Computer Methods and Programs in Biomedicine (2004) **76**, 1—11

ELSEVIER

Computer Methods and Programs in Biomedicine

www.intl.elsevierhealth.com/journals/cmpb

An efficient filling algorithm for counting regions

W.G.M. Geraets*, **A.N. van Daatselaar**, **J.G.C. Verheij**

Academic Center for Dentistry Amsterdam (ACTA) Radiology Department, Louwesweg 1, 1066 EA Amsterdam, The Netherlands

Received 22 October 2002; received in revised form 18 September 2003; accepted 19 September 2003

KEYWORDS
Region filling;
Seed filling;
Counting regions in binary images;
Connectivity

Summary Region filling has many applications in computer graphics and image analysis. Some region filling tasks can be performed by fast scan line filling algorithms. Other region filling tasks require seed filling algorithms which are more general but slower. This paper introduces a seed filling algorithm that is designed to count regions irrespective of their shape. The method is described and its performance is compared with three alternative algorithms by applying them to a collection of 34 test images. The four methods showed complete agreement with respect to the counted numbers of regions. The proposed method was found to be fastest and requiring least memory.

1. Introduction

Region filling has many applications in computer graphics and image analysis. At the ACTA Radiology Department, it is used in the analysis of radiographs of bone. In bones of the human skeleton many changes are observed during growth, exercise regimes, and fracture healing. Considerable amounts of bone mass are lost in situations with reduced mechanical loading, for example during space flights or bed rest. The strength of bones is determined by the amount of bone mass as well as by the porous structures inside the bones. For the purpose of studying growth, diagnosing diseases, and for the development of therapies there is a need for methods to investigate the internal structure of bone. In patients with osteoporosis, small samples of bone are taken both for diagnosis as well as for research. The invasive nature of this approach puts strong restrictions on its applicability. Tomography with X-rays represents a powerful non-invasive technique to obtain 3D representations of the bone structure, however, the applicability of X-ray tomography systems is limited due to the radiation burden for patients. Plain film radiography does not record the exact architecture of bone, but rather the overlapping X-ray shadows, produced by calcified tissues at different distances to the X-ray source, however, it is a non-destructive, inexpensive, and widely available technique used for obtaining information about bone structure in vivo.

The increasing speed and power of computer systems have made it possible to automate image analysis techniques that are very tedious to perform by hand. These developments have stimulated the research on bone structure and its relation to mechanical properties. At the ACTA Radiology Department several studies have been done to increase the diagnostic yield of plain film radiographs of bone [1—5]. Areas of interest on radiographs of bone were scanned and filtered to obtain binary images as illustrated in Fig. 1. Several measuring modules were designed, implemented, and evaluated. Evidence was found that the numbers of regions have clinical relevance with respect to osteoporosis.

*Corresponding author. Tel.: +31-20-5188517.
E-mail address: wgeraets@acta.nl (W.G.M. Geraets).

0169-2607/$ — see front matter
doi:10.1016/j.cmpb.2003.09.004

Fig. 1 Binary pattern originating from a radiograph of bone containing 52 dark regions and 33 bright regions.

2. Background

An example of the images to be analyzed is shown in Fig. 1. The entire image is composed of dark and bright regions. Some regions have simple shapes, while others have complicated shapes with branches, holes, and islands. The smallest possible region consists of one single pixel, but regions can also be large, in extreme cases the entire image might consist of one large region. Mathematically, a region is defined as a collection of interconnected pixels with the same grayvalue. Any two member pixels of a region must be interconnected by fellow members of the region. If every two pixels of the region can be joined using only up, down, left or right moves then the region is 4-connected. This is the type of regions considered in this paper. When diagonal moves are also taken into consideration then the region is 8-connected, thus every 4-connected region is also 8-connected [6—8].

2.1. Scan line filling

In many applications the images to be filled have a property which can be used to simplify and speed up the region filling algorithm. For example when it is known that not any dark region touches the image border a category of fill algorithms can be used that are called ''edge filling'', ''scan line filling'', or ''parity check filling'' [9—11].

Scan line fill algorithms are based on the principle that in many cases a straight line intersects the contour of a region an even number of times. If we know that the first point of the scanline lies outside the region, then we can traverse it and decide which segments are in the interior by counting the number of intersections. If the number is odd, then the segment belongs to the interior. However, one may find situations in which the principle leads to incorrect filling, for example when parts of the contour are parallel to the intersecting straight line [11—13]. Moreover, if scan line algorithms have to cope with regions touching the image boundaries then they become more complex and slower [14,15]. Complications due to regions intersecting the image borders are also reported by Xu et al. [16] using a software package dedicated to the analysis of microscopic images containing different types of cells. The operator has several tools to correct mistakes happening when cells intersect with the border.

Because regions intersecting the image borders are very common in images of bone structure as illustrated in Fig. 1, it was decided to put no restrictions on size, shape, or location of the regions in the input image and that the precision of the filling method should not be compromised for the sake of speed.

2.2. Seed filling

Another family of filling algorithms is known in literature as ''seed filling'', ''connectivity filling'' [11], or ''region growing''. This type of algorithm consists of two basic parts. The first part searches for a start point (seed) of a new region, and the second part searches all pixels in the image that are connected with the seed point and provides them with a mark which indicates that they have been taken into account.

In some applications the seed points are searched automatically, in others the seed points are chosen by the user [17—19]. All seed filling algorithms need working memory for keeping record of pixels that have been classified. Sivewright and Elliot [19] report a 3D filling algorithm that helps marking important anatomical structures in CT and MR images. The method requires one empty work image and a list of boundary points. Henrich [20] describes several seed filling algorithms that make optimal use of the existing image buffer and require less additional memory.

2.3. Special purpose filling methods

Shi and Govindaraju [21] describe another region growing algorithm that is applied to fill small gaps in handwritten characters, and to improve automatic

recognition. Lind and Hrycej [22] and Shani [6] address the problem of filling with a pattern rather than with a specific fill ''color''. Atkinson et al. [13] describe a filling algorithm that is to be used on 2D and 3D images that are composed of blocks and cubes of various sizes.

In hospitals and other places where large amounts of images are used, there is a need to reduce the space needed to store the images. One of the methods for the compression of binary images is to store only the contours of each image. When the image needs to be decompressed a filling algorithm is applied to fill in the contours, thus obtaining the original image [12,23]. Tsai and Chung [24] describe a region filling algorithm that is to be used in combination with bincode data compression for binary images. Tang and Lien [25] describe a region filling method that is to be applied in combination with the Freeman code representation of contours.

2.4. Recursive methods

Some algorithms can be implemented as a function that generates calls to the same function. Recursive code is simpler and shorter than non-recursive code but also slower, due to the time required to maintain the data in the stack. If the number of recursive function calls is excessive, stack overflow errors occur. Albert and Slaaf [26] describe and compare four different recursive algorithms for seed filling. Martín et al. [14] describe a recursive method for filling the interiors of regions in a binary image. Oikarinen [15] describes a recursive seed filling method used to accelerate the rendering of 3D regions.

3. Design considerations

The proposed method of filling, named RegionCount, uses a two dimensional array of bytes with the same number of rows and columns as the pixels in the original image. This array is used to keep record of the pixels that have been identified as a member of a region. The array elements have two states: ''marked'' and ''unmarked'', indicated by 1 and 0. At the start of the analysis all array elements are unmarked (0), and when a pixel has been identified as a member of a region then the corresponding array element is marked (1). When only two states are possible then it would suffice to use an array of bits instead of bytes [20]. It is merely for convenience that bytes are used as the elements of the array.

When the algorithm searches a seedpoint it only has to find a pixel that is unmarked. According to the grayvalue of the seedpoint, either the counter of dark regions or of the bright regions is incremented. Starting with the seed point, further inner pixels are identified by a so called cursor [20]. The term cursor indicates the pixel whose neighbors are being analyzed. During the filling process, the cursor scans the entire region and marks all member pixels of the region. Because the cursor needs to move freely over the image, random access to the pixels is required. If the cursor points to a pixel with grayvalue different from the seedpoint, the case is not interesting. The most basic design feature of the algorithm acknowledges that the interesting situations are formed by two neighboring pixels having the same grayvalue as the seedpoint with one of the pixel being marked and the other unmarked. When the cursor arrives at a marked pixel having the same grayvalue as the seedpoint, its neighboring pixels are tested, and if any of them has the appropriate grayvalue then it is connected with the seedpoint, and it should be marked. When all pixels have been

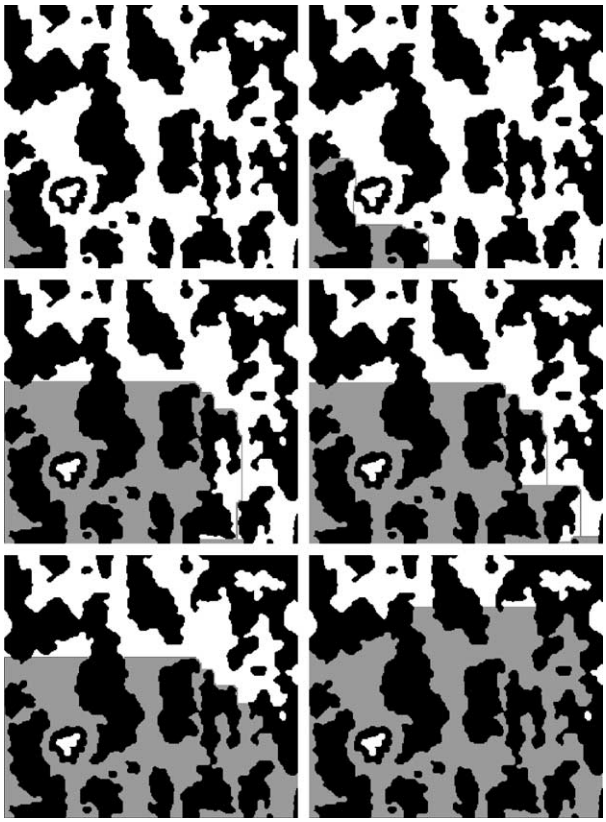

Fig. 2 Six stages in the RegionCount filling process of a huge bright region. Bright pixels that are marked are represented in gray. First the branch is filled that is connected with the seed point in the lower left corner. When this branch overflows the cursor moves down into the next branch.

marked then the algorithm has finished the job of counting the dark and bright regions.

4. System description

How the cursor fills the entire region is illustrated by Fig. 2 showing six steps in the RegionCount process of filling a huge bright region with a complex shape. Bright pixels that are marked are represented in gray. First the branch is filled that is connected with the seed point in the lower left corner. When this branch is filled, the cursor moves deep down into another branch, which is filled until it can move downward into yet a third branch. The filling process resembles the flow of water in a porous material: water moves downward whenever it can, and so does the cursor.

The filling algorithm is described in more detail by the flow diagram in Fig. 3. The flow diagram contains 17 decision blocks of which 12 are marked by capitals A to L. The search for a seedpoint is dealt with in blocks A—C and the search for the fellow member pixels of the region is done in blocks D—K. The tendency of the cursor to move downward is caused by block D. If the lower neighbor of the cursor has the proper grayvalue and if it is unmarked then the cursor moves 1 step down and program flow is redirected to block D. Only if the program flow reaches block H the cursor can move upward.

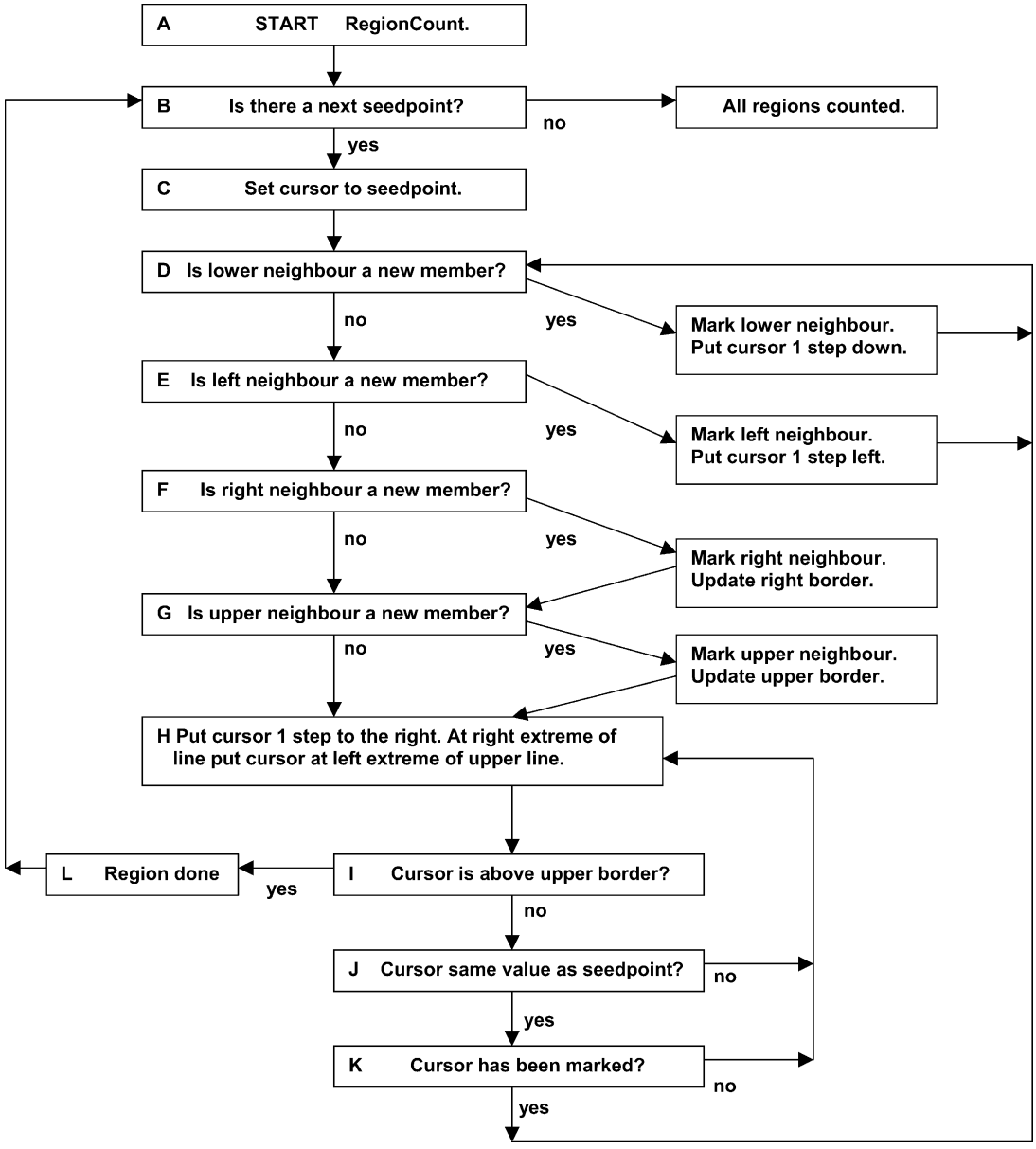

Fig. 3 Flow diagram of the proposed region filling algorithm.

The position of the cursor changes only in blocks D, E and H. In block H, the cursor first tries to step to the right, and when arriving at the right extreme of the image, it jumps to the left extreme of the line above the present one. This pattern of movement is complementary to the tendency of movement in blocks D and E. These movements of the cursor guarantee that all member pixels of the region will be visited. In order to explain the algorithm further, the original C code is provided with many comments in Appendix A.

5. Evaluation and conclusions

In order to evaluate the performance of the present algorithm for region counting, three alternative seed fill algorithms were implemented on the same PC. The four methods search for seedpoints in similar ways but they have different strategies for finding the pixels that are connected to the seedpoint. The ''column method'' searches the image for an uncounted pixel bordering the counted pixels. If one is found, all pixels above and below with the proper grayvalue are counted and marked as counted. The process is repeated until no more uncounted pixels are found. The ''tracing method'' fills a region by going clockwise around pixels that have been recognized as members of the present region. When a pixel of different grayvalue is encountered, a bifurcation may occur, in which case a stack is used to keep track of locations that need further analysis. ''Classical recursive'' fills a region in the classical recursive way, starting at the seed point. If the direct neighbors have the proper grayvalue and if they have not yet been counted then the recursion deepens and the new neighbors are tested.

In order to keep record of the pixels that have been identified a 2D array of bits is used both by the column method and by the tracing method similar to the RegionCount method. In order to store pixels requiring further analysis a stack is used by the tracing method and by the classical recursive method. In the worst case, the stack memory required by the classical recursive method is more than 180 times the memory needed by the RegionCount method and the column method. Although the classical recursive method will not use the maximum amount of stack memory constantly, reserving less stack memory can lead to stack overflow and malfunctioning.

The performance of the present algorithm and the three alternatives was measured using a set of test images, consisting of 17 segmented images of 256×256 pixels plus their enlarged versions measuring 512×512 pixels as illustrated by Figs. 4–7. The four methods yielded identical numbers of dark and bright regions. For example, the four methods agreed that the image in Fig. 4 contains 20 black and 4 white regions. Without exception the numbers of regions counted by the four methods were in complete agreement on all test images. In order to compare the speed of the algorithms the processing times were measured and summarized in Table 1.

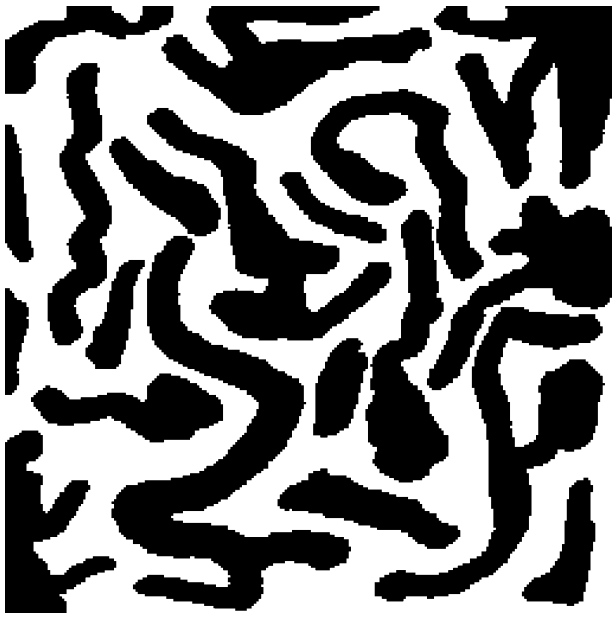

Fig. 4 Test image with 20 dark regions and 4 bright regions.

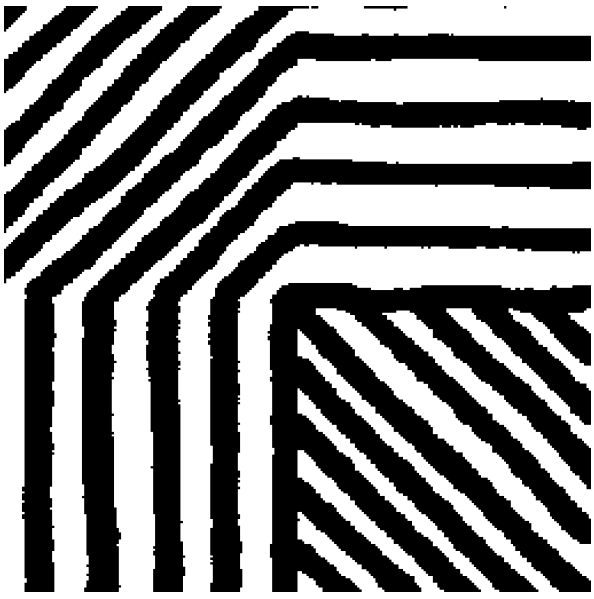

Fig. 5 Test image with 13 dark regions and 19 bright regions.

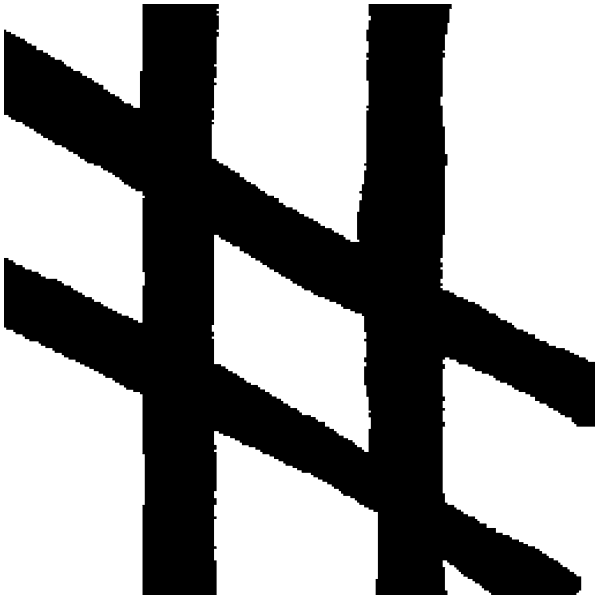

Fig. 6 Test image with 1 dark region and 9 bright regions.

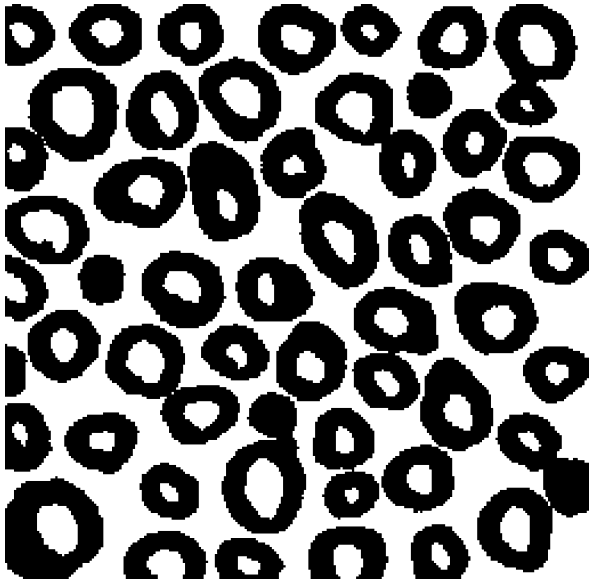

Fig. 7 Test image with 46 dark regions and 59 bright regions.

Table 1 suggests that the RegionCount method is faster than the other methods, however, when applied to the smaller input images the difference with the classical recursive method is insignificant. In all four methods, the processing time depends not only on the size of the input image but also on the geometry of the input image. In the set of small input images, as well as in the set of large images the processing time of the column method was found to vary more than a factor 10. The processing time of the RegionCount method was found to vary less than a factor 4.

The ''column method'' is the slowest of the four methods. The time required to process the large images is disproportionally larger than the time needed to process the smaller images. While the area of the larger images is four times the area of the smaller ones, the time required for analyzing the large images with the column method is more than seven times the processing time of the smaller images. The processing time of the three other methods seems proportional to the area of the input image.

6. Discussion

Figs. 4–7 illustrate the variability of the test cases, however, it is impossible to select a limited set of test cases representing the virtually infinite number of images that must be analyzed correctly. This hampers comparison of the algorithms by experiment and it seems unavoidable to settle for a comparison that is not completely conclusive.

In general, the speed of an image processing algorithm depends on the program design, speed of the hardware, and the size of the image to be analyzed. Moreover, the speed of filling algorithms is also

Table 1 Processing time (in ms) (mean ± S.D., $n = 17$)

	Size of image	
	256 × 256	512 × 512
RegionCount	32.0 ± 2.9	126.1 ± 11.5
Tracing method	35.8 ± 0.4**	144.7 ± 1.6**
Classical recursive	33.1 ± 0.8 NS	175.9 ± 4.2**
Column method	776.1 ± 111.5**	5681.2 ± 863.4**

NS: no significant difference from RegionCount.
** Significantly different from RegionCount according to Student's *t*-test.

influenced by the geometry of the input image. This causes non-zero standard deviations in Table 1. Both the tracing method as well as the classical recursive method seem to depend much less on the geometry of the input images than the RegionCount method. This might be explained by the use of stack memory to store pixels for further analysis. Taking the next pixel to be analyzed from the stack requires a fixed amount of time, whereas ''geometrical'' algorithms need more time to find the next pixel to be analyzed when it is located at greater distance.

Fig. 1 shows various regions meeting the boundaries of the image, a condition that is common in images representing bone patterns. Because of this condition the scan line filling algorithms do not function properly [15]. Some authors propose to tackle this problem by increasing the size of the image with the addition of an extra layer of pixels [14]. However, this solution can affect the connectedness of the regions, leading to incorrect results. Consider, for example, Fig. 4. In case an extra layer of bright pixels is put around it then the number of bright regions will drop from 4 to 1. And if the extra layer is dark, then the number of dark regions drops from 20 to 14.

7. Future plans

The presented method for counting regions has been used for research on bone structure, however, it is expected that the method is also of interest to other image analysis applications in the medical and industrial fields. The algorithm can be modified into a manifold of powerful image transformations, for example, a ''labeling transformation'' in which the pixels of each region are assigned a unique label [18,19]. By means of the labels it becomes easier to select regions according to location, size, area or shape, thus facilitating the design of ''intelligent'' filters.

Appendix A

In order to explain the algorithm further, the original C code is provided with many comments. The capitals A to L correspond with capitals A to L in Fig. 3.

(A) When an image is to be analyzed the module RegionCount is called and pointers are provided to variables iBlacks and iWhites for returning the numbers of dark and bright regions. For storage of grayvalues bSeed is used. Before the counting starts the numbers of dark and bright regions iBlacks and iWhites are initiated with zeros. Also initiated with zeros are the elements of the array that is used for keeping record of the pixels that have been identified as a member of a region, this array of bytes is indicated as notepad memory. The elements of the notepad memory have two states: unmarked (0) and marked (1). The original image is stored in a part of memory indicated as Image. Width and height of the original image are provided by functions Get_Width and Get_Height. Both the pixels in the Image array as well as the bytes in the Notepad memory are accessible through the functions ReadPixel and WritePixel.

```
void RegionCount (int * iBlacks, int * lWites) {//function body
     BYTE       bSeed;
     long       KX, lWidth, XSeed, XCursor, XRight;
     long       LY, lHeight, YSeed, YCursor, YUpper;
     iBlacks  = 0;    //initiate counter of dark regions
     iWhites  = 0;    //initiate counter of bright regions
     lWidth   = Get_Width (Image); //store width of Image
     lHeight  = Get_Height(Image); //store height of Image
     for (KX = 1;  KX <= lWidth;   KX++)
     for (LY = 1;  LY <= lHeight;  LY++)
          WritePixel (Notepad, KX, LY, 0);
```

(B) The outermost loop of the program searches a seedpoint of a region. The search starts at the left column of the input image and ends at the right column. Any pixel that has been marked previously belongs to a region already analyzed and it will direct the program flow to entry point NextRegion in part L near the end of the outermost loop. The first pixel that has not been marked yet is a suitable seedpoint and causes the program flow to enter the routines in parts C—K.

```
for (XSeed = 1;  XSeed <= lWidth;  XSeed++)
for (YSeed = 1;  YSeed <= lHeight; YSeed++)  {//searching seedpoint
     if (ReadPixel (Notepad, XSeed, YSeed) > 0) goto NextRegion;
```

(C) Especially with large regions the grayvalue of the seedpoint is needed many times. Using the subroutine ReadPixel to read the grayvalue of the seedpoint every time it is needed would slow down the program unnecessarily. Therefore, the grayvalue of the seedpoint pixel is stored in the variable bSeed which is accessed more quickly than a pixel.

```
bSeed  = ReadPixel (Image, XSeed, YSeed);
```

As soon as a seedpoint has been found the counters of dark and bright regions are updated. Please note that the original image is implicitly segmented into dark and bright areas with one grayvalue between 0 and 127 and the other grayvalue between 128 and 255.

```
if (bSeed <= 127) iBlacks++;
if (bSeed >= 128) iWhites++;
```

The seedpoint needs to be marked, it is the first pixel identified as a member of a new region. The other member pixels of the region, if any, still need to be identified.

```
WritePixel (Notepad, XSeed, YSeed, 1);
```

The cursor point is positioned at the newly found seedpoint.

```
XCursor = XSeed;
YCursor = YSeed;
```

A simple bookkeeping is initiated to increase program efficiency.

```
XRight = XSeed;
YUpper = YSeed;
```

(D) When relating pixel coordinates to directions up, down, left and right it must be considered that the pixel with coordinates (1, 1) is located in the lower left corner of the image, and that the positive *X*-axis is pointing to the right while the positive *Y*-axis is pointing upward. Therefore, the right border of the image corresponds with the highest *X*-coordinate, and the image line at the top of the image corresponds with the highest *Y*-coordinate.

When investigating the aspect of connectivity between the pixels constituting the original image it is essential that all pixels of the image are investigated, including the pixels at the border of the image. Although border pixels have less neighbors than internal pixels, they need to be studied as well. In the present algorithm all border pixels are included and compared with the neighboring pixels that are part of the image to be analyzed; comparison of border pixels with neighboring pixels outside of the image is avoided by means of the leading ''if-statement'' in blocks D, E, F, and G.

Once a seedpoint has been found the program flow enters control block D which tries to put the cursor closer to the bottom of the image. If the lower neighbor of the cursor has the same grayvalue as the seedpoint and has not been marked yet, the lower neighbor is marked, the cursor is moved 1 pixel down, and then again the next lower neighboring pixel is tested.

```
WaterFall:
if (YCursor - 1 >= 1)
if (ReadPixel (Image,   XCursor, YCursor - 1) == bSeed)
if (ReadPixel (Notepad, XCursor, YCursor - 1) ==     0)
   {WritePixel (Notepad, XCursor, YCursor - 1, 1);
    YCursor = YCursor - 1;
    goto WaterFall;}
```

(E) If the left neighbor of the cursor has the same grayvalue as the seedpoint and has not yet been marked, then another member pixel of the region has been found. This new member is marked, the cursor moves 1 pixel to the left, and program flow jumps to the top of the previous control block D.

```
if (XCursor - 1 >= 1)
if (ReadPixel (Image,   XCursor - 1, YCursor) == bSeed)
if (ReadPixel (Notepad, XCursor - 1, YCursor) ==     0)
   {WritePixel (Notepad, XCursor - 1, YCursor, 1);
    XCursor = XCursor - 1;
    goto WaterFall;}
```

Control blocks D and E control the ''water fall'' movement, intended to let the cursor move downward whenever possible. Moving the cursor down has higher priority than moving the cursor to the left.

(F) If the right neighbor of the cursor has the same grayvalue as the seedpoint, and has not yet been marked, the right neighbor is marked and the bookkeeping is updated.

```
if (XCursor + 1 <= lWidth)
if (ReadPixel (Image,   XCursor + 1, YCursor) == bSeed)
if (ReadPixel (Notepad, XCursor + 1, YCursor) ==     0)
   {WritePixel (Notepad, XCursor + 1, YCursor, 1);
    if (XRight < XCursor + 1) XRight = XCursor + 1;}
```

(G) If the upper neighbor of the cursor has the proper grayvalue and has not yet been marked, the upper neighboring pixel is marked and the bookkeeping is updated. Please note that control blocks F and G leave the position of the cursor unchanged.

```
if (YCursor + 1 <= lHeight)
if (ReadPixel (Image,   XCursor, YCursor + 1) == bSeed)
if (ReadPixel (Notepad, XCursor, YCursor + 1) ==     0)
   {WritePixel (Notepad, XCursor, YCursor + 1, 1);
    if (Yupper < YCursor + 1) YUpper = YCursor + 1;}
```

(H) In control block B a loop with increasing *Y* parameter is nested in a loop with increasing *X* parameter. Therefore it can be understood that any seedpoint encountered by this loop will be located at the utmost left side of the region. Thus, XSeed equals the minimum value of *X*-coordinates of all pixels in the region. With respect to the maximum *X*-coordinate of the cursor it can be understood that connectivity is violated if XCursor>XRight. These properties are used to increase program efficiency; instead of letting the cursor move over the full width of the image it is confined between columns XSeed and XRight.

```
WaterRising:
XCursor = XCursor + 1;
if (XCursor > XRight)
   {XCursor = XSeed;
    YCursor = YCursor + 1;}
```

Please note that the position of the cursor pixel is moved only in blocks D, E and H. In block H the cursor moves from left to right and from down to up. This direction of movement is called ''water rising'' movement, and it is complementary to the ''water fall'' movement described in E.

(I) A simple bookkeeping was initiated in control block C and updated in blocks F and G every time a pixel was marked. Thus, parameter YUpper refers to the highest image line in which a pixel has been found that was connected with the present seedpoint. Considering that new member pixels are found only when the cursor pixel has been marked, it is understood that there is no need to move the cursor above the YUpper line. When the cursor arrives above the upper line the region has been filled completely.

```
if (YCursor > YUpper) goto NextRegion;
```

(J) If the cursor does not have the proper grayvalue then the situation is not interesting and program flow is directed to control block H where the position of the cursor will be updated.

```
if (ReadPixel (Image, XCursor, YCursor) != bSeed) goto WaterRising;
```

(K) If the cursor has been marked it is necessary to investigate the neighboring pixels by directing program flow to entry point WaterFall in control block D. If the cursor is positioned at an unmarked pixel, then it is no use to test neighboring pixels because connectedness of the region cannot be guaranteed, therefore, the control flow is directed to entry point WaterRising in control block H where the cursor position will be updated.

```
if (ReadPixel (Notepad, XCursor, YCursor) > 0) goto WaterFall;

goto WaterRising;
```

In control blocks D and E the cursor moves from the top of the image to the bottom and from the right to the left, whereas in control block H the cursor moves from left to right and from bottom to top. This combination of two complementary movement patterns is an important design feature of the proposed method, making sure that even the most awkwardly shaped region will be scanned completely.

(L) End of outer loop searching for seedpoints. When all regions have been filled, there are no more seedpoints to be found and the program flow returns counters iBlacks and iWhites to the invoking routine.

```
                    NextRegion:;  }//searching seedpoint
return;  }//function body
```

References

[1] W.G.M. Geraets, P.F. van der Stelt, C.J. Netelenbos, P.J.M. Elders, A new method for automatic recognition of the radiographic trabecular pattern, J. Bone Miner. Res. 5 (1990) 227—233.

[2] W.G.M. Geraets, P.F. van der Stelt, Analysis of the radiographic trabecular pattern, Pattern Recogn. Lett. 12 (1991) 575—581.

[3] W.G.M. Geraets, P.F. van der Stelt, P.J.M. Elders, The radiographic trabecular bone pattern during menopause, Bone 14 (1993) 859—864.

[4] W.G.M. Geraets, Computer aided analysis of the radiographic trabecular pattern, Thesis Vrije Universiteit, Amsterdam, November 1994.

[5] W.G.M. Geraets, P.F. van der Stelt, P. Lips, F.C. van Ginkel, The radiographic trabecular pattern of hips in patients with hip fractures and in elderly control subjects, Bone 22 (1998) 165—173.

[6] Shani U., Filling regions in binary raster images: a graph-theoretic approach, in: Proceedings of the 7th annual conference on computer graphics and interactive techniques, pp. 321—327 (ACM Press, Seattle, Washington, 1980).

[7] K.P. Fishkin, B.A. Barsky, A family of new algorithms for soft filling, Comput. Graph. 18 (1984) 235—244.

[8] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics, Principles and Practice, Addison-Wesley, Reading, 1990/1996 (Chapter 19).

[9] S. Lejun, Z. Hao, A new contour fill algorithm for outlined character image generation, Comput. Graph. 19 (1995) 551—556.

[10] S.V. Burtsev, Y.P. Kuzmin, An efficient flood-filling algorithm, Comput. Graph. 17 (1993) 549—561.

[11] T. Pavlidis, Algorithms for Graphics and Image Processing, Computer Science Press, Maryland, 1982 (Chapter 8).

[12] T. Pavlidis, Filling algorithms for raster graphics, Comput. Graph. Image Process. 10 (1979) 126—141.

[13] H.H. Atkinson, I. Gargantini I, T.R.S. Walsh, Filling by quadrants or octants, Comput. Vision Graph. Image Process. 33 (1986) 138—155.

[14] M. Martín, M. Martín, C. Alberola-López, J. Ruiz-Alzola, A topology-based filling algorithm, Comput. Graph. 25 (2001) 493—509.

[15] J. Oikarinen, Using 2- and 2.5-dimensional seed filling in view lattice to accelerate volumetric rendering, Comput. Graph. 22 (1998) 745—757.

[16] Y.H. Xu, G.L. Sattler, H. Edwards, H.C. Pitot, Nuclear-labeling index analysis (NLIA), a software package used to perform accurate automation of cell nuclear-labeling index analysis on immunohistochemically stained rat liver samples, Comput. Methods Programs Biomed. 63 (2000) 55—70.

[17] C. Revol, M. Jourlin, A new minimum variance region growing algorithm for image segmentation, Pattern Recogn. Lett. 18 (1997) 249—258.

[18] A. Mehnert, P. Jackway, An improved seeded region growing algorithm, Pattern Recogn. Lett. 18 (1997) 1065—1071.

[19] G.J. Sivewright, P.J. Elliott, Interactive region and volume growing for segmenting volumes in MR and CT images, Med. Informat. 19 (1994) 71—80.

[20] D. Henrich, Space-efficient region filling in raster graphics, Visual Comput. 10 (1994) 205—215.

[21] Z. Shi, V. Govindaraju, Character image enhancement by selective region-growing, Pattern Recogn. Lett. 17 (1996) 523—527.

[22] B. Lind, T. Hrycej, Graphics Goodies #1: a filling algorithm for arbitrary regions, Comput. Graph. 21 (1987) 281—282.

[23] Y.M.Y. Hasan, L.J. Karam, Morphological reversible contour representation, IEEE Trans. Pattern Anal. Mach. Intell. 22 (2000) 227—240.
[24] Y.H. Tsai, K.L. Chung, Region-filling algorithm on bincode-based contour and its implementation, Comput. Graph. 24 (2000) 529—537.
[25] G.Y. Tang, B. Lien, Region filling with the use of the discrete Green theorem, Comput. Vision Graph. Image Process. 42 (1988) 297—305.
[26] T.A. Albert, D.W. Slaaf, A rapid regional filling technique for complex binary images, Comput. Graph. 19 (1995) 541—549.