A Fast Sequential Rainfalling Watershed Segmentation Algorithm

Johan De Bock, Patrick De Smet, and Wilfried Philips

Ghent University, Belgium
jdebock@telin.UGent.be

Abstract. In this paper we present a new implementation of a rainfalling watershed segmentation algorithm. Our previous algorithm was a one-run algorithm. All the steps needed to compute a complete watershed segmentation were done in one run over the input data. In our new algorithm we tried another approach. We separated the watershed algorithm in several low-complexity relabeling steps that can be performed sequentially on a label image. The new implementation is approximately two times faster for parameters that produce visually good segmentations. The new algorithm also handles plateaus in a better way. First we describe the general layout of a rainfalling watershed algorithm. Then we explain the implementations of the two algorithms. Finally we give a detailed report on the timings of the two algorithms for different parameters.

1 Introduction

Image segmentation is the process of partitioning a digital image in meaningful segments, i.e. segments that show a certain degree of homogeneity. Image segmentation can be interpreted and implemented in many ways. The division into edge detection and region growing algorithms could be a rough classification of segmentation algorithms. The watershed transform can be attributed properties of both classes, i.e. it tries to find the homogeneous closed regions by using an edge indication map as input. In case of intensity segmentation, the edge indication map can be created by calculating the gradient magnitude of the input image. The watershed transform then regards the edge indication map as a topographic landscape in which "valleys" correspond to the interior of segments, whereas the "mountains" correspond to the boundaries of segments. The watershed algorithm derives the "mountain rims" from the landscape and those mountain rims then delineate the segments in the image.

Watershed algorithms can be divided in two classes depending on the method that is used to extract the mountain rims from the topographic landscape. The first class contains the flooding watershed algorithms. These algorithms extract the mountain rims by gradually flooding the landscape. The points where the waterfronts meet each other constitute the mountain rims. This process is displayed chronologically in Fig. 1. A well-known example of this class is the discrete Vincent-Soille flooding watershed algorithm [1,2]. The second class contains the

J. Blanc-Talon et al. (Eds.): ACIVS 2005, LNCS 3708, pp. 476–482, 2005.

rainfalling watershed algorithms. These type of algorithms will be discussed in this paper. Examples of this class are the algorithms described in [3,4] and our previous algorithm [5]. In Sect. 2 we describe the general layout of a rainfalling watershed algorithm. In Sect. 3 we explain the implementations of our two rainfalling watershed algorithms. First we will describe the implementation of our previous algorithm [5]. All the steps needed to do a complete watershed segmentation were done in one run over input data, hence the name one-run algorithm. Next we will describe the implementation of our new algorithm. We separated the watershed algorithm in several low-complexity relabeling steps that can be performed sequentially on a label image, hence the name sequential algorithm. We give a detailed report on the timings of the two algorithms for different parameters in Sect. 4. Finally we draw some conclusions in Sect. 5.

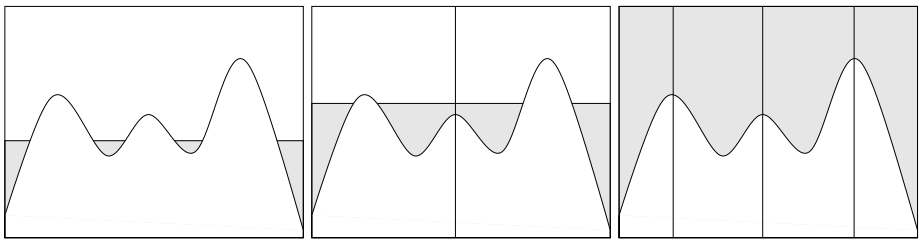

Fig. 1. Chronological stages in the flooding process

2 General Layout of a Rainfalling Watershed Algorithm

A rainfalling watershed algorithm exploits a different concept (compared to the flooding watershed) to extract the mountain rims. For each point on the topographic landscape an algorithm tracks the path that a virtual droplet of water would follow if it would fall on the landscape at that point. All droplets or points that flow to the same local minimum constitute a segment. This concept is depicted in Fig. 2 for the two-dimensional case. The lowest mountains (weakest edges) can be suppressed by drowning them. All the mountains below a certain drowning threshold will not be taken into account. This is shown in Fig. 3.

In the implementation, the rainfalling concept is carried out by calculating the steepest descent direction for each pixel. The directions are limited to the pixels neighboring the central pixel. For a four-neighborhood configuration this results in searching for the lowest neighboring pixel, for an eight-neighborhood configuration we have to take into account an additional $1/\sqrt{2}$ factor for the diagonal directions. A visualization of the steepest descent directions for an image of 10×10 pixels is given in Fig. 4 (eight-neighborhood). The pixels marked with a circle in the middle are pixels from where there is no descent possible. Hence, they are the local minima of the topographic landscape. Every group of pixels that is connected by the same tree of arrows leading to a local minimum must now make up one segment.

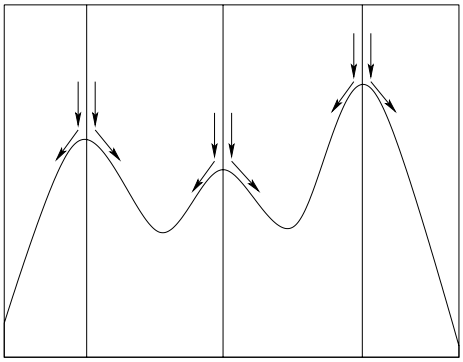

Fig. 2. Rainfalling concept

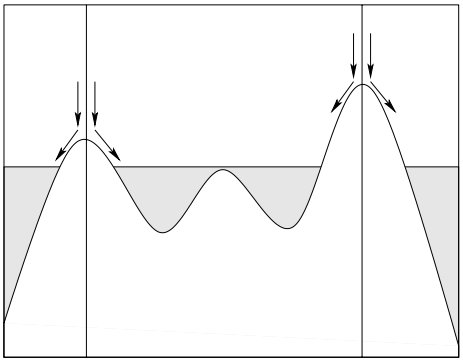

Fig. 3. Drowning threshold

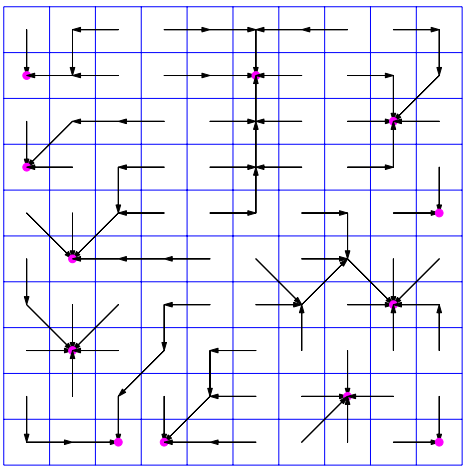

Fig. 4. Steepest descent directions

3 Description of the Rainfalling Watershed Implementations

The input matrix G for both algorithms is the floating point gradient magnitude of an image containing n pixels. The drowning threshold dt and the neighborhood $nbh \in \{4, 8\}$ are the two input parameters. The drowning threshold can also be expressed by the relative drowning threshold rdt, this is dt divided by the maximum value of G. The output data for both cases is a segment label image S, i.e. an image with for each pixel a label of the segment to which the pixel belongs.

First we describe our previous one-run rainfalling watershed segmentation algorithm which has been available publicly. The main data structures are:

- The segment label image, a matrix S of labels. S is initialised in the following way: for each i the algorithm sets $S(i) := i$, with i being the one-dimensional index into S, in video scanning order (from left to right, top to bottom).
- An array P of pointers to pixels. For each segment, P contains a singly linked list of pixels belonging to that segment. To be more precise, $P(i)$ gives the next pixel in the list of pixels of the segment to which pixel i belongs. The start of the list of pixels of the segment with label i is given by $P(i)$. The last pixel i of a list is indicated by $P(i) := -1$. P is initialised with -1 indicating that initially each pixel is a separate segment.

The only operation that will be applied on these two data structures is the $merge(labela, labelb)$ operation. This operation will merge the segments with *labela* and *labelb* by relabeling one of the two segments with the other label in S. P is needed to efficiently locate the pixels with a certain label during the relabeling process. After the relabeling, the lists in P of the two segments are updated by linking the tail of one list with the head of the other list.

The algorithm visits all pixels i in G in video scanning order. If the central pixel i is below dt then all neighboring pixels nb (depending on nbh) are investigated; if nb is below dt then the algorithm executes $merge(S(i), S(nb))$. If the central pixel is above dt then the steepest descent direction (depending on nbh) is calculated. If there is a steepest descent pixel (direction) *steepest* then the algorithm executes $merge(S(i), S(steepest))$. By applying these merge operations the algorithm ensures that after investigating the last pixel, S is in the desired state. Every group of pixels that is connected by the same tree of arrows will now have the same unique label (cfr. Fig. 4).

Now we describe our new sequential rainfalling watershed segmentation algorithm. The main data structures are:

- The segment label image, a matrix S of pointers to pixels, or labels depending on the interpretation.
- The local minima image, a matrix M of labels. M is initialized with 1.

In the first step the algorithm visits all pixels i in G in video scanning order. If the central pixel i is below dt then $S(i) := i$. If the central pixel is above dt then the steepest descent direction (depending on nbh) is calculated. If there is no steepest descent (local minimum) then $S(i) := i$. If there is a steepest descent pixel *steepest* then $S(i) :=$ *steepest* and $M(i) := 0$. The visual interpretation of the state of S after the first step is shown in Fig. 4. After the first step M indicates the locations of the local minima (the pixels below dt are included here).

In the second step the pointers in S are propagated until each pixel points to one of the local minima. For each pixel i the algorithm repeats $next := S(next)$ starting with $next := i$ until it reaches a local minimum, i.e. until $next = S(next)$, then the algorithm sets $S(i) := next$. This step thus implements the tracking of the virtual droplet described above.

In the third step the algorithm applies a connected components algorithm directly on the local minima image M. This connected components algorithm

assigns a different label to each separately connected group of local minima. This step is necessary to be able to merge the connected local minima.

In the final step the algorithm incorporates the new labels given by the connected components algorithm by doing the following relabeling: for each pixel i the algorithm sets $S(i) := M(S(i))$. S is now in the desired state.

Assuming that the labels and pointers all take up 4 bytes, then the one-run algorithm uses approximately $8n$ bytes and the sequential algorithm uses approximately $9n$ bytes. These n bytes extra are used up by the connected components algorithm.

Theoretically these two implementations are identical, except for the handling of plateaus. A plateau is a group of connected local minima above the drowning threshold. In the sequential algorithm these plateaus are handled by the connected components step. Consequently, the individual pixels of the plateau are merged to one segment. In the one-run algorithm these plateau pixels will not be merged and therefore will form individual segments. If the topographic landscape is created by calculating the gradient magnitude of the input image, then linear gradients in the input image will result in plateaus in the landscape. Perceptually it is more appropriate to segment a linear gradient into one segment instead of individual pixel segments.

4 Results

To compare the performance of our two rainfalling watershed algorithms, we tested them on the well-known test image PEPPERS 512x512. The algorithms were implemented in C, compiled with gcc 3.4.2 with optimization parameter -O3 and run on an Intel Pentium 4 2.8 GHz. To obtain very accurate and reliable timings, we measured the time needed to execute 1000 watershed runs. The results for one watershed run are displayed in Fig. 5 for an eight-neighborhood configuration and Fig. 6 for a four-neighborhood configuration. We can see that the new implementation is approximately two times faster for parameters that

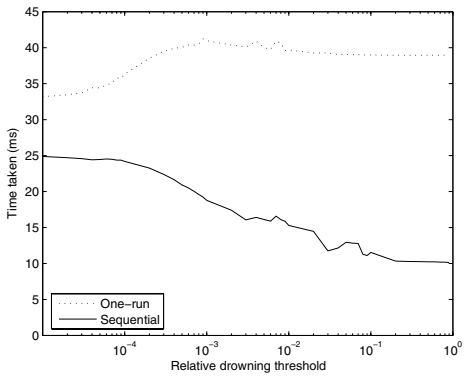

Fig. 5. PEPPERS 512x512, $nbh = 8$

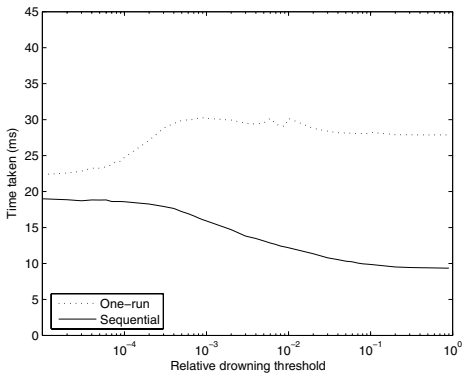

Fig. 6. PEPPERS 512x512, $nbh = 4$

Fig. 7. PEPPERS 512x512, segmented with $rdt = 0.001$, $nbh = 8$

produce visually good segmentations. An example segmentation result is given in Fig. 7. The difference in computation time can be explained by the more efficient merging of a group of connected pixels below the drowning threshold. The difference clearly shows when we test the algorithms on the specific test pattern depicted in Fig. 8. The one-run algorithm needs 48.9 ms to segment the pattern, the sequential algorithm only needs 11.6 ms. Almost all pixels in this pattern are below the drowning threshold, thus the running time is almost completely dominated by the part that merges the connected pixels below the drowning threshold.

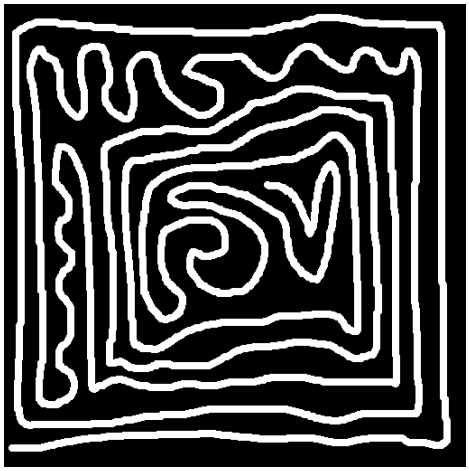

Fig. 8. Test pattern

For a comparison of our one-run algorithm with a Vincent-Soille based flooding watershed algorithm we refer to our previous paper [5]. That paper showed that the one-run algorithm is significantly faster than the flooding watershed algorithm.

5 Conclusion

In this paper we presented a new implementation of a rainfalling watershed segmentation algorithm. We separated the rainfalling watershed algorithm in several low-complexity relabeling steps that can be performed sequentially on a label image. The new algorithm handles plateaus in a better way and is approximately two times faster than our previous implementation for parameters that produce visually good segmentations. This is mostly due to the more efficient merging of a group of connected pixels below the drowning threshold. With execution times of approximately 20 ms (for images of size 512x512), this algorithm can be used to perform real-time video segmentation with a normal PC.

References

1. Vincent, L., Soille, P.: Watersheds in digital spaces: An efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence **13** (1991) 583–598
2. Soille, P.: Morphological Image Analysis: Principles and Applications. Springer (1999)
3. Beucher, S.: Segmentation d'images et morphologie mathématique. PhD thesis, School of Mines (1990)
4. Moga, A., Cramariuc, B., Gabbouj, M.: A parallel watershed algorithm based on rainfalling simulation. In: Proc. 12th European Conf. Circuit Theory and Design. Volume 1. (1995) 339–342
5. De Smet, P., Pires, R.: Implementation and analysis of an optimized rainfalling watershed algorithm. In: Proc. Electronic Imaging, Science and Technology, Image and Video Communications and Processing. (2000) 759–766