region, and in particular, it should preserve connectedness and bear resemblance to the original image.

Many approaches have been tried for the problem, and it has proved difficult to find fast algorithms that produce recognizable skeletons. The next algorithm adapts the ideas used in the algorithms above, and is similar—but more efficient—to that of Zhang and Suen [4].

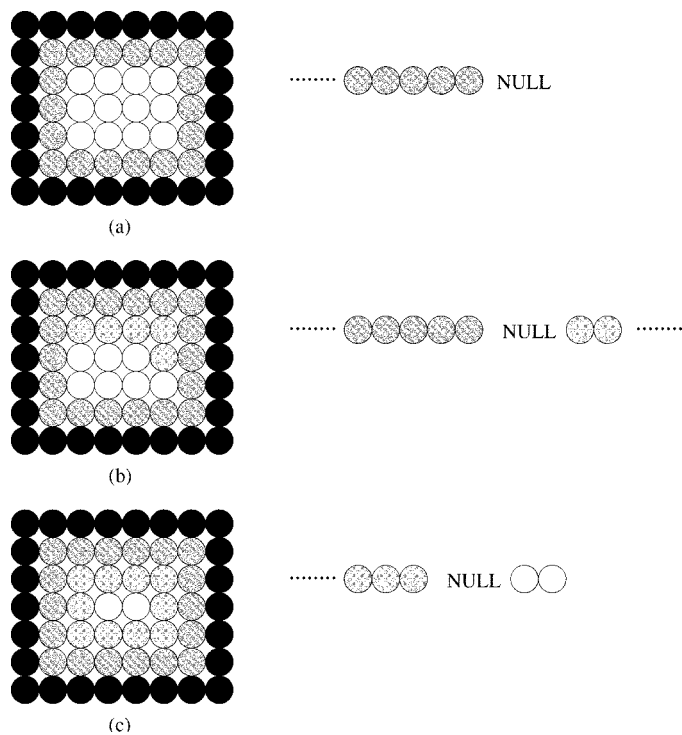


Fig. 4. Distance transform of region. At the right we see the boundary queue, tail to the right. (a)–(c) show different layers of the region, each given a different greylevel. In the boundary queue, each layer is separated by NULL.

The idea is that skeletonizing should be implemented by removing layer after layer of boundary pixels, like iterated erosions, only taking care not to delete some pixels which are key in preserving connectedness and shape in the region. Pixels that can be safely deleted are distinguished by two characteristics, according to Zhang-Suen

- their crossing index is 1;
- they have more than one and less than seven foreground 8-neighbors.

The crossing index can be obtained by traversing the 8-neighbors in order, and counting the number of transitions from grey to any other color (since boundary pixels are grey).

A skeletonization algorithm can be easily derived from the multiple erosion algorithm above: until the boundary queue is empty, for each foreground neighbor of the head of the queue, if it can be removed according to Zhang-Suen, enqueue it and color it grey; if not, give it a special color or greylevel and discard it. When the boundary queue is left empty, the “special” pixels comprise the skeleton.

Efficiency considerations are the same as for the distance transform. Again, the algorithm requires only one pass over the *region*, while traditionally several scans of the whole image are needed. This algorithm works well only for regions with a very smooth boundary. Regions with a jagged boundary give false skeletal lines. More sophisticated ways of computing the skeleton of an image have been defined (see Borgefors [3]). The algorithm discussed here is basic; it was meant to show the power of using the boundary queue together with breadth-first search. As before, the boundary queue technique can serve to implement more sophisticated versions of skeletonization efficiently.

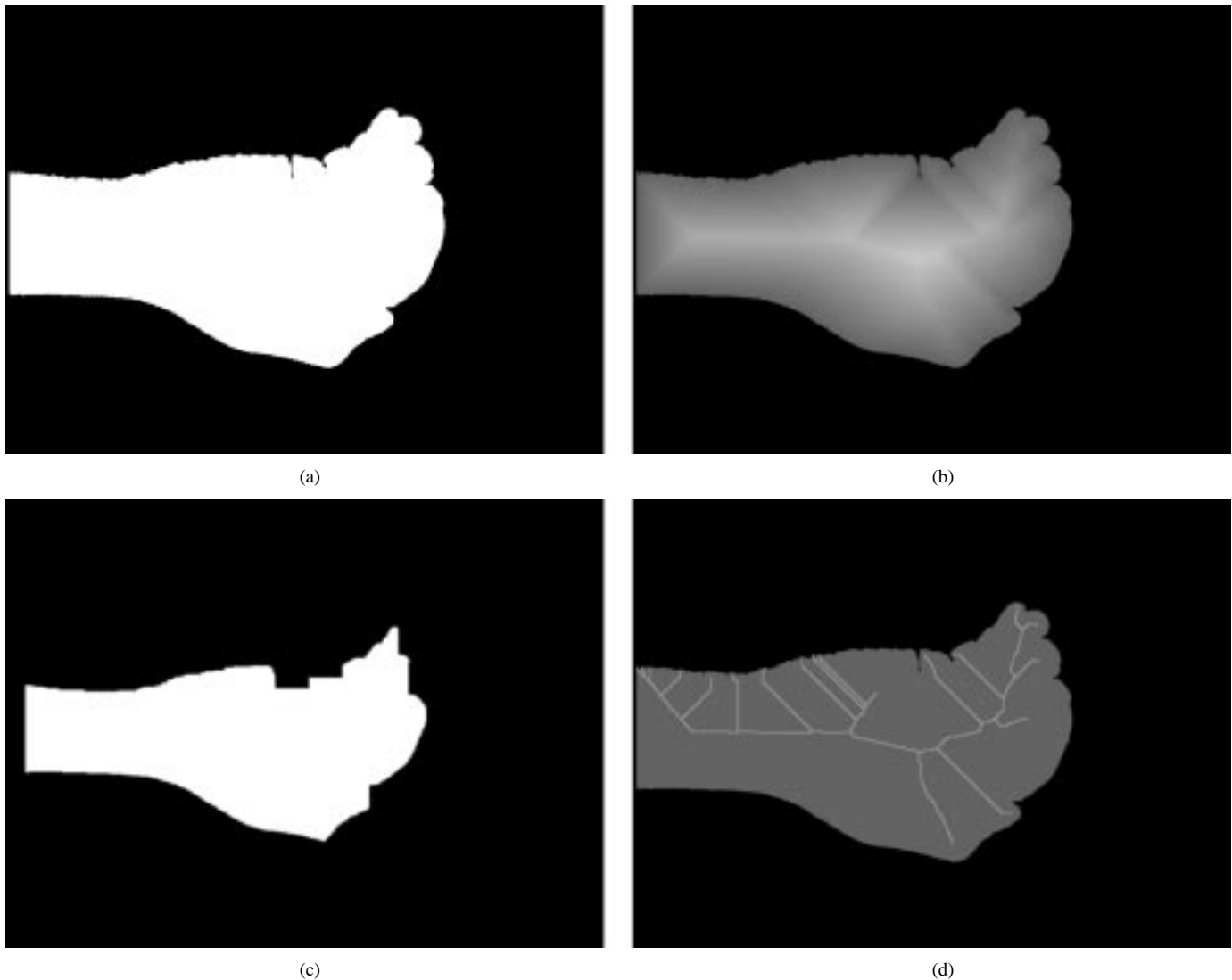


Fig. 5. Results of the operations. (a) Original region, (b) after distance transformation, (c) after 20 erosions, and (d) after skeletonization (foreground pixels are shown to aid comparison).

TABLE I
TIME TAKEN BY EACH OPERATION

Operation	Time (s.)	
	with BFS	without BFS
Flood-fill	0.14	—
Store boundary	0.20	—
Distance transform	0.18	6.98
Skeletonize	0.50	9.85
100 erosions	0.18	9.27
100 dilations	0.34	31.14

VII. CONCLUSIONS

All the algorithms above (except the first), use a boundary queue, which holds boundary pixels colored so they stand out from both background and foreground. It would be

convenient to obtain the boundary queue only once, using STORE-BOUNDARY; this requires that the boundary queue be a global variable. The procedures in this article, then, would use and modify this global queue, like filters. This way, we could cascade operations without recomputing the boundary.

The algorithms above prove that breadth-first traversal, and queues, provide a simple way of expressing algorithms that depend on traversing layer after layer of a region. They are easily translated into code, and produce short, efficient programs.

The author has written all of them in C++, using the Standard Template Library (STL) implementation of queues. Distance transform, for instance, is implemented with a 22 line function (including braces and declarations). The algorithms have been tried on TIFF images of 768×576 pixels and 8 bits per pixel. As time complexity for all algorithms is proportional to the size of the region, they have been timed for a substantial (116 400 pixels), irregular region, on a Pentium 166 MHz running Linux. They have been compared against implementations of the same

algorithms in the iterative, “intuitive” way presented in introductory textbooks [8], [9]. Table I shows the results of these comparisons. One can see the result of applying the algorithms to a test image in Fig. 5.

ACKNOWLEDGMENT

The first author would like to acknowledge the help of C. G. Parodi during the preparation and editing of the manuscript.

REFERENCES

- [1] M. Levoy, “Area flooding algorithms,” Hanna-Barbera Prod., Tech. Rep., June 1981.
- [2] L. Vincent, “Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms,” *IEEE Trans. Image Processing*, vol. 2, pp. 176–200, Apr. 1993.
- [3] G. Borgefors and G. S. di Baja, “Skeletonizing the distance transform on the hexagonal grid,” in *Proc. 9th Int. Conf. Pattern Recognition*, 1988.
- [4] T. Y. Zhang and C. Y. Suen, “A fast parallel algorithm for thinning digital patterns,” *Commun. ACM*, vol. 27, pp. 236–239, 1984.
- [5] A. R. Smith, “Tint fill,” in *Proc. SIGGRAPH '79*, 1979, pp. 276–283.
- [6] K. P. Fishkin and B. A. Barsky, “A family of new algorithms for soft filling,” in *Proc. SIGGRAPH '84*, 1984, pp. 235–244.
- [7] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
- [8] J. R. Parker, *Practical Computer Vision Using C*. New York: Wiley, 1993.
- [9] E. R. Davies, *Machine Vision: Theory, Algorithms, Practicalities*, 2nd ed. London, U.K.: Academic, 1997.
- [10] L. Vincent, “Exact Euclidean distance function by chain propagations,” *Comput. Vis. Pattern Recognit.*, pp. 583–598, 1991.



Jaime Silvela received the M.S. degree in electrical engineering from the Polytechnic University of Madrid, Madrid, Spain, in 2000.

He is currently with Nortel Networks, Madrid, where he specializes in voice over IP. His interests are mathematical analysis, computer languages, image processing, and artificial intelligence.



Javier Portillo received the telecommunication engineering degree in 1985 and the Ph.D. in telecommunication engineering in 1991, both from the Polytechnic University of Madrid, Madrid, Spain.

Currently, he is Professor with the Signal, System and Radiocommunication Department, Telecommunication Engineering School, Polytechnic University of Madrid. His research interests are image processing, computer vision, and simulation. He is author or co-author of many research papers and technical reports and he has worked in or managed

more than 30 research public projects and private projects with enterprises.