Memory efficient propagation-based watershed and influence zone algorithms for large images

I. Pitas   C. I. Cotsaces

Department of Informatics
University of Thessaloniki
Thessaloniki 540 06, GREECE
(E-mail: pitas@zeus.csd.auth.gr)

EDICS No: *2-MRPH*

Abstract

Propagation front or grassfire methods are very popular in image processing because of their efficiency and because of their inherent geodesic nature. However, because of their random-access nature, they are inefficient in large images that cannot fit in available random access memory. In this paper, we explore ways to increase the memory efficiency of two algorithms that use propagation fronts: the skeletonization by influence zones and the watershed transform. Two algorithms are presented for the skeletonization by influence zones. The first computes the skeletonization on surfaces without storing the enclosing volume. The second performs the skeletonization without any region reference, by using only the propagation fronts. The watershed transform algorithm that was developed keeps in memory the propagation fronts and only one greylevel of the image. All three algorithms use much less memory than the ones presented in the literature so far. Several techniques have been developed in this work in order to minimize the effect of these set operations. These include fast search methods, double propagation fronts, directional propagation, and others.

I. Introduction

The concept of distance [1] is fundamental to most areas of image processing and analysis. The *geodesic* distance functions are particularly useful in many tasks that require the description of distance within a specific region. There are a number of methods that can segment an image according to a distance criterion. Such a method is the skeletonization by influence zones [2], which classifies points of the image according to their geodesic distance from a predefined number of markers. This is similar to the Voronoi tessellation [3], only the latter is a primarily continuous procedure that utilizes geometric methods and computes non-geodesic distances. Another commonly used segmentation method that relies primarily on distances is the watershed transform [4] – [6]. It also takes into consideration the greyvalues of the image when computing the segmentation. It is commonly accepted that the above methods are by far best performed by propagation fronts.

Here we shall concentrate on the use of propagation-based algorithms on images that cannot fit in available random-access memory. In the following, we shall call such images simply *large images*. Such are the three-dimensional images, whose third dimension greatly increases the image size, especially in high resolutions. Three dimensional images are used mostly in biomedical applications [9] [10], but also in geophysics, in industrial quality control and elsewhere. Large images are also met where exceptionally high resolution is required, as in high quality digital photography, topography and geodesy. In the following, the description of the algorithms will be given for the case of 3-D images. However, the algorithms are equally applicable in two or multiple dimensions. In order to avoid confusion by using the terms pixel or voxel, we are going to use the term *point* when referring to a discrete image element.

When processing large images, in most cases it is practically impossible to keep the entire image in the computer memory (RAM). Usually, the image is stored in the permanent storage medium (magnetic disk, WORM, CDROM) and the algorithm reads parts of the image that are needed in the computation. The part of the image that is read each time has to be small enough to fit in available memory. Each point in the image is usually read as many times as the algorithm processes it. However, seek time exceeds read time by orders of magnitude, and therefore it is important to read the data sequentially, when this is possible. Apart from that, naturally it is also important that each point of the image be read as few times as possible.

Most image processing transforms are local and, therefore, can be implemented by sequential algorithms [11]. These algorithms process the image using a predefined scanning order, which can be defined to coincide with the order in which the image is stored on the magnetic disk. Consequently the image may be read from the permanent storage medium in segments that have size equal to the available memory, thus minimizing the delay caused by the seek operations. However, in propagation-based methods for the skeletonization by influence zones and for the watershed transform there is no predefined order for processing the image, thus making any attempt to read the image in blocks pointless. Furthermore, it is pointless to use cache memory, since each image point is usually accessed only once. Therefore, the use of propagation front algorithms in large images presents serious problems.

In the following, we shall present a number of algorithms that solve the above problems by minimizing memory requirements. First, in Section II, the theoretical foundations of the skeletonization by influence zones and the watershed segmentation are briefly reviewed. Then the existing algorithms are described. In Section III, two new memory-efficient algorithms for the skeletonization by influence zones are presented. The first is designed to function on surfaces or other regions of interest, while the second to process complete domains. In Section IV we present a new watershed algorithm that relies on the decomposition of the input image into greylevels in order to greatly reduce memory requirements. All the algorithms mentioned above are given in pseudocode, and their performance is analyzed.

II. Theoretical background and algorithms

A. Basic definitions

Let an elementary basic vector be of the form $(x_1, x_2, \ldots, x_n)$, $x_i \in \{0, 1, -1\}$. A set C of elementary basic vectors defines a *connectivity* in $\mathcal{Z}^n$. A point $\mathbf{p}$ is said to be *adjacent* to another $\mathbf{p}'$ with respect to C, if their vector difference belongs to C [12]. This adjacency is denoted by $\mathbf{p} \in N_C(\mathbf{p}')$. It is worth noting that, in the general case, adjacency is not commutative. In the following, we shall assume that connectivity is chosen in such a way so that it is commutative.

Let D denote an image region $D \subset \mathcal{Z}^n$. A path of length n in D is defined as a sequence of points $(\mathbf{p_0}, \mathbf{p_1}, \ldots, \mathbf{p_n})$, $\mathbf{p_i} \in D$ such that $\mathbf{p}_{i+1}$ is adjacent to $\mathbf{p}_i$. The *discrete geodesic distance* from point $\mathbf{p}$ to point $\mathbf{p}'$ is defined as the minimum length of all paths starting at $\mathbf{p}$ and ending at $\mathbf{p}'$ and will be denoted $\mathbf{pp}'$. Furthermore, the distance from a set $S \subset D$ to a point $\mathbf{p}$ is defined as the minimum of all distances from points of S to $\mathbf{p}$. A set S is *connected* if for all $\mathbf{p}, \mathbf{p}' \in S$, a path exists between $\mathbf{p}$ and $\mathbf{p}'$. We define a *growth* of S within D as the set $S(1) = S \cup G(1)$ where $G(1) = \{\mathbf{p} \in D - S,\ \mathbf{p} \in N_C(\mathbf{p}'),\ \mathbf{p}' \in S\}$. $G(1)$ contains only all points that are adjacent to S. The n-th growth of S, $S(n)$ is the growth of $S(n-1)$. Similarly, $S(n) - S(n-1)$ contains all points that have a distance of n from S. In morphological terms, if the connectivity C is seen as a structuring element, the n-th growth of set S is expressed as $S \oplus nC^s$.

B. Influence zones transform

Let us suppose that there are m sets $S_1, \ldots S_m$, $S_i \in D, S_i \cap S_j, i \neq j$ (called *markers*). We define the *influence zone* of S_i as the set:

$$IZ_D(S_i) = \{\mathbf{p} \in D,\ S_i\mathbf{p} < S_j\mathbf{p},\ \forall j \neq i\}$$

The skeleton by influence zones of D is defined as:

$$SKIZ_D = \{\mathbf{p} \in D : \exists i \neq j,\ S_i\mathbf{p} = S_j\mathbf{p}\ \mathit{and}\ \mathbf{p} \in N_c(IZ_D(S_i)) \cap N_c(IZ(S_j))\}$$

A point $p \in D$ belongs to $IZ_D(S_i)$ if and only if:

$$p \in S_i(n),\ p \notin S_j(m),\ \forall j \neq i,\ \forall m \leq n$$

By defining as $D(n) = \{\mathbf{p} \in D, \exists i : \mathbf{p} \in S_i(n),\ \nexists j :\ \ \mathbf{p} \in S_j(m)\ m < n\}$, it is clear that if $\mathbf{p} \in D(n)$, $\mathbf{p} \in SKIZ_D$ then for all $\mathbf{p}'$ such that $\mathbf{p}' \in N_C(\mathbf{p})$, $\mathbf{p}' \in D(n+1)$ hold that $\mathbf{p}' \in SKIZ_D$. Thus, the influence zones transform can yield very thick skeletons. It is consequently necessary to modify its computation in such a way as to minimize the number of skeleton points without sacrificing consistency. This can be achieved by modifying D, defining $D'(n) = D(n) - SKIZ_D(n)$ and calculating $D(n+1)$ using $D'(n)$. This effectively eliminates the effect described above, without altering the classification of points already in influence zones.

C. Algorithms for influence zone transform using propagation fronts

There are many serial and parallel algorithms for computing the influence zones. The fastest though, are those that use *propagation fronts* (also called grassfires or wavefronts) to process each point in D only once, when a zone growth reaches it. These propagation fronts are implemented by a data structure (queue, list, stack, array) that stores all points in the edge of the influence zone. In the following, we shall refer to that structure as a propagation front, without specifically stating how it is implemented.

These algorithms have the following steps:

1. Label the points of the influence zone of each marker S_i with a unique label in a label image. Labels are employed by all influence zone algorithms reported in the literature.
2. Put the points of each marker in the propagation front.

3. For each top point of each propagation front, find all adjacent points that are not labeled in the label image, put them in the propagation front and label them.
4. Remove each point from a propagation front after it has been processed, then go to step 3.

Region growing in such algorithms can be implemented in two, roughly (but not completely) equivalent ways [12]:
Read formalism: A new point is labeled according to its neighbors' labels.
Write formalism: A point labels all its unlabeled neighbors with its label.

Algorithms also differ in the way they handle region collisions. They either:
- Use the above definitions of influence zones to define the skeleton points.
- Label no skeleton points.
- Define extra skeleton points so that the skeleton is always one point thick and no influence zone is adjacent to another.

However, the last two methods are not *consistent*, because their result depends on the order by which the markers are processed. Thus in the following, the first method will be used.

D. Watershed transform

A discrete grayscale image in domain $D \subset \mathcal{Z}^n$ defined as:

$$f(\mathbf{p}) \in [0, \ldots, n], \ \mathbf{p} \in D$$

can be decomposed into a series of binary images $f_i = \{\mathbf{p} \in D, f(\mathbf{p}) = i\}$ which will be called *greylevels*.

In the continuous case, the watershed transform treats the image as a relief and traces the flow of an inertialess liquid originating in a point $\mathbf{p}$, until it reaches a minimum of the relief. The result of this transformation is the segmentation of the image into *catchment basins*, one for every regional minimum of the image. At this point we need to define a regional minimum as a connected set of points in a specific greylevel surrounded only by points of higher graylevels.

The absence of continuous gradients in digital images makes the computation of the watershed transform less obvious. However, it can be simulated by assuming that the flow from a point in a graylevel is directed towards the closest point belonging to a lower graylevel (according to the discrete geodesic distance within the current greylevel). From there, it continues likewise to the next closest point of a lower greylevel, until it reaches a regional minimum.

By taking the above into account, we define the *flowdown distance* $dist_W(\mathbf{p}, \mathbf{p}')$ between point $\mathbf{p} \in f_i$ and point $\mathbf{p}' \in f_j$ where $i \leq j$ as the length of the non-decreasing path which connects $\mathbf{p}$ to $\mathbf{p}'$ that contains the minimal number of points in the highest graylevel and, progressively, the minimal number in the successively lower graylevels. If such a path does not exist, the flowdown distance is infinite, meaning that it is impossible for the liquid to flow from $\mathbf{p}$ to $\mathbf{p}'$. It must also be noted that the flowdown distance is not commutative. The flowdown distance between a point $\mathbf{p}$ and a set S is defined as the minimum distance between $\mathbf{p}$ and all points in S. If we suppose that there are n regional minima $M_1, \ldots, M_n$ in an image, the catchment basin of a minimum M_i is defined as:

$$CB_i = \{p \in D : \ dist_W(M_i, p) < dist_W(M_j, p), \ \forall j \neq i\}$$

The watershed points of the image are defined as:

$$WSHD = \{p \in D : \exists i, j \ dist_W(M_i, p) = dist(M_j, p)\}$$

It can be shown that the above definition of the watershed transform yields unnecessarily thick watershed regions. This problem can be solved in a similar way as in the case of the influence zones, i.e. by excluding from consideration watershed points close to the minima during the computation of further distances.

E. *Watershed algorithm by immersion*

The computation of the watersheds using the definition in II-D is impossible in practice, since all possible paths would have to be considered. Again, various algorithms have been proposed for the computation of watersheds. The most commonly used is the immersion algorithm, since almost all others are inexact and/or slow [4].

The immersion algorithm works by inverting the process of computing the watersheds. Instead of tracing the flow from the points to be classified to the minima, it 'floods' the image by computing the skeleton influence zones in each graylevel using as markers the minima of the image and the catchment basins computed in previous graylevels. It can be shown that the above procedure yields a result identical to the one that results from the definitions in II-D. Again, the algorithm can handle basin collision in different ways, consistently or not. In the following, we shall implement the watershed transform using consistent watershed point labeling.

III. Influence zones transform in large images

A. *Influence zones transform on surfaces and other regions of interest*

In various applications in image processing, graphics and other fields, it is often necessary to perform a geodesic influence zones transform on a surface. For example, this is the case when computing the Delauney triangulation of a surface [14], based on a previously calculated geodesic influence zones transform (Voronoi tessellation).

However, in the case of the algorithms described in Section II-C, the storage of the enclosing rectangular volume is needed in order to store the surface in RAM. If the surface is not approximately planar, the size of the volume that needs to be stored in memory exceeds the size of the surface by orders of magnitude. Thus, it is common that the volume containing a surface is too large to fit in available memory, although the surface itself is not. Because the transformation is performed on an image region too large to fit in memory, the speed of the classical algorithms drops drastically. The proposed solution to this problem is to store only the surface points, and not the entire volume in RAM. This solution can also be used for skeletonizing arbitrarily shaped regions of interest (ROIs) as well. Details of the proposed method are described in the following.

A.1 Storage of the region of interest

In image processing, the most common method to represent an image or a volume is the bitmap form, that is in the form of a two- or three-dimensional array which contains the image information, whether it is greyvalues or labels. This array is stored in consecutive memory addresses, and the coordinates of each point are determined by the memory position it occupies, and vice versa. However, because of the one-to-one correspondence between memory and image, the shape of a region stored as a bitmap can only be a parallelepiped. Thus, in order to store an arbitrarily shaped region or surface, this classic representation method must be abandoned and alternative ones must sought.

The obvious method for the representation of an arbitrarily shaped point set is to store each point separately in an array, defining it with its coordinates and value. This is the method that is going to be used in the following. The disadvantage of this method is that accessing the value of a point, given its coordinates, requires a search through the entire image region that is stored in memory. The solution that is applied to solve this problem is two sided. Firstly, the minimization of the seek time of a point in the region is attempted. Secondly, ways to limit the search operations of the algorithm as much as possible are sought.

In order to minimize the seek time of a point in a region stored in an array of points in coordinate form is to sort this array according to an ordering of our choice, and then to perform sorted search operations on it. The ordering that is going to be used in the following is the one commonly used in the storage of images. In the three-dimensional case, we sort first by the z-coordinate, then by the y-coordinate and then by the x-coordinate. From now onwards, this ordering will be called the *basic order*. In the following it will be shown that this choice of ordering will be beneficial because of its correspondence to the way images are stored on disk.

A.2 Use of double propagation fronts

The only region access operations performed by the propagation-front based algorithm are those that find the region points adjacent to a propagation front point and, depending on their labels, may add them to the propagation front and label them accordingly. These new points can only be of four types:
1. Points not yet visited by the algorithm.
2. Points visited by the algorithm in the current propagation step.
3. Points visited by the algorithm in the previous propagation step (and thus being in the current propagation fronts).
4. Point visited by the algorithm two propagation steps before (and thus having been in the previous propagation fronts).

Since the size of a propagation front is much smaller than the size of the image region, point searching in it is much faster in the front than in the region. Naturally, this only holds if the points in the propagation front are sorted, making searching with logarithmic computational complexity possible. Thus, if the points in the current propagation fronts are sorted before the propagation begins and each point adjacent to them is sought among them before being sought in the region, significant performance gains can be achieved.

In order to find more quickly points of the 4th type, in addition to those of the 3rd type, the immediately previous propagation fronts must be retained, and points must be searched in them as well as in the current propagation fronts. These previous propagations constitute what we will call the *second* or *backup level* of the propagation fronts. Their importance will be much greater in the front-only algorithm to be presented in Section III-B. The storage of the second level propagation fronts doubles the memory requirements of the propagation fronts. However, since these are small compared to the memory requirements for the storage of the image region, the memory burden is not very significant.

Although it is clear that points belonging in the 1st category must be searched in the region, it might seem possible that points in the 2nd category might need not. However, the points added to the fronts in the current propagation step cannot be efficiently sorted in the basic order, because new points are constantly being put in this category. Therefore, checking points against this category is inefficient because it requires linear searches.

The final method used for the minimization of search operations on the entire image region stores the two levels of the propagation fronts. Thus, each point adjacent to a point that belongs to the current propagation front is first sought in these two levels of the propagation fronts and, if it is not found there, it is sought in the region. On average, half of these adjacent points are found in the propagation fronts and the rest must be searched for both in the region and in the propagation fronts. Thus, the speed of this part of the algorithm is effectively doubled (or, more accurately, the slowdown compared to the original algorithm is halved).

Because of the heavy use of data objects in the algorithm, it is given in object-oriented pseudocode, similar to $C++$. The object types used are described in Figure 1 at the start of this section. The final algorithm is given in Figure 2.

A.3 Simulation Experiments

A test image region was created in order to verify the algorithm experimentally. It consists of a spherical surface having a radius of 127 enclosed in a $256 \times 256 \times 256$ volume. In order to ensure surface connectivity, the sphere thickness was chosen to be greater than one voxel (average thickness $\sqrt{3}$). The number of points in the surface was 346394. It should be noted that, if the entire enclosing volume is stored, the storage requirements would be over 16 million points (i.e. over 25 times the needed memory for storing the points of the surface). In this sphere a number of point markers were randomly inserted, their number ranging from 2 to 65536. The tests were performed on a PC with a Pentium processor at 133 MHz and 16 Mbytes of RAM, running Windows NT.

The speed of the algorithm was satisfactory, being almost constant at 13 to 14 seconds for most of the tests, as can be seen in Figure 3a. It only rises significantly when more than 16192 marker points are used. Above that number, the number of markers becomes comparable to the number of points in the surface, and the entire algorithm degenerates. It must be noted that the execution time remains constant although the number of iterations decreases with the number of markers as can be seen in Figure 3b, meaning that the time spent on each iteration increases with marker number, as expected. The memory requirements were also reasonable. The points needed for the storage of the surface region were naturally constant at 346394, and although the number of points in the propagation fronts increased with marker number they never exceeded the number of points in the region, as can be seen in Figure 3c. In most cases, the average number of points in the fronts remained well over one order of magnitude below the number of region points.

If we assume a uniform distribution of markers on the image region, the number of points belonging to each influence zone can be approximated as $P_{zone} = P_{Region}/n$ where n the number of markers and P_{Region} the constant number of points in the region. If the region is modeled as a volume, the radius of the influence zone is $r = k_1 \sqrt[3]{P_{zone}}$. Therefore the number of points on the front is $P_{front} = k_2 (P_{Region}/n)^{2/3}$. Thus the total number of points in the fronts is $P_{tot} = k_3 \sqrt[3]{n}$. If the region is modeled as a plane, the radius is $r = k_4 \sqrt{P_{zone}}$, the number of points in fronts is $P_{front} = k_5 \sqrt{P_{Region}/n}$ and the total number $P_{tot} = k_6 \sqrt{n}$. Because the region used in this experiment is almost exactly the surface of a sphere, the experimental results for the above quantities should be closer to that of the planar case. Indeed the measurements shown in Figure 3c were modeled by the least-squares method to be as follows:

- The maximum number of points stored in propagation fronts during the execution of the algorithm is modelled by $\max(P_{tot}) = kn^{0.47}$.
- The average number of points stored in propagation fronts during execution was modelled by $\mathrm{avg}(P_{tot}) = kn^{0,40}$.

As expected, the number of propagation steps, as shown in Figure 3b was modelled as $\max(r) = kn^{-0,42}$.

B. Influence zones transform without region storage

The algorithm presented in Section III-A.1 is very efficient for performing the transform on regions of interest whose enclosing volume is a large image. However, it is of no help in the case that this region *itself* cannot fit in available random access memory. In this case, the need to store the influence zone labels leads to serious performance problems. The only way to avoid these problems is to avoid the storage of the image region in memory, store only the propagation fronts and compute the transform solely with operations on them.

Unfortunately, the image is not only used to store the result of the segmentation, but also to provide a reference to enable the determination of the status of each point. Without it, there is no simple way to know whether a point belongs to the skeleton or to an influence zone (and to which) or to neither. The influence zone algorithm requires that each point adjacent to a propagation-front point is labeled as an influence zone or skeleton point and, accordingly, is appended to the propagation fronts or not. Thus, we have to find some other way to determine whether such an adjacent point has already been visited by the propagation fronts, if it is a new propagation front point, or if it is a collision point that belongs to the skeleton. The methods that were used to achieve this objective, as well as a discussion of ways to retrieve the result of the transform, are presented in the following.

B.1 Determining whether points are new

The first step towards determining the status of a point adjacent to a propagation front is to establish whether it has already been visited by that propagation front. As described in Section III-A.2, a way to achieve this is to use double (or backed-up) propagation fronts. Because the maintenance and operation of the double fronts is inexpensive in both memory and computation terms, and because they do not require image reference operations, they are a natural choice for this task.

However, the double fronts cannot determine whether a point has been visited by *another* propagation front, or whether it is the point where two propagation fronts collide. Determining the first would require a search for the new point in all the propagation fronts, while determining the second case would require checking the new point against all other new points. This means either that $2n$ logarithmic search operations would be required for each new point (where n the number of markers and therefore of the propagation fronts), or that all propagation fronts should be merged and search operations be performed on them. Both options are clearly unacceptable, thus in the following section we present a solution to this problem.

B.2 Determining collisions

Having determined that a new point has not been already visited by its respective propagation front, an efficient way must be found to check whether it appears in any other new point set or propagation front.

In order to achieve this, a data structure called a *minimum heap* is used [13]. The heap's function is to extract quickly the minimum element it contains, and to accept new elements in any order. The computational cost for the extraction of the minimum of the heap is $O(\log(i))$, and the cost for the insertion of an arbitrary element is also $O(\log(i))$, where i is the number of elements in the heap.

The heap is used to receive the points from the propagation fronts and the sets of new points that are to be appended to each of them. Only one point from each front needs to be kept in the heap at any time. Thus the size of the heap is $2n$ points and the computational cost of the insertion $O(\log(n))$, where n is the number of markers. This is because both the fronts and the new point sets are sorted on the basic order beforehand. Thus, if their first points are inserted into the heap, the next point extracted from the heap will be the minimum of all the points left in them. It must be noted that the second level of the propagation fronts is unnecessary at this stage.

Thus, by successively extracting points from the heap, and by then inserting into the heap the next point from the front or the new point set that the extracted point originated from, we can guarantee to receive all the points in the basic order. Duplicate points will be in succession and can be identified as such. If there is a number of identical points from different new point sets, and there is no point from a propagation front among them, it means that this point is a skeleton point. In this case, all the points are removed from their respective sets (so that the propagation will not continue from them), and appended to the set of skeleton points. If there is a point originating from a propagation front among a number of identical points, this point takes precedence and all other points (which originate from new point sets) are removed from their respective sets. This is because the propagation front point originates from an earlier propagation step than the others. It is impossible to have two identical points from two different propagation fronts, since they would have been labeled as skeleton points in the previous propagation step.

Thus, the determination of the collisions between fronts can be achieved with only $O(\log(n))$ computational complexity per point.

B.3 Skeleton maintenance

In order to have a proper computation of the influence zones transform, the skeleton points must be properly computed. Because the propagation step does not continue through skeleton points, and because of the geodesic nature of the transform, all skeleton points need to be kept in order to ensure the correct propagation of the fronts. Also, the skeleton points that have been found in the previous propagation step need to be accessible in the basic order so that they may be properly inserted into the heap for comparison.

In order to achieve the goals mentioned above the skeleton points found by using the heap at the end of each propagation step are temporarily kept in a list. When the heap has processed all points, this list is sorted and merged into an array containing all skeleton points found in previous propagation steps. This array is then used as an input to the heap, in order to determine further collisions.

B.4 Saving the result

The fact the image is not kept in memory means that the result, which consists of the labels identifying whether a point belongs to a specific influence zone or to the skeleton, is not explicitly available. However, it is possible to store the result of the transformation to disk, as the algorithm runs. This, for reasons of efficiency and consistency, will need to happen once for each point, and only after that point has received its final label.

This is best done at the end of each propagation step, when all points pass through the heap and receive their final labels. At that time, the points that have been reached in the current propagation step are written, in coordinate form, to an intermediate disk file corresponding to their z-coordinate. Because the points are coming out of the heap in the basic order (which was chosen to be based first on their z-coordinate), they are written to disk continuously and, therefore, without delays. After the end of the algorithm, these files are read and the points they contain are written to their final positions in the output file, which is in bitmap format.

B.5 Memory requirements and pseudocode

According to the previous analysis, it can be seen that the total memory requirements of the algorithm are only those of the propagation fronts and the edges. Even if we consider the fact that the propagation fronts, being double, consume twice as much memory as in the classical algorithms, these memory requirements are very small. However, these requirements increase with marker density, since this results in a corresponding increase in the density of the propagation fronts.

The algorithm is presented in the form of object-oriented pseudocode in Figure 4. The data structures it contains are described in Figure 1.

B.6 Simulation Experiments

The fronts-only, which is described in Figure 4, algorithm was used to perform the influence zone transform in a series of experimental cases. In the experiments the volume to be processed had a size of $256 \times 256 \times 256$ and contained a number of markers which ranged from 2 to 16385, and which were randomly distributed on the volume. The machine used to perform the tests was a personal computer with an Intel Pentium processor clocked at 90 MHz, 64 Mbytes of random access memory, running Miscrosoft Windows NT Server.

As can be seen in Figure 5a, the processing time of the algorithm rises only slightly with the increase of the number of markers when they are a few (under 1024). It rises much faster when they exceed that number. This happens because, above this number, the influence of the stack becomes the determining factor in the performance of the algorithm, and the computational cost of the operation of the stack is directly related to the number of influence zones (since its size is equal to that number). Execution time of the algorithm ranged from 35 minutes to an hour, which is reasonable considering that the classical algorithms cannot operate under such memory constrains. The number of propagation steps as can be seen in Figure 5b decreases greatly with increasing marker numbers, since each influence zone becomes smaller. As expected, the memory requirements were proportional to the number of markers, that is to the number of influence zones, as shown in Figure 5c. These requirements were on average low, as expected. They come within the order of magnitude of the memory requirements of the volume which was being processed only when there were over 16382 influence zones.

IV. Decomposition watershed algorithm

In order to implement the watershed transform in large images, a way must be found to perform the transformation on a part of the image at a time. The obvious way to achieve this is to take advantage of the fact that the algorithm processes the greylevels of the image sequentially, from the lowest to the highest. After decomposing the image into its constituent greylevels, each one could be processed separately. Thus, only a small part of the image must be kept in memory at a time. The loading of this part can be done in one step, without the need for unnecessary disk seek operations.

As was seen when examining the influence zone transform, an image region that does not have a rectangular shape cannot be stored as a matrix. The same approach that was used there will also be used here as well. In essence, considering that the computation of the watersheds by immersion consists of performing successive influence zone transformations in each greylevel, we use the region storage algorithm that was described in Section III-A.1. The configuration of the propagation fronts that are used for the computation of the influence zones will be the same, as will be their handling. It must be noted that the *basic order* that is used in this algorithm (see Section III), is chosen to be the same as the order in which the image is stored on the disk. Furthermore, in the case of the watershed transform, all the sets of points used will be sorted in that order.

However, there are some issues in the watershed transform by immersion which require some attention. The first is finding the image minima in each greylevel, from which the propagation fronts emanate. The second is the transition of the propagation fronts from one greylevel to the next, without the storage of unnecessary points. In the following, we shall discuss these issues, as well as the procedure for the decomposition of the image into its constituent greylevels and its recomposition from them. We shall also describe the modifications that need to be made to our algorithm for performing *marker*-based image segmentation.

A. Image decomposition

In the decomposition watershed algorithm, the image must be decomposed into separate greylevels. These greylevels need to be stored on disk, each on a separate file so that they can be read independently from each other as the algorithm progresses. At the end of the algorithm, these files will must be merged to give the reconstructed output image.

The decomposition of the image into greylevels poses no particular problems. Since the basic order that is used in the algorithm is the same as the order in which the image is stored on disk, the only thing that needs to be done is to read the input image serially and write each point (in coordinate form) to the file that corresponds to its greyvalue. In order to avoid having too many files open at the same time and also to avoid needless disk seek operations, we use buffers, one for each file that corresponds to one greylevel. Points are first written to these buffers. When one of them is full its points are written to disk all at the same time. Because this preprocessing happens before the beginning of the actual watershed transform, the buffers can use all available memory.

B. Output image reconstruction

The reconstruction of the output image is performed in two phases, one during the execution of the algorithm and one after its end. After each greylevel is processed, its points are written (again as coordinates) to intermediate files that correspond to the z-planes of the image (the planes with a constant z coordinate). Because the basic ordering of the image is primarily according to the z coordinate, the disk write head needs to move only as many times as the size of the image along the z-coordinate. When the algorithm finishes, the intermediate files are read again and the points they contain are written at their final position in the output file (which, like the input file, is in bitmap format).

From the above analysis it can be seen that, the algorithm requires reading/writing $3+3+1 = 7$ bytes per image point, since we need one byte per coordinate. This is not satisfactory and a simple method for the compression of the point coordinates was used in order to reduce the number of bytes that are read/written. In this method, the coordinate difference between successive points is stored, if it is within certain limits, otherwise the coordinates themselves are stored. The gain is that the difference between points is calculated in such a manner, as to fit within a single byte. In more detail, the steps of this method are the following:
1. Find the difference of the coordinates between the current point (x_1, y_1, z_1) and the previous one (x_0, y_0, z_0).
2. If $123 \geq x_1 - x_0 \geq -1$, $y1 - y_0 \in \{0, 1\}$ and $z_1 = z_0$, the difference of the y coordinates is stored in the most significant bit, and the difference of the x coordinates is stored in the 7 least significant bits of the output byte.
3. Otherwise, the output consists of a zero-valued byte and the three coordinate bytes.

Since the points that are being written are sorted first on z, then on y and finally on x, only one byte is stored in most cases. In practice, 1.2 bytes are required on average per point (the compression ratio is 1:2.5). Thus, in the end, 3.4 byte readings/writings per point are required by the algorithm.

C. Propagation front maintenance

In the classic watershed-by-immersion algorithms, the transition of the propagation fronts from one greylevel to the next does not present any problems. Since all points, irrespective of greyvalue, are directly accessible, it suffices to check whether a point is adjacent to another of lower greyvalue, in order to determine that the propagation front continues from this point. In the decomposition watershed algorithm we can only access the points of the current greylevel. Thus, in order to perform the transition, we must keep, at the current greylevel and for every propagation front, those points that are adjacent to other points that have a higher greylevel. Because the fronts propagate into all the subsequent greylevels and not only into the next one, such points must be kept in storage until they cease to be adjacent to points of higher greylevels. However, at the same time, the storage of points that are only adjacent to lower greylevels must be avoided for memory minimization. In order to achieve the above objectives, a rather complex data structure is used, consisting of a dynamically managed array of points which also contain information about the position of adjacent points belonging to higher greylevels. For the sake of brevity, this structure will be called an *interfront* (from inter-greylevel propagation front).

The basic functions that must be performed on the interfront are the following:
- Storage of those points of the currently processed greylevel that are adjacent to points belonging to a higher greylevel. These are easily found by checking if the neighboring points of every point that a propagation front goes through, belong to the current greylevel (which is something that the algorithm does anyway). If they do not belong to one, we can conclude that they belong to a higher greylevel and subsequently store the current point in the interfront.
- Reading all the stored points, which will initiate the propagation fronts in the next greylevel. For this initiation, at each new greylevel, every point in the interfront must be checked for adjacency with any points in that greylevel. If such points exist, then they are appended to the propagation front, while the interfront point adjacent to them is appended to the backup level of the propagation front. It must be noted that point reading from the interfront occurs only once per greylevel, that is at the initiation of the propagation.

• Removal of a point from the interfront when it ceases to be adjacent to points of a higher greylevel than that which the algorithm has reached.

In order to avoid keeping more points in the interfront than needed, the position of all adjacent points that the algorithm has not visited are stored for every interfront point (in a fashion similar to [17]). This has two advantages:

• First, that a point can be removed from the interfront when it no longer has such adjacent points.

• Second, at each greylevel we only need to search for points adjacent to an interfront point in the directions we know they exist (which, owing to the size of the interfront, leads to a considerable acceleration of the algorithm).

In order to save memory, the position of these adjacent points is stored as flags (single bits). Each direction of the connectivity corresponds to one such flag, which is set to 1 if there is a point in that direction, or to 0 if there is not. If the algorithm works in 6-connectivity, six flags will be needed and a single byte will suffice for their storage. As mentioned, a point is added to the interfront if some points in its neighborhood are not in the current greylevel. As it is being added, a label is appended to it which contains, in coordinate form, the directions in which no points were found. Thus, when the propagation at the next greylevel starts, the directions where adjacent points may exist are read for each interfront point, and the current greylevel is searched to find these points. If they exist in the current greylevel, it is certain that no other points will exist in this direction, and the corresponding flag in the label byte is cleared. When all flags are cleared, this point is no longer useful and is removed from the interfront.

Because the propagation in each greylevel begins from only a few interfront points, the interfront contains significantly fewer points than the greylevel (considering the fact that the sizes of different greylevels vary significantly). Thus, the greater part of the algorithm's memory requirements is due to the interfront. This means that the reduction of its size is desirable. In order to achieve this, we take advantage of the fact that the addition and readout of points happens separately and consecutively in each greylevel. We also take advantage of the fact that the reading of points takes place in an arbitrary order which can be chosen to be the basic order. The above lead to the compression of the interfront points, which can produce a significant memory gain. The compression is performed approximately as in the writing of the greylevels to disk, that is by storing the difference between consecutive points in a single byte. Since in this case the label must also be stored, the compression ratio is 0.55.

The above compression takes place at the end of the processing of each greylevel, since the number of points added to the interfront during this processing is negligible compared to the total number of interfront points. Because the number of points added to the interfront at each greylevel is not known in advance, appending points to it is implemented using a dynamically managed list. In order to save memory, each point of the list is not a point but an array of many points (in practice 1024 were used). The compression of these new points is performed as this list is merged with the compressed array containing the previous points. All the above are illustrated in Figure 7.

Most of the memory requirements of the algorithm are due to the interfront. This is because it encloses all the area processed by the algorithm, while the current greyscale only adjoins part of it. However, since we took care to compress the interfront, the memory consumed by much smaller than the entire region. If the greyvalue distribution of the image is particularly uneven, the current greylevel can also consume a significant amount of memory. Since the propagation fronts in the current greylevel are much smaller than the greylevel itself, their memory burden is negligible.

The overall decomposition algorithm is given in Figure 8, and the routine that processes each greylevel is given in Figure 9. Again, they are given in object-oriented pseudocode and the structures are described in Figure 1.

D. Marker-based watershed transform

The basic watershed transform frequently presents significant oversegmentation problems when used for image segmentation. To overcome this problem, *markers* are used to define the desired catchment basins [4]. These markers (or seeds) are the only points from where propagation fronts start, and no other fronts are started at the regional minima of the image. In the full image storage algorithms this means in practice that the propagation may also take place toward greylevels lower than the current (but not toward higher ones).

Naturally, the decomposition watershed algorithm does not have access to greylevels other than the one currently being processed. To overcome this limitation the algorithm needs, at each greylevel, to be granted access to all points of previous greylevels that do not belong to catchment basins. This is achieved by scanning each greylevel once after it has been processed, searching for points that have not been labeled as belonging to a catchment basin or to the watersheds. These points are stored in an list, which is subsequently merged with the next greylevel. The merging has a low computational cost because both the current greylevel and the above list are already sorted in the basic order. It should be explained that the list is also on the basic order because it was computed as a part of the previous greylevel using a basic order scan. Because the current list is merged with the current greylevel and from this merged greylevel the next list is derived, each list contains all the non-processed points not only from the current greylevel but also from the previous ones. After the merging, the resulting greylevel is processed as in the regular partial storage algorithm, except of course from finding regional minima and starting new catchment basins.

E. Simulation experiments

In order to verify the algorithm experimentally, we applied it to perform the segmentation of a three-dimensional biomedical image. The volume used was a series of slice CT scans depicted a human head and had a size of $256 \times 256 \times 256$, part of which is shown in Figure 10a. The purpose of this procedure was to segment as accurately as possible the skull and the two lobes of the brain, as well as the background.

As is common practice when using the watershed transform for image segmentation, an edge detection operation was first applied on the input volume. In this case, Sobel edge detection was used, as shown in Figure 10b. We performed a histogram equalization on the edge image in order to equalize its greylevels. Then, markers were defined interactively for identifying the objects of interest, and the marker-based watershed transform (described in IV-D) was performed. One marker was defined for each lobe of the brain, two for the skull, three for the region between brain and skull, and one for the background. The machine used to compute the segmentation was a personal computer with an Intel Pentium processor clocked at 133 MHz and 32 Mbytes of random access memory running Microsoft Windows NT Server. Some slices of the segmented volume are given in Figure 10c. The segmented skull is shown in Figure 11 and a rendering of all segmented regions in Figure 12.

The segmentation was correct and its quality satisfactory, since the segmented lobes and skull got the shape they have in reality. There was some distortion in the area of the skull, at the top of the head and at its bottom near the face. This was expected, since the edges in these areas were particularly weak and has nothing to do with the correctness of the algorithm. Naturally, what is of importance is the efficiency of the algorithm in terms of computational cost and memory requirements.

As far as computational cost is concerned, the execution time of the algorithm was on average 2 hours and 25 minutes. If we consider the fact that Vincent's algorithm [4] executes the same transform in 30 minutes (if of course the necessary memory of over 64 Mbytes is available), it can be understood that the execution time is very reasonable. As for the memory requirements, these stem mostly from the interfront since the memory requirements of the propagation fronts are lower by an order of magnitude. The maximum number of points stored in the interfront during the execution of the algorithm never exceeded $1,200,000$, and thus memory requirements are much lower than in the traditional algorithms.

V. Conclusions

We have presented three algorithms to reduce greatly the memory requirements of the influence zone and watershed algorithms, thus making them computable on large images. The first of these algorithms which computes the skeletonization by influence zones on surfaces, was demonstrated to have in satisfactory speed and to achieve very large memory savings. The performance of the second algorithm, which uses only the propagation fronts to perform the transform still achieves significant memory gains with a reasonable slowdown. Because of its complexity, the watershed algorithm is the slowest of the three. However it achieves large memory savings.

References

[1] A. Rosenfeld and J. Pfalz, "Distance Functions on Digital Pictures", *Pattern Recognition*, vol. 1, no. 1, pp. 33–61, 1968.

[2] C. Lantuejoul and F. Maisonneuve, "Geodesic Methods in Quantitative Image Analysis" *Pattern Recognition*, vol. 17, no. 2, pp. 177–187, 1984.

[3] F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

[4] L. Vincent and P. Soille, "Watershed in Digital Spaces: An Efficient Based on Immersion Simulations" *IEEE Trans. Pattern Anal. Machine Intel.*, vol. 13, no. 6 pp. 583–598, June 1991.

[5] P. Soille and M. Ansoult "Automated Basin Delineation from DEMs using Mathematical Morphology," *Signal Processing*, vol. 20, pp. 171–182, 1990.

[6] S. Beucher and F. Meyer, "The Morphological Approach to Segmentation: The Watershed Transformation", *Math. Morph. in Image Proc.*, E.R. Doherty, Ed., Marcel Dekker, pp. 433–481, 1993.

[7] G. Borgefors "Distance Transformations in Digital Images" *Computer Vision, Graphics, Image Proc.*, vol. 34, pp. 344–371, 1986.

[8] I. Pitas, "Performance Analysis and Parallel Implementation of Voronoi Tessellation Algorithms Based on Mathematical Morphology", *IEEE Pattern Anal. Machine Intel.*, submitted for publication, April 1996.

[9] I.E. Pratikakis, H. Sahli and J. Corvelis, "Watershed Analysis and Relaxation Labeling", *Math. Morph. and Appl. to Image and Signal Proc.*, P. Maragos *et al.*, Eds. Kluwer, pp. 425–432, 1996.

[10] J. Madrid and N. Ezquerra, "Automatic 3-Dimensional Segmentation of MR Brain Tissue Using Filters by Reconstruction", *Math. Morph. and Appl. to Image and Signal Proc.*, P. Maragos *et al.*, Eds. Kluwer, pp. 417–424, 1996.

[11] I. Pitas, *Digital Image Processing Algorithms*, Prentice Hall, 1993.

[12] B.J.H. Verwer, P.W. Verbeek and S.T. Dekker, "An Efficient Uniform Cost Algorithm Applied to Distance Transforms", *IEEE Trans. Pattern Anal. Machine Intel.*, vol. 11, no. 4, pp. 425–429, April 1989.

[13] M. Atkinson, J. Sack, N. Santoro and T. Strotthotte, "Max-Min Heaps and Generalized Priority Queues", *Communications of ACM*, vol. 29, no. 10, pp.996–1000, 1986.

[14] L. Vincent, "Graphs and Mathematical Morphology", *Signal Processing*, vol 16, pp. 365–388, 1989.

[15] R. van den Boomgard and A. Smeulders, "The Morphological Structure of Images: The Differential Equations of Morphological Scale-Space", *IEEE Pattern Anal. Machine Intel.*, vol. 16, no. 11, pp. 1101–1114, Nov. 1994.

[16] P. Salembier and A. Oliveras, "Practical Extensions of Connected Operations", *Math. Morph. and Appl. to Image and Signal Proc.*, P. Maragos *et al.*, Eds. Kluwer, pp. 97–118, 1996.

[17] L. Vincent "Morphological Transformations of Binary Images with Arbitrary Structuring Elements" *Signal Processing*, vol. 22, pp. 3–23, 1991.

Constants:

`UNLABELED,SKEL,WSHD,TRUE,FALSE`

Structures used and operations available :

`point`	coordinates of a point
-ordering operators $(=, >, <)$	
`label_point`	coordinates of a point and associated label
-ordering operators $(=, >, <)$	
-label write and read	
`queue`	FIFO queue containing `points`
`-read()`	get a `point` from the start of the queue
`-write(point)`	put a `point` in the start of the queue
`-find(point)`	check if a `point` is in the queue
`-empty()`	check if queue has no more `points` in it
`-prune()`	remove unused capacity from queue
`-sort()`	sort all `points` in queue removing duplicates
`block_list`	list of blocks of `points`
`-write(point)`	write point to list
`-read_all()`	read all points from list
`heap`	minimum heap of set maximal size
`-insert(label_point)`	insert a labeled point in the right place on the heap
`-minimum()`	extract the minimum point of the heap
`-empty()`	`TRUE` if no more points in the heap
`dynarray`	combination of list and sorted array
`-read()`	read point from array
`-write(point)`	write `point` to list
`-condense()`	transfer all points from list to array and sort them
`-done()`	`TRUE` if all points in array are read
`vec_dyn_array`	`dyn_array` with adjacency information in each point
`-write(point,int)`	put a labeled point in the list
`-read()`	read labeled point from array
`-remove_label(int)`	reset specific direction in label of last point read
`-condense()`	transfer all points from list to array sorting them, and delete array points having no direction indicated in their labels

External functions:

search(point)	find a point in an array of points, returns pointer
`domainbounds(point)`	returns `TRUE` if `point` within domain

Figure 1. Data structures used in the pseudocode that is used to present the algorithms.

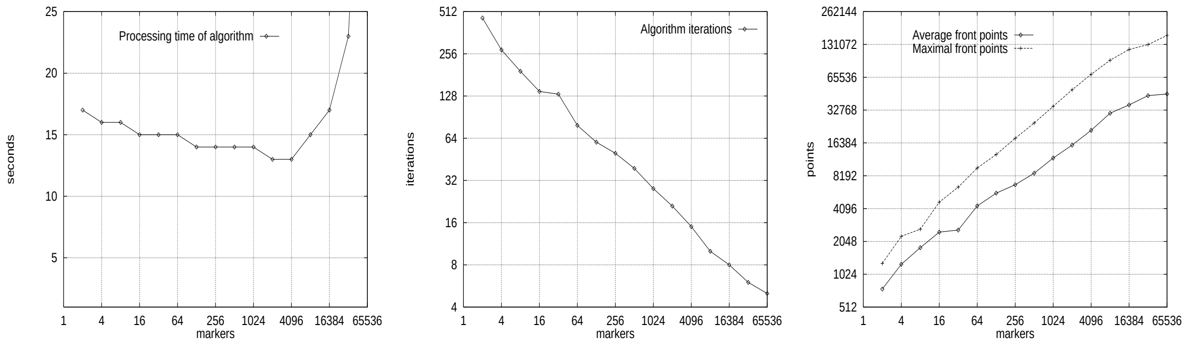
Variables used:

`label_point image[]`	an array of `points` containing the ROI
`integer sign`	flag used to discriminate between points labeled in each step
`point pt,npt`	current point and point adjacent to it
`label_point` pointer `ipt,inpt`	pointers to corresponding points in `image`
`queue front[]`	array of fronts, each corresponding to one marker
`queue oldfront[]`	array of backup fronts
`queue new`	used to hold new points

Algorithm

```
for all markers
    queue[i].write(marker)
sign=1
repeat
    sign=sign*−1
    for all markers
        pt ← front[i].read()
        if ipt ← search(image,pt) exists
            if ipt.label ≠ SKEL
                for all npt adjacent to pt
                    if front[i].find(npt) = 0 and oldfront[i].find(npt) = 0
                        inpt ← search(image,npt)
                        if inpt.label = UNLABELED
                            npt.label ← i*sign
                            new.write(npt)
                        else if inpt.label*sign<0
                            inpt.label ← SKEL
        new.prune()
        new.sort()
        deallocate oldfront[i]
        rename front[i] as oldfront[i]
        rename new as front[i]
until all front[i].empty()
```

Figure 2. The influence zones transform on surfaces algorithm given in pseudocode form.



(a) (b) (c)

Figure 3. (a) Processing time for the region storage influence zone algorithm. (b) Maximum number of iterations for the region storage influence zone algorithm. (c) Maximum and average number of points in propagation fronts for the region storage influence zone algorithm.

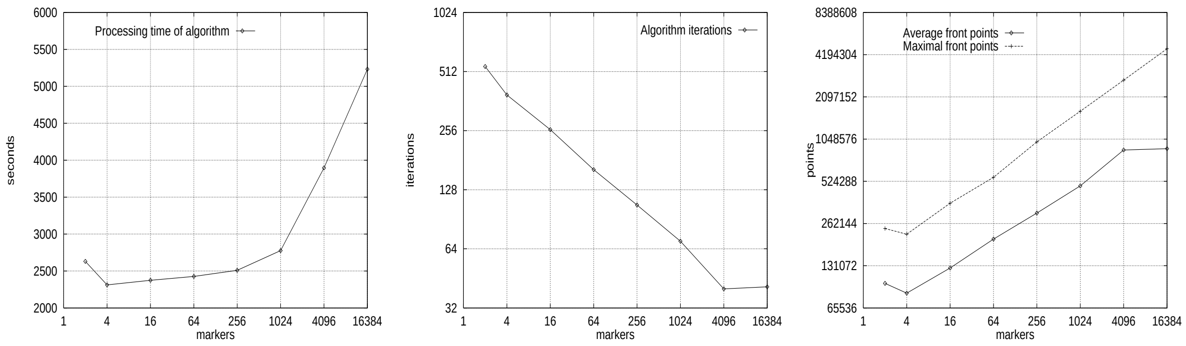
Variables used:

`point pt,npt`	current point and point adjacent to it
`queue front[]`	array of fronts, each corresponding to one marker
`queue oldfront[]`	array of backup fronts
`queue new`	used to hold new points
`heap comparator`	heap used to find duplicates
`label_point buffer[]`	array temporarily holding duplicate points
`dynarray skeleton`	holds the skeleton by influence zones

Algorithm

```
repeat
    for all markers
        pt ← front[i].read()
            for all npt adjacent to pt
                if front[i].find(npt) = FALSE and oldfront[i].find(npt) = FALSE
                    if regionbounds(npt) = TRUE
                        new.write(npt)
        new.prune()
        new.sort()
        deallocate oldfront[i]
        rename front[i] to oldfront[i]
        rename new to front[i]
    for all markers
        comparator.insert(front[i].read(),i)
        comparator.insert(front[i].read(),i+markernumber)
    comparator.insert(skeleton.read(),2*markernumber)
    repeat
        lastpt ← currpt
        currpt ← comparator.minimum()
        if lastpt = currpt
            if no points in buffer[]
                buffer[] ← lastpt
            buffer[] ← currpt
        else if points in buffer[]
            for all points in buffer[]
                if buffer[j].label indicates a skeleton point
                    set skelflag
                if buffer[j].label indicates a front
                    front[buffer[j].label].delete(buffer[j])
            if skelflag not set
                skeleton.add(buffer[j])
    until comparator.empty = TRUE
    comparator.condense()
until all front[i].empty() = TRUE
```

Figure 4. Algorithm for the computation of the fronts-only influence zone transform, given in pseudocode.



(a) (b) (c)

Figure 5. (a) Processing time for the fronts-only influence zone algorithm. (b) Maximum number of iterations for the fronts-only influence zone algorithm. (c) Maximum and average number of points in propagation fronts for the fronts-only influence zone algorithm.

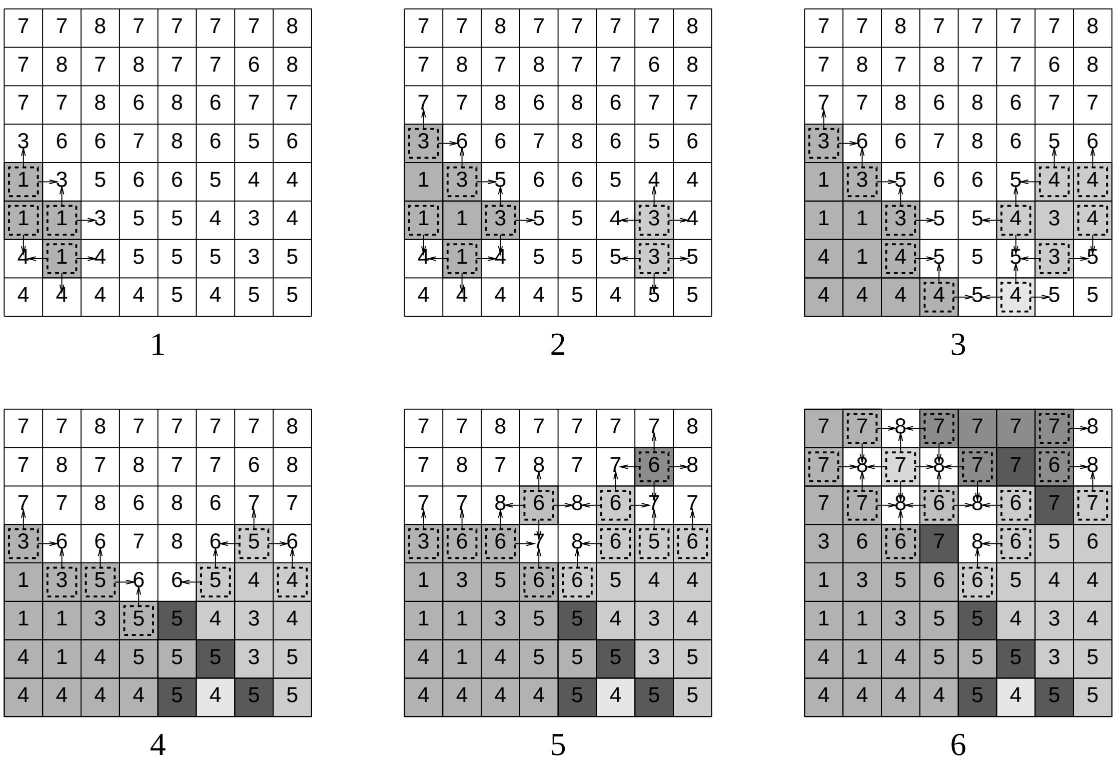


Figure 6. Example of the operation of the interfront. The various lightly shaded squares are points belonging to catchment basins, the dark squares are watershed points, the dashed borders indicate points belonging to the interfronts, and the arrows show the directions stored in the flags of the labels of the intergrey points.

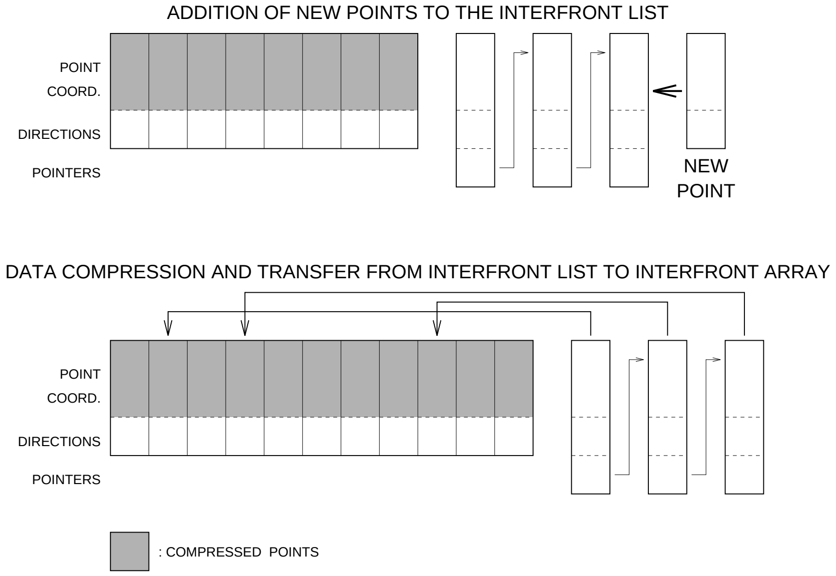


Figure 7. Overview of the structure of the interfront.

Variables used:

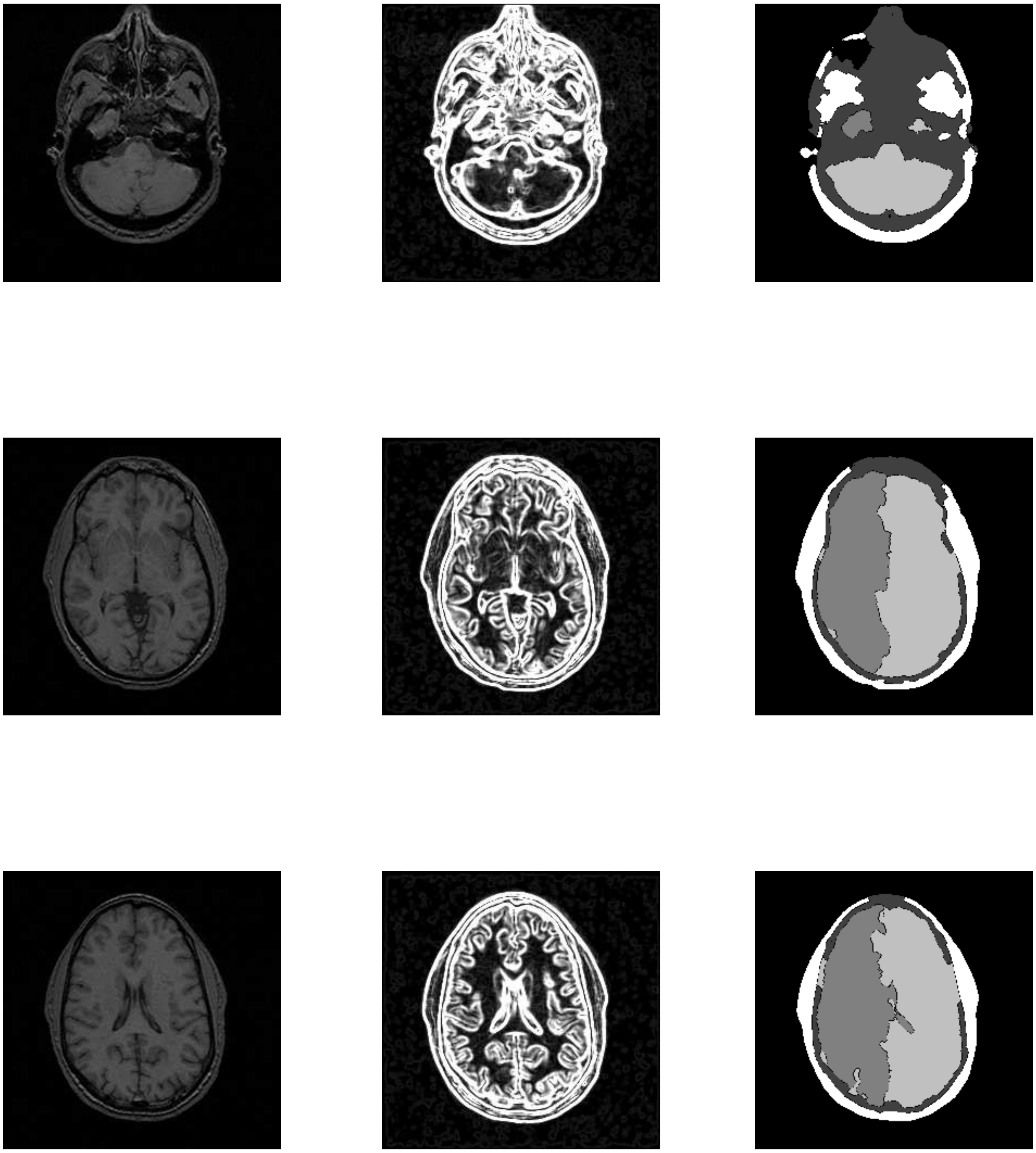
`point image[]`	an array of `points` containing the RoI
`integer sign`	flag used to discriminate between points labeled in each step
`point pt,npt`	current point and point adjacent to it
`point` pointer `ipt,inpt`	pointers to corresponding points in `image`
`queue front[]`	array of fronts, each corresponding to one marker
`queue oldfront[]`	array of backup fronts
`queue new`	used to hold new points
`label_point greylevel[]`	array holding current greylevel
`vec_dyn_array interfront`	propagation front between greylevels
`file greyfile`	temporary storage for greylevels
`file outfile`	output file
`int temp_label`	holds directions where no adjacent points exist

Algorithm
separate image into `greyfiles`
for all greylevels
    `greyfile.readgreylevel[k]`
    merge `intergrey` with `greylevel[k]`
    skeletonize()
    for all points in `greylevel[k]`
        if `pt.label = UNLABELED skeletonize()` with `pt` as only marker
    write `greylevel` to `outfile`
    deallocate `greylevel`

Figure 8. General structure of the decomposition watershed algorithm given in pseudocode.

```
Skeletonize()
    sign=1
    for all markers
        while pt ← interfront.read() possible
            ipt ← search(image,pt)
                if ipt.label ≠ SKEL
                    for all npt adjacent to pt in the directions specified by its label
                        if inpt ← search(greylevel[k],npt) exists
                            if inpt.label = UNLABELED
                                npt.label ← i*sign
                                front[i].write(npt)
                            else if inpt.label*sign<0
                                inpt.label ← WSHD
                            interfront.remove_label() for direction corresponding to npt
            if there were points adjacent to pt
                oldfront[i].write(pt)
        front[i].prune()
        front[i].sort()
        oldfront[i].prune()
        oldfront[i].sort()
repeat
        sign=sign*−1
        for all markers
            pt ← front[i].read()
            ipt ← search(image,pt)
                if ipt.label ≠ SKEL
                    for all npt adjacent to pt
                        if front[i].find(npt) = 0 and oldfront[i].find(npt) = 0
                            inpt ← search(greylevel[k],npt)
                            if inpt.label = UNLABELED
                                npt.label ← i*sign
                                new[i].write(npt)
                            else if inpt.label*sign<0
                                inpt.label ← WSHD
                        else set direction corresponding to npt on temp_label
                    interfront.write(pt,temp_label)
            new.prune()
            new.sort()
            deallocate oldfront[i]
            rename front[i] as oldfront[i]
            rename new as front[i]
    until all front[i].empty()
    deallocate all fronts and oldfronts
```

Figure 9. The processing of each greylevel in the decomposition watershed algorithm, given in pseudocode.



(a) (b) (c)

Figure 10. Sections of the head volume used for testing the algorithm. On the left is the original image (a), in the center the edge image (b) and on the right the segmented image (c)

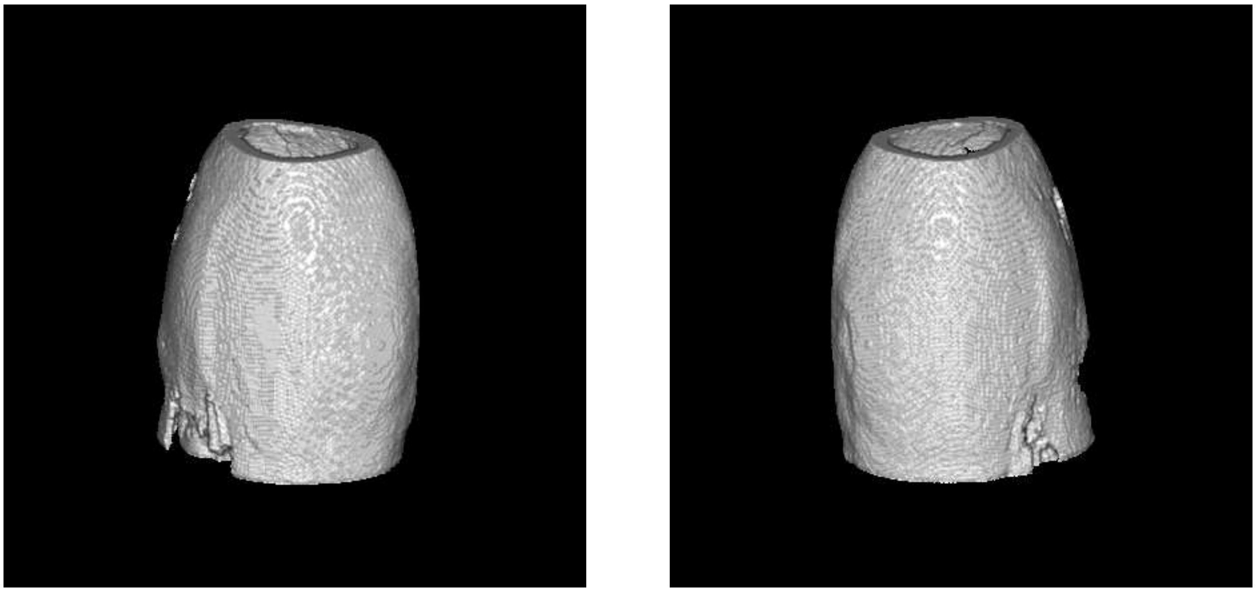


Figure 11. The skull, as it was segmented by the algorithm.

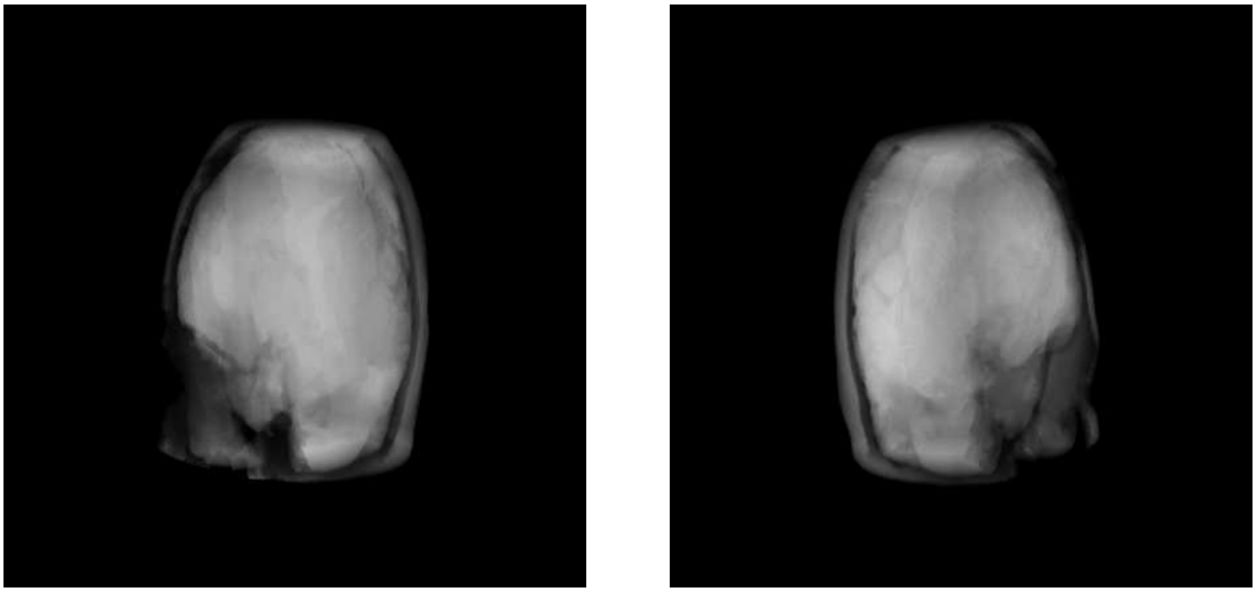


Figure 12. The segmented brain with the various regions shown in transparency.