View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at University of West Bohemia

RaFSi – A Fast Watershed Algorithm Based on Rainfalling Simulation

Stanislav L. Stoev

Computer Science Department,
University of Tübingen,
WSI/GRIS, Auf der Morgenstelle 10, C9,
72076 Tübingen, Germany
sstoev@gris.uni-tuebingen.de

ABSTRACT

In this paper, we present a fast watershed algorithm based on the rainfalling simulation. We present the various techniques and data structures utilized in our approach. Throughout this work, the processing of large data sets (images as well as volume data) is especially emphasized. The results' correctness, the fast execution time, and the memory requirements are discussed in detail.

First we introduce a sequential algorithm and discuss the cases, where the known algorithm produces erroneous results. Afterwards, the presented watershed algorithm is compared with immersion based watershed algorithms with respect to running time and memory requirements.

Keywords: watershed transformation, rainfalling simulation, image segmentation, geodesic reconstruction, watersheds.

1 Introduction and Related Work

In the past years the watershed transformation has proven to be a very useful and powerful tool for morphological image segmentation. The first algorithmic approaches originate naturally from the field of topography [Band86]. Since then, the watershed transformation is becoming more and more popular in different science areas like biomedical, medical image processing [Higgi93], computer vision [Bilod94] etc. The idea of the watershed construction is quite simple. A gray-scale picture is considered as a topographic relief. Every pixel of this digital image is assigned to the catchment basin of a regional minimum. This defines the influence zones of each of the pre-determined regional minima. The watershed lines are now defined as the lines separating influence zones from each other (as depicted in Figure 3).

Numerous techniques for computing the watersheds have been introduced during the past years. The first who proposed an immersion based watershed algorithm are Beucher and Lantuéjoul [Beuch79]. In [Meyer94] and [Beuch93] couple of techniques and algorithms related to the problem of watershed computing are described. Furthermore, Meyer defines in [Meyer94] the watershed transformation in the continuous and in the digital space in terms of a distance function, called topographic distance. One of the classical algorithms for computing the watershed transformation for a gray-scale image is also found in this work. The common strategy described in the literature first determines the regional minima independent of their altitude [Meyer94]. Afterwards, the adjacent pixels of these minima are added to a hierarchical queue. At each iteration the pixels with the lowest altitude are popped from the queue and processed. This step is repeated until all pixels are processed, simulating an over-flooding of the processed data (called in [Meyer94] *hill climbing*). Thus, a sort of ordered region growth is performed.

Another approach for catchment basin computing is described in [Vince91]. The authors simulate a flooding process, whereas the water is coming up out of the ground and flooding the catchment basins without predetermining the re-

gional minima. The preprocessing step here consists of sorting all (pointers to) pixels in an array. Utilizing a First-In-First-Out (FIFO) structure, the pixels at altitude *h+1* are processed after those at altitude *h*. This divides the problem into *m* subproblems, where *m* is the number of all present pixel altitudes. Due to the processing of pixels at altitude *h* in every iteration, the problem is reduced to calculating the geodesic *skeleton of influence zones* (SKIZ). After sorting the pixels depending on their altitude, in order to guarantee fast access to pixels at given *h*, the SKIZ for each *h* is computed. Hence, the plateaus at the current altitude are flooded. Whenever two floods originating from different catchment basins reach each other, a dam is built to prevent the basins from merging. The presented approach is applied in [Vince91] to several data structures, including graphs and grids with an arbitrary connectivity.

The authors of [Moga95] describe another approach for computing the watershed transformation, based on rainfalling simulation within a gray-scale image. In their work a parallel algorithm is described. The first step transforms the original image into a *lower complete* image I_l. In I_l the pixels belonging to a non-minimum plateau are labelled with the geodesic distance to the plateau's nearest outdoor. In doing so, a second ordering relation for the pixels in a non-minimum plateau is introduced in the resulting image. Afterwards a raindrop starts at each pixel and its path toward the line with the steepest descent (due to gravity) is followed until a regional minimum is reached (as shown on the right in Figure 1). The set of all pixels attracted on the way to a particular regional minimum defines the catchment basin for this minimum. This process is sequentially performed for every pixel, which results in a set of catchment basins. Adjacent catchment basins are separated by watershed lines (depicted in below). Thus, raindrops falling on both sides of a watershed line flow into different catchment basins.

2 Motivation

The algorithms introduced in the previous section work well with regular gray-scale images. However, as we will show in Section 4 and 5, when processing large images or even volume data sets, the time cost is significant.

In this work we present a new algorithm for computing the correct watershed transformation based on the rainfalling simulation. Our algorithm utilizes structures similar to the *Arrowing*-technique presented in [Meyer94], while improving the memory and time cost of previous ones. Although, our algorithm is applied to rectangular grid structure, it is also applicable to grid structures with higher connectivity.

The idea of the rainfalling simulation is presented in [Moga95]. Unfortunately, several problems occur with the implementation as we will outline in the remainder of this work. Furthermore, we propose efficient removing of these obstacles and discuss in detail the time and memory requirements for the presented approach (which is omitted in [Moga95]).

3 Algorithm

In this section we describe our algorithm. First, some useful notations are defined, then we outline a description of the proposed algorithm. Finally, special attention is payed on the details of the introduced steps, illustrated in Figures 1 and 2.

3.1 Notations

For clarity, we introduce the single steps for the 2D case considering pixels located on a regular rectangular grid without loss of generality. Before describing in detail our approach, we make some definitions used throughout the remainder of this work. We define D_f to be the domain of the gray-scale image or volume data set, where f denotes the image (volume) function. $N_G(p)$ stands for the neighbours of a pixel *p* on the underlying grid *G*. Furthermore, we define the following terms:

- A pixel $p \in D_f$ is called an *isolated minimum* if $f(p) < f(q),\ \forall q \in N_G(p)$;
- A pixel $p \in D_f$ is defined as being *on a plateau* P with altitude h (or $p \in P_h$), if $\exists q \in N_G(p)$ with $h = f(p) = f(q)$;
- A pixel $p \in D_f$ is called an *outdoor* of a plateau P, if p is on the plateau P and $\exists q \in N_G(p)$ such that $f(p) > f(q)$;
- A pixel $p \in D_f$ is called an *inner pixel* of a plateau P, if $\forall q \in N_G(p), f(q) = f(p)$;
- A pixel $p \in D_f$ is called a *border pixel* ($p \in B(P)$) of a plateau P, if $p \in P$ and p is not an *inner pixel*;

- A plateau P is called a *minimum plateau* (or P_M) in D_f, if $\nexists p \in B(P)$, such that p is an outdoor;
- A plateau P is called a *non-minimum plateau* (or P_N) in D_f, if $\exists p \in P$, such that p is an outdoor.

3.2 The Rainfalling Simulation

The first step performed in [Moga95] is a preprocessing step, which determines the regional minima, as well as the lower distance within non-minimum plateaus (the image is said to be transformed into a *lower complete* image). Afterwards, the simulation is started. In our algorithm, this step is *not* performed. We sequentially scan the data only once, by performing the following steps: Every pixel *p* is compared with the adjacent pixels and if possible the path of steepest descent is followed and *p* is pushed on a stack S_c[1], containing the pixels on the current path. Otherwise, if a plateau *P* is reached, the whole plateau is processed in order to determine the nearest outdoor *o* (see also Section 3.3). All pixels on the plateau along the path toward *o* are pushed on the stack S_c as well. The algorithm continues with the pixel *o*. Notice that $o \in B(P)$, hence we are still on the plateau, when continuing with *o*. This way we are able to handle pixels, for which more than one $q \in N_G(o)$ with $f(q) < f(o)$ exists, as this is the case for *p=(3,3)* in Figures 1 and 2. Every time a regional minimum is reached, which is either a plateau without outdoors or an isolated minimum, the pixels pushed on the stack S_c are traversed and marked with the label of the reached minimum.

Now let the pixel p_n be the next unprocessed pixel on the path $(p_1, \ldots, p_n)$ with coordinates *(x,y)*. At this stage we have the following options:

1. $\nexists q \in N(p_n)$ with $f(p_n) > f(q)$, hence p_n is an isolated regional minimum, which is marked with the next available basin Id;
2. $\exists! q \in N(p_n)$ with $f(p_n) > f(q)$, this is the regular case, where the algorithm follows the steepest path toward a regional minimum: along the shortest topographic distance;
3. $\nexists q \in N(p_n)$ with $f(p_n) > f(q)$, however $\exists q \in N(p_n)$ with $f(q) = f(p_n)$, which means, that p_n belongs to a (minimum or non-minimum) plateau;
4. $\exists q_i \in N(p_n),\ 1 \leq i \leq m$ with $f(q_i) = f(q_{i+1})$ for $i = 1,..,m-1$ and $f(p_n) > f(q_i), \forall i$. In this case the algorithm cannot determine which of the neighbouring pixels is the one, the raindrop should flow to.

In case 2, p_n is pushed on the stack S_c and q is set to be the current pixel, since this is the pixel with the lowest topographical distance to p_n. The case 1 terminates the current loop and the pixels pushed on S_c are traversed and marked with the label of the reached regional minimum. More difficult to treat are the cases 3 and 4. In case 3, the pixels belonging to the reached plateau P are determined and P's outdoors are pushed on another stack H_L. If H_L is empty when all pixels belonging to P are processed, P is a regional minimum, thus a new Id is assigned and the pixels in S_c (and P) are marked with this Id. Otherwise, the plateau is processed as described in Section 3.3.

Finally, when case 4 occurs, p_n is pushed on S_c and each of the eligible pixels q_i is considered as points hit by a raindrop and processed. Since the last pixel p_n of the current path $(p_1, \ldots, p_n)$ has a higher altitude than the pixels q_i and a path is always following the steepest slope, none of the pixels $(p_1, \ldots, p_n)$ is affected while q_i are being processed. This allows for the algorithm to remain consistency in this case. Hence, after processing each q_i, the computation of the steepest path for the pixel p_n can be continued.

Conversely to [Moga95], with our algorithm no preprocessing and precomputing of the lower complete image is necessary. Moreover, our approach does not require additional memory, while producing correct results corresponding to the ones computed with the flooding algorithms discussed in Section 1.

3.3 Within a plateau

In order to correctly compute the flooding of a non-minimum plateau P, the pixels pushed at the stack H_L have to be sorted. This in turn guarantees, that a pixel $p \in P$, which is simultaneously reached by two outdoors o_1 and o_2, is correctly marked with the label of the outdoor with the lower neighbour[2]. Therefore, we utilized a sorted *heap* data structure offered by the *STL* library [Budd98, Musse96] for the first stage. Hereby, the outdoors are pushed on the heap sorted with

[1] To speed up the algorithm, S_c is in fact not realized as a stack, but every time we say that a pixel *p* is pushed on S_c, the value of *p* in the output image is set to *point* to the predecessor of *p*. Hence, when a minimum is reached, the path constructed via setting the *arrow* in the direction of the predecessor is traversed backwards and the pixels are labelled.

[2] This is the way the pixels are labelled when the image is flooded.

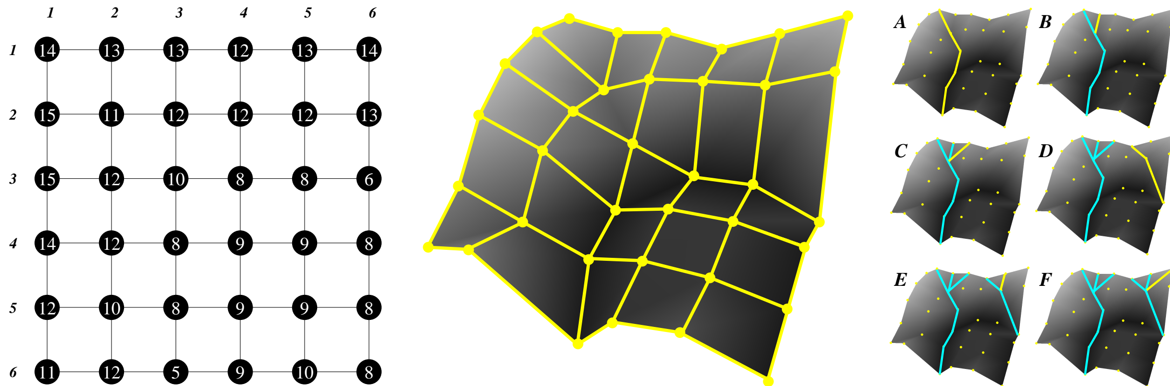

Figure 1: The (simple) image on the left and the corresponding relief in the middle. On the right, the first row of pixels is processed.

respect to the outdoor's neighbour altitude. When flooding the plateau, sorting is no longer required, because the pixels are already stored in the appropriate order. Hence, the neighbours of a popped pixel in the plateau are processed in the correct order. A simple index mechanism allows to assigning the distance to an outdoor without additional overhead. At the beginning, the number i of pixels in H_L is determined and saved. As soon as the currently popped pixel is the pixel with the cardinal number i, the distance to the outdoors is incremented. Since this step is skipped in [Moga95], the produced results cannot be correct.

As introduced above, every time a plateau is reached, it is completely processed and the inner pixels are assigned to the appropriate outdoors. However, this presumes that the outdoors are already assigned to a particular catchment basin. Since this is in general not the case, we code the flooding results in an arrow-like manner, such that it can be utilized for the further data processing. Similar to the *arrowing* technique described in [Vince91] and [Meyer94], we save for every pixel a *coming from*-flag as depicted in Figure 2. This is a six bits long value, representing one of up to 64 directions which the raindrop can follow (which limits the approach to 64-connectivity grids). When an unlabelled border pixel is hit, the algorithm follows the arrows toward the appropriate outdoor, which is the next processed pixel. In case a labelled pixel p' is reaches within the plateau, the catchment basin Id of p' is used to label the current path (see Section 3.5).

3.4 On plateaus' border

The next problem occurs when the currently processed pixel p, which may be an outdoor as well, is adjacent to m pixels $q_i, i = 1, .., m$ with the same altitude $f(q_i) = f(q_{i+1}), i = 1, .., m-1$, such that $f(q_i) < f(p)$ (corresponds to case 4 in Section 3.2). In this case the intuitive solution is to determine the pixel with the shortest distance to an outdoor (if q_i is not on a plateau, the outdoor distance is 1). However, there are situations even in the special 2D case with an eight connectivity grid, where this criteria is not enough to select the next pixel on the current path (as depicted in the Figure 2 for *p=(3,3)*). This inaccuracy is removed by applying the following method during the flooding process: Every time an unvisited pixel with higher altitude than the currently active plateau is detected, additional information is stored in it (see Figure 2). Hereby, not only the *coming from* field is set to point to the currently active plateau pixel. Moreover, the altitude of the nearest outdoor's neighbour is saved[3].

When a pixel *p* is reached, which is adjacent to processed (labelled or not) pixels with lower altitude, all pixels $q \in N_G(p)$ are compared with *p*'s altitude. Those, which have the lowest gray level are stored in a simple queue (for *p=(3,3)* in Figure 2, these are *(4,3)* and *(3,4)*). In order to determine the right successor out of the queue, the outdoor distance of each of these pixels is considered. As introduced, in some cases it is not enough to perform this task, e.g. for *p=(3,3)* in Figure 2. Additionally, we take into account the altitude of the outdoor's neighbour for each equidistant outdoor as shown in Figure 2. In this special case both, *(4,3)* and *(3,4)* belong to the same plateau, which is not the required in general.

Let us assume, that the first raindrop hits p_0 = *(1,1)*. Clearly the next processed pixel is

[3] Due to the sorted order of processing (flooding), the *nearest* outdoor with the *lowest* neighbouring pixel reaches (and marks) correctly this pixel first (see Section 3.3).

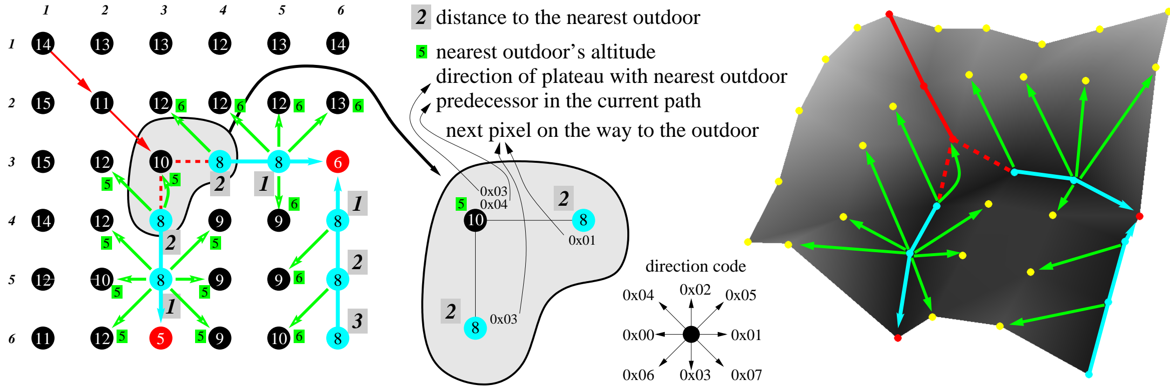

Figure 2: When the pixel *p=(2,2)* is reached, no unique pixel can be selected for continuing the path with steepest descent. The thin arrows show how the pixels with higher altitude are marked, during flooding the plateau at altitude *8*.

p_1 = *(2,2)*, followed by p_2 = *(3,3)*. Continuing with p_2, we cannot unequivocally decide yet which pixel is the right successor. To guarantee the correct computation, the adjacent pixels, independent of whether they belong to a plateau or not, have to be processed first, in a sequential order. In Figure 2 the plateau at altitude *8* is processed and the adjacent pixels with higher altitude are labelled. When this step is performed, some pixels, e.g. *p=(2,6)*, are also labelled, even though this is obviously wrong. However, when *p=(2,6)* is processed, the adjacent pixel with the lowest altitude is *q=(3,6)* and the stored information is not applied. Merely if the pixel in the stored direction and the lowest adjacent pixel have the same altitude and distance to an outdoor, the stored one is selected, as this is the case for *(3,3)* and the plateau at altitude *8*.

Even though we reduced expensive recursive function calls to the minimum, this is an expensive step. This is due to the fact, that all the data in the current scope have to be stored, the pixels processed, whereupon the data has to be restored. However, due to the fact, that during the recursive calls the image is processed without affecting the current path, that the maximal recursion depth (for all data sets discussed in Section 5) is 8, and the average number of recursions[4] is 2.1162 (per 100 processed pixels), this is not significantly slowing down the algorithm's performance.

In [Moga95], the authors consider the first detected pixel with the lowest altitude as the next pixel to be processed. This produces erroneous results as shown in Figure 3, where the framed pixels are ev. misclassified. They may be assigned to the basin with the regional minimum at *(3,6)* or *(6,3)*. With the presented strategy this situation is managed correctly.

[4]The values are statistically determined with the data sets discussed in Section 5.

3.5 Early path termination

In order to speed up the algorithm, the process of rain falling simulation is terminated whenever a marked pixel is reached. A marked pixel is a pixel, belonging to an already processed path, which is labelled with a particular basin Id. In mathematical terms, let the sequence $(p_1, \ldots, p_n)$ be a path with p_n belonging to a regional minimum or being an isolated minimum. If $(q_1, \ldots, q_m)$ is the path already pushed on the stack S_c, $p_i \in N_G(q_m)$, and p_i is a pixel, chosen to be the successor of q_m, then $(q_1, \ldots, q_n, p_i, \ldots, p_n)$ is a *complete steepest* slope path. Notice, that p_i needs not to be the beginning of a steepest slope path, but can be any arbitrary pixel lying *on* a processed steepest slope path. When this case occurs, the pixels in S_c are labelled with the Id of the basin, to which the pixel p_i belongs. This technique we call the *early path termination*. The correct result of the rainfalling simulation for the grid depicted in Figure 2 is shown in Figure 3.

4 Performance Discussion

Requirements for our algorithm are: the *random access* to all image pixels p and adjacent pixels $q \in N_G(p)$. Since the pixels q are accessed only for reading, they are cached when p is read out of the memory.

4.1 Time Cost

The proposed algorithm runs in linear time $O(N)$ with respect to the number of input pixels N.

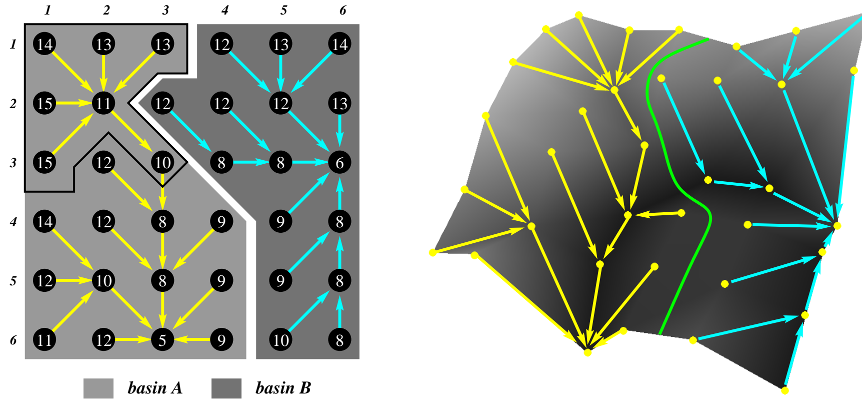

Figure 3: Final catchment basins. Choosing an arbitrary pixel, when processing *p=(3,3)* causes erroneous classification of the framed pixels.

During the data processing, each pixel p which is not belonging to a plateau ($\forall q \in N_G(p),\ f(p) \neq f(q)$) is processed only twice. This happens the first time, when a raindrop following the steepest path to a regional minimum attracts p, or when p is hit itself by a raindrop. When the regional minimum is reached, the pixel is processed again and labelled[5] with the regional minimum's Id. On the other hand, each $p_i \in P$ is processed not more than three times. If P is a minimum plateau, all $p \in P$ are processed and the pixels adjacent to the plateau are compared. Afterwards, if P does not have an outdoor, the pixels are labelled with the corresponding basin Id. Otherwise, the plateau is flooded, whereby every pixel is processed again in order to assign the distance to the nearest outdoor. When a raindrop follows the steepest path along the plateau, the pixels are processed once again during the labelling phase. Merely the sorting step, performed once per non-minimum plateau P over P's outdoors, requires in general $O(n \cdot \log n)$ steps. To avoid this expensive step, the outdoors can be visited first, in order to get the frequence distribution of the outdoors' altitudes. During the second loop, the pixels can be directly inserted in the right location in the heap. Since all these steps require linear time, the entire algorithm is running in linear time. This holds also for the algorithm discussed in [Vince91]. Due to the sorting step, the method proposed in [Meyer94] requires $O(n \cdot \log n)$ time. Furthermore, while our approach processes each pixel not more than 3 times[6], these two algorithms require at least 3 pixel accesses. In particular, the algorithm performing the flooding out of predetermined regional minima requires 3 steps, one of which is an expensive sorting step (therefore $O(n \cdot \log n)$). The first scan determines the regional minima in the data set. Hereby, the regional minima are processed a second time in order to label the pixels. During the second phase of flooding the image, the pixels are processed once again. Additionally, when a pixel p is processed and $q \in N_G(p)$ are added to the pixel queue, they have to be inserted on the right position, which requires one more access. The second referenced approach [Vince91] scans the whole data set two times to construct the sorted array of pointers to pixels. During the flooding step each pixel is scanned three times in average (as described in [Vince91]).

4.2 Memory Requirements

Concerning the memory requirements, it is noticeable, that our algorithm requires only $6\frac{1}{4}N$ bytes of memory, assuming that the input consists of N pixels. In contrast, the first reference method [Meyer94] requires $7N$ bytes of memory ($4N$ bytes for the pixel pointer in the queue and $2N$ for the result and N bytes for additional flags[7]). The approach presented in [Vince91] requires even $7\frac{1}{4}N$ bytes. In our approach the input data consists of 2 bytes (or 65K gray values). For the output we provide $3\frac{1}{2}$ bytes[8]. The first bit marks always whether the pixel is already labelled or not. This defines how to treat the following 27 bits. If it is set, the catchment basin's

[5] Hereby, the second access is much cheaper, as long as *arrowed* pixels can be incrementally processed.

[6] The comparison with the adjacent pixels $q \in N_G(p)$ is not considered as access of q, since this is only a reading access and the values are read and cached when p is read.

[7] Actually the queue for the pixels at altitude h requires additional $4N$ bytes, however, when summarized, the memory required for both queues does not exceed $4N$ in total.

[8] Unfortunately, no time and memory requirements are discussed in [Moga95], hence no comparison can be performed.

Id follows. Otherwise, 6 bits are used to code the *coming from* direction within a non-minimum plateau as introduced above (Section 3.3). Two bytes (or 16 bits) are utilized to code the nearest outdoor's altitude in an adjacent plateau with lower altitude (see Section 3.3). In addition, 6 bits are used to code (the direction of) a lower plateau with the lowest outdoor (described in Section 3.4). This information is stored in the final image and removed, when a label is assigned to a pixel (totalling $3\frac{1}{2}N$ bytes). Unfortunately, there is information, which is required even if the pixel is marked with a particular label: the distance to the nearest outdoor in a non-minimum plateau. This is stored in two auxiliary bytes ($2N$). Since most of the discussed queues are realized through *arrowing* within the presented data structures, the additional memory required in our approach is negligible. Only the step of flooding a non-minimum plateau requires a sorted heap (for the outdoor pixels) resp. queue structure (for the further processing), which consumes in the worst case less than N bytes of memory for all outdoors $p \in P$.

5 Results

The result of the algorithm's application is an image with pixels, labelled with the Id of the catchment basin they belong to. This result can be utilized for further data processing in terms of the specific application area. In order to extract watersheds lines, an incremental loop over the result is performed and watershed lines along basin borders are extracted (as shown in Figure 4). Since the results produced with both immersion based methods and the presented algorithm differ only in single pixels, in Figure 4 we present the original Image (on the left) and the result of applying the watershed transformation (the right image). In Table 1, some running times for processing different data are depicted. They show, that the presented algorithm saves at least 20% up to more than 50% processing time, achieving an average speedup of 1.75 compared to the Meyer's algorithm and 1.3 compared to the Vincent-Soille's algorithm.

6 Conclusion

In this work we presented an algorithm for computing the watershed transformation for a gray-scale (gradient) image. As we have shown, the approach presented in [Moga95] produces in some cases incorrect results (pointed out in Section 3.2). In contrast to this, the approach described in this work produces the correct catchment basins like the ones computed with the classical immersion based watershed algorithms [Meyer94, Vince91]. The first major difference between the presented approach and the one described in [Moga95] is defined by sorting the outdoors, when flooding a non-minimum plateau. This step is required to correctly compute the distance from every inner pixel to an outdoor of the plateau P. In addition, for a path with currently processed pixel p such that the successor cannot be uniquely determined (there is more than one lowest adjacent pixel), the authors in [Moga95] select the first detected lowest pixels to be one processed next. This is the main source of error, since the distance to an outdoor[9] plays an important role in the flooding algorithms. Furthermore, we introduce a third ordering relation for pixels with equal distance to an outdoor[10]: the altitude of the lowest outdoor's neighbour. This way, every pixel is assigned to the correct catchment basin. Finally, the presented algorithm does not need any precomputed information, in contrast to [Moga95], where the data set is prescanned in order to locate the regional minima and preprocess the non-minimum plateaus. Through skipping this step, the presented algorithm can be utilized to start at an arbitrary pixel in the data set and extract only one catchment basin and a given number of adjacent basins. This is of great importance, when large (e.g. volume) data are processed and one is interested only in a particular data region. In this case the steepest path to a regional minimum is followed and a modified local flooding is performed.

The approach presented here is faster and more efficient than the ones described in the literature. We verified this theoretically (in Section 4) and through comparing the computation times of all algorithms discussed in the introduction, while processing the same data (in Section 5, Table 1).

REFERENCES

[Band86] L. E. Band. Topographic partition of watersheds with digital elevation models. *Water Recources Res.*, 22(1):15–24, 1986.

[9] In case this is not a lower *plateau*, the distance is considered as *1*.

[10] Even though, there are pixels with completely equal values with respect to altitude, distance, and outdoor neighbour's altitude, e.g. crest pixels in the relief. In this case no criteria can be defined to choose the correct one. The rule first came first served is applied, like this is the case when an image is flooded with other watershed algorithms.

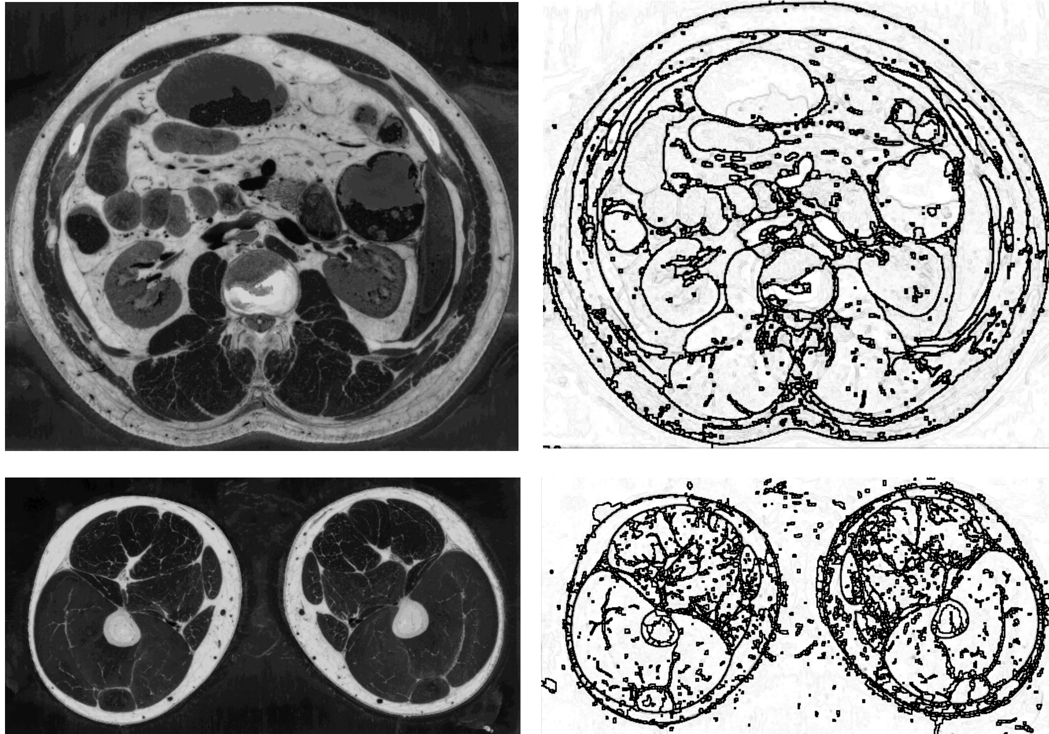

Figure 4: Two of the images utilized for the performance comparison of the algorithms: body and legs cross-sections.

Image	Image Size	num. regions	[Meyer94]	[Vince91]	our approach
Lenna	200x200	2742	2.07 sec	1.4 sec	0.9 sec
body cross-section	519x454	15591	11.31 sec	7.2 sec	4.82 sec
legs cross-section	660x327	9085	8.42 sec	6.23 sec	5.88 sec
6 head CT slices	256x256x6	4097	25.76 sec	22.18 sec	18.93 sec
complete CT head	256x256x113	54110	492.74 sec	430.72 sec	377.61 sec

Table 1: Comparison of the algorithms' runtimes for different input data on a SGI O2 machine.

[Beuch79] S. Beucher and C. Lantuéjoul. Use of watersheds in contour detection. In *International Workshop on Image Processing*, Rennes, Sep 1979. CCETT/IRISA.

[Beuch93] S. Beucher and F. Meyer. The morphological approach to segmentation: the watershed transformation. In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 12, pages 433–481. Marcel Dekker, New York, 1993.

[Bilod94] M. Bilodeau and S. Beucher. Road segmentation using a fast watershed algorithm. In J. Serra and P. Soille, editors, *ISMM'94: Mathematical morphology and its applications to image processing —Poster Contributions—*, pages 29–30. Ecole des Mines de Paris, September 1994.

[Budd98] Timothy Budd. *Data Structures in C++ Using the Standard Template Library*. Addison-Wesley, Reading, MA, USA, 1998.

[Higgi93] W. Higgins and E. Ojard. Interactive morphological watershed analysis for 3D medical images. *Computerized Medical Imaging and Graphics*, 17(4/5):387–395, 1993.

[Meyer94] F. Meyer. Topographic distance and watershed lines. *Signal Processing*, 38(1):113–125, July 1994.

[Moga95] Alina N. Moga, Bogdan Cramariuc, and Moncef Gabbouj. A parallel watershed algorithm based on rainfalling simulation. In *European Conference on Circuit Theory and Design*, volume 1, pages 339–342, Istanbul, Turkey, August 1995.

[Musse96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading (MA), USA, 1996.

[Vince91] Lee Vincent and Pierre Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE PAMI, 1991*, 13(6):583–598, 1991.