A Disjoint Set Algorithm for the Watershed Transform

Meijster, Arnold; Roerdink, Jos B.T.M.

Published in:
Proc. IX European Signal Processing Conference (EUSIPCO '98), Rhodos, Greece

A DISJOINT SET ALGORITHM FOR THE WATERSHED TRANSFORM *

Arnold Meijster, *Jos B.T.M. Roerdink*
University of Groningen
Institute for Mathematics and Computing Science
P.O. Box 800, 9700 AV Groningen, The Netherlands
Tel. +31-50-3633931, Fax. +31-50-3633800
e-mail: arnold@cs.rug.nl, roe@cs.rug.nl

Abstract

In this paper the implementation of a watershed transform based on Tarjan's Union-Find algorithm is described. The algorithm computes the watershed as defined by Meyer in [4]. The algorithm consists of two stages. In the first stage the image to be segmented is transformed into a lower complete image, using a FIFO-queue algorithm. In the second stage, the watershed of the lower complete image is computed. In this stage no FIFO-queues are used. This feature makes parallel implementation of the watershed transform much easier.

1 INTRODUCTION

A commonly used algorithm for digital image segmentation in the field of *mathematical morphology* is the *watershed transformation* (see [4]). The basic idea is to look upon a digital gray scale image as a landscape. The gray level of a pixel is regarded as the altitude of that pixel. A drop of water on the surface of this landscape will flow down along a path of steepest descent until it reaches a (regional) minimum. The set of all pixels for which a drop of water will end in the same minimum is called a *catchment basin*. For some pixels it cannot be decided to which catchment basin they belong. These pixels form the boundaries between the catchment basins. These boundaries are called *watershed lines*.

*To appear in: Proc. EUSIPCO'98, IX European Signal Processing Conference, September 8 - 11, 1998, Rhodes, Greece. Postscript version obtainable at http://www.cs.rug.nl/~roe/

Several mathematical definitions of this informal concept exist (see [3, 4, 6]). None of these definitions is mathematically equivalent to one of the others, but in most practical cases the differences are negligible. In this paper we adopt the definition given by Meyer in [4] based on shortest paths. We start with a short summary of this definition.

A digital gray scale image is a function $f : D \longrightarrow \mathbb{N}$, where $D \subseteq \mathbb{Z}^2$ is the domain of the image and $f(p)$ denotes the gray value of a pixel $p \in D$. Let E denote the underlying grid, *i.e.* E is a subset of $\mathbb{Z}^2 \times \mathbb{Z}^2$. A *path* P of length l between two pixels p and q is an $(l+1)$-tuple $(p_0, p_1, \ldots, p_{l-1}, p_l)$ such that $p_0 = p$, $p_l = q$ and $\forall i \in [0, l) : (p_i, p_{i+1}) \in E$. The length of a path P is denoted by $l(P)$. We denote the set of all paths from p to q by $p \rightsquigarrow q$. A *descending path* is a path along which the altitude does not increase. By $\Pi_f^{\downarrow}(p)$ we denote the set of all descending paths starting in a pixel p and ending in some pixel q with $f(q) < f(p)$. For a pixel $p \in D$ the set of neighboring pixels of p is defined as $N_E(p) = \{q \in D \mid (p, q) \in E\}$.

For pixels in the interior of a plateau, *i.e.* a region of constant altitude, it is not clear in which direction a drop of water would flow. Several solutions for this problem have been proposed. First, let us assume that the function f is *lower complete*. A gray scale image $f : D \longrightarrow \mathbb{N}$ is called *lower-complete* if and only if

$$(\forall p \in D : (\exists q \in N_E(p) : f(q) < f(p)) \vee \Pi_f^{\downarrow}(p) = \emptyset)$$

The interpretation of this formula is that each pixel has at least one neighbor which has a smaller gray-value, or the pixel is located inside a regional minimum.

We also assume that for every pixel p which is inside a regional minimum, we have $f(p) = 0$. The *lower slope*, which is the *maximal* slope linking a pixel p to any of its neighbors of *lower altitude*, is defined as

$$LS_f(p) = \max_{q \in \{p\} \cup N_E(p)} (f(p) - f(q))$$

The cost for walking from one pixel p to a neighboring pixel q is defined as

$$cost_f(p, q) = \begin{cases} LS_f(p) & \text{if } f(p) > f(q) \\ LS_f(q) & \text{if } f(p) < f(q) \\ \frac{LS_f(p) + LS_f(q)}{2} & \text{if } f(p) = f(q) \end{cases}$$

The *topographical distance* between two pixels p and q *along a path* $P = (p = p_0, \ldots, p_{l(P)} = q)$ is defined as

$$T_f^P(p, q) = \sum_{i=0}^{l(P)-1} cost_f(p_i, p_{i+1})$$

The *topographical distance* between points p and q is defined as the minimum of the topographical distances along all paths between p and q:

$$T_f(p,q) = \min_{P \in p \rightsquigarrow q} T_f^P(p,q)$$

The topographical distance between a point $p \in D$ and a set $A \subseteq D$ is defined as $T_f(p,A) = \min_{a \in A} T_f(p,a)$. Note that the topographical distance is not a real distance, since the topographical distance between two different pixels p and q in the same regional minimum is 0. This poses no problems for the implementation, however.

Let $(m_i)_{i \in I}$ be the minima of the function f. The catchment basin of a minimum m_i, denoted $CB_f(m_i)$, is defined as the set of points $p \in D$ that are topographically closer to m_i than to any other minimum m_j:

$$CB_f(m_i) = \{p \in D \mid \forall j \in I \backslash \{i\} : T_f(p,m_i) < T_f(p,m_j)\}$$

The watershed of a function f is the set of points of its domain which do not belong to any catchment basin:

$$Wsh(f) = D \cap (\cup_{i \in I} CB_f(m_i))^c$$

In practice, of course, images are not always lower complete, and the altitude of the regional minima need not be zero. The following construction can be used to transform an image f such that it satisfies these requirements. We compute a function f^* in which for every regional minimum (which could be a plateau) we set the altitude to 0, and for all the other pixels p in the image we set the altitude to the length of the shortest path in $\Pi_f^{\downarrow}(p)$:

$$f^*(p) = \begin{cases} 0 & \text{if } \Pi_f^{\downarrow}(p) = \emptyset \\ \min_{P \in \Pi_f^{\downarrow}(p)} l(P) & \text{otherwise} \end{cases}$$

Let $L_c = \max_{p \in D} f^*(p)$. We construct the function f_{LC} as follows:

$$f_{LC}(p) = \begin{cases} 0 & \text{if } f^*(p) = 0 \\ L_c \cdot f(p) + f^*(p) - 1 & \text{otherwise} \end{cases}$$

The function f_{LC} is lower complete, while for a pixel p in a regional minimum we have $f_{LC}(p) = 0$. A linear time algorithm, given in Fig. 2, using a FIFO-queue breadth-first algorithm to propagate distances computes the function f_{LC}.

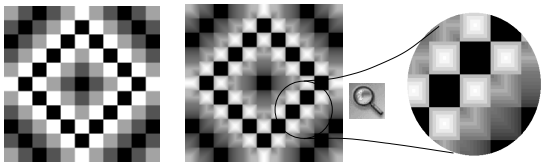

Fig. 1: *Image (left), and lower complete image (right).*

2 TARJAN'S UNION-FIND ALGORITHM

In [5] Tarjan presents an algorithm for maintaining *disjoint sets* under the set-union operation. Since catchment basins are disjoint sets by definition this algorithm seems applicable. However, some modifications are necessary. In this section we present Tarjan's algorithm for disjoint sets, and in the next section we will show how to modify this algorithm such that it can be used for the computation of watersheds.

Tarjan stores sets in trees. Each node in the tree points to its parent. The root of a tree points to itself. Two objects x and y are members of the same set if and only if x and y have the same *canonical element*. The canonical element of x is the root of the tree in which x is stored. There are three important operations.

- *MakeSet(x)*: Create a new singleton set $\{x\}$. This operation assumes that x is not already member of any set.
- *FindRoot(x)*: Return the *canonical element* (the root of the tree) of the set containing x.
- *Union(x,y)*: Form a new set that is the union of the two sets whose canonical elements are x and y. This operation assumes $x \neq y$.

The trees are implemented in a linear array, named $parent$, of which the indices are of the same type as the type of the objects stored in it (usually integers). The value $parent[x]$ gives the parent of x in the tree x is contained in. When x is a canonical element, we have $parent[x] = x$.

Obviously, the operations *MakeSet(x)* and *Union(x,y)* can be performed in constant time, but the operation *FindRoot(x)* requires a search for the canonical element of x. This operation takes time linear in the length of the path from x to its canonical element. Tarjan uses two important techniques to keep these paths reasonably short.

The first technique is called *path compression*. Every time the operation *FindRoot(x)* is applied, the parent pointer of the nodes on the *find-path* (the path from x

```
procedure Lower (im[1:HEIGHT, 1:WIDTH] : int)
    returns lc[1:HEIGHT, 1:WIDTH] : int
 # Init queue with pixels that have a lower neighbor
 queue := EmptyQueue;
 forall (i, j) ∈ D do
    lc[i, j] := 0;
    if (∃(ii, jj) ∈ N_E(i, j) : im[ii, jj] < im[i, j]) then
       FifoAdd((i, j), queue); lc[i, j] := −1
    endif
 endforall
 dist := 1; FifoAdd((−1, −1), queue);
 while queue ≠ EmptyQueue do
    (i, j) := FifoRemove(queue);
    if (i, j) = (−1, −1) then
       if queue ≠ EmptyQueue then
          FifoAdd((−1, −1), queue)
          dist := dist + 1;
       endif
    else
       lc[i, j] := dist;
       forall (ii, jj) ∈ N_E(i, j) such that
             im[ii, jj] = im[i, j] ∧ lc[ii, jj] = 0 do
          FifoAdd((ii, jj), queue);
          lc[ii, jj] := −1 # To prevent from queueing twice
       endforall
    endif
 endwhile
 forall (i, j) ∈ D such that lc[i, j] ≠ 0 do
    lc[i, j] := dist ∗ im[i, j] + lc[i, j] − 1
 endforall
end
```

Fig. 2: *Transformation into a lower complete image.*

to the root of the tree) is changed to point directly to the root of the tree. Thus, after the operation *FindRoot(x)*, a second operation *FindRoot(y)*, with y on the find-path of x, takes constant time.

The second technique, *union by rank*, is used in the operation *Union(x,y)*. The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. However, this technique is not used in this paper.

Tarjan [5] shows that for an intermixed sequence of m operations the time complexity of this algorithm is for all practical purposes linear in m.

```
procedure MakeSet (x : int)
 parent[x] := x; rank[x] := 0
end;

procedure Link (x, y : int)
 parent[x] := y
end;

procedure FindRoot (x : int) returns root : int
 if x ≠ parent[x] then parent[x] := FindRoot(x) endif
 root := parent[x]
end;

procedure Union (x, y : int)
var px, py : int
 px := FindRoot(x); py := FindRoot(y)
 if rank[px] > rank[py] then Link(py, px)
 elseif rank[px] < rank[py] then Link(px, py)
 else Link(px, py);
     rank[py] := rank[py] + 1
 endif
end;
```

Fig. 3: *Tarjan's algorithm for disjoint sets.*

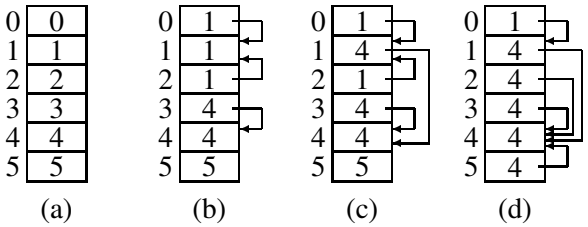

Fig. 4: *(a) MakeSet(0); . . .; MakeSet(5) (b) Union(0,1); Union(1,2); Union (3,4) (c) Union (1,3) (d) Union (2,5).*

3 A MODIFICATION OF TARJAN'S ALGORITHM FOR COMPUTING WATERSHEDS

In this section we will show how Tarjan's algorithm can be used to compute the watershed of a lower complete image f, of which the regional minima are uniquely labeled. The label assigned to a minimum is the canonical element for that minimum.

```
procedure Resolve (p : pixel) returns ce : pixel
 # Returns canonical element of pixel p, or
 # WSHED=(-1,-1) in case p lies on a watershed
 i := 1; ce := (0,0); # some value such that ce ≠ WSHED
 while (i ≤ 4) ∧ (ce ≠ WSHED) do
     if (sln[p,i] ≠ p) ∧ (sln[p,i] ≠ WSHED) then
        sln[p,i] := Resolve(sln[p,i])
     endif
     if i = 1 then ce := sln[p,1]
     elseif sln[p,i] ≠ ce then
        ce := WSHED;
        for i := 1 to 4 do sln[p,i] := WSHED endfor
     endif
     i := i + 1
 endwhile
end;

procedure Basins ()
 forall (i,j) ∈ D do Resolve ((i,j)) endforall
end;
```

Fig. 5: *Resolving the downstream paths of the DAG.*

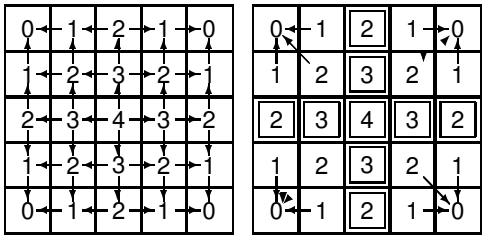

Fig. 6: *Left: image and its corresponding DAG; right: DAG after resolving (watershed-pixels are surrounded by a box).*

In the remainder of this paper we use 4-connectivity (although the algorithms can also be applied in the case of 8-connectivity).

The algorithm proposed is based on the following theorem (see [4]). Let $m \subseteq D$ be a regional minimum, and $q \in m$. If a pixel $p \in D$ belongs to the catchment basin of m, then the difference in altitude of p and q equals the topographical distance between p and q, *i.e.*

$$(\forall p \in D : p \in CB(m) \Rightarrow T_f(p,q) = f(p) - f(q)),$$

and the path realizing the distance $T_f(p,q)$ is a path of steepest descent, *i.e.*, each step along the path is to a neighbor $p' \in N_E(p)$ with the lowest altitude. In some cases there will be several such neighbors. In these cases there is no preference for any of the neighbors, and all paths via these neighbors are followed. In the case that at least two of these paths end in different regional minima, say m_0 and m_1, we have $T_f(p,m_0) = T_f(p,m_1)$, and hence p belongs to the set of watershed pixels.

The disjoint set forest used by Tarjan is replaced by a directed graph, whose only cycles are self-loops. With some abuse of terminology we refer to this as a *directed acyclic graph (DAG)*. This is similar to the arrowing method of [1, 4]. Let $G_{CB} = (D, E)$ be this DAG. For a pixel $p \in D$ which is not in a regional minimum, and for each of its lowest neighbors $q \in N_E(p)$, we have $(p,q) \in E$. For a regional minimum m a single pixel $r \in m$ is chosen as the canonical element of this minimum, and we have $\forall(p \in m : (p,r) \in E)$ (since $(r,r) \in E$ there are self-loops). The reason we use a DAG instead of a disjoint set forest, is the fact that a pixel can have more than one steepest lower neighbor, and the fact that we cannot determine on the fly whether a pixel belongs to a catchment basin or it belongs to the set of watershed pixels, and thus it is simply added to the DAG. The algorithm is given in Fig. 5. The DAG is stored in an array sln, where $sln[p,i]$ is a pointer to the i^{th} steepest lower neighbor of pixel p. The DAG G_{CB} can be constructed in a single pass scan-line algorithm, in which for each pixel only its neighbors are referenced, which results in optimal use of the cache memory of the processor. After the DAG is constructed, all the directed paths in the DAG can be resolved by following the outgoing pointers of each node until a canonical element is reached. In the case that two or more different canonical elements can be reached from a node v in the DAG, the node v is a watershed node. The resolving algorithm closely resembles Tarjan's $FindRoot$ operation. For reasons of elegance, the algorithm is presented as a recursive algorithm, however, in practice the recursion should be eliminated to reach optimal performance.

4 RESULTS AND CONCLUSIONS

We applied the algorithm (both stages) to a number of test images. The results shown in the table below are for hundred runs, and are performed on square images of sizes 256×256, 512×512, and 1024×1024 respectively. The computer used is a 300MHz Pentium PC, with 64 Mb RAM memory, and 512 Kb cache memory.

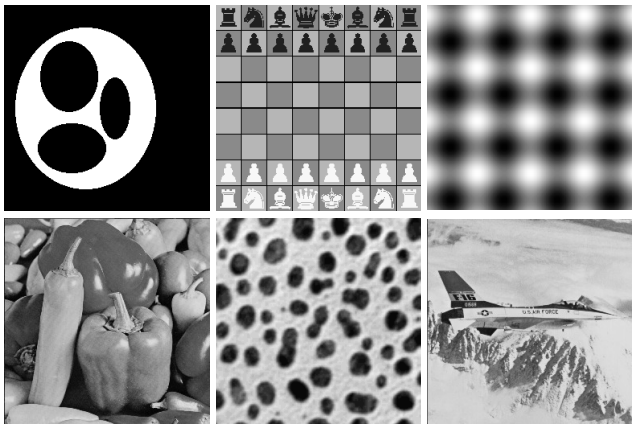

Fig. 7: *Test images (from top left to bottom right): (a) blobs (b) chess (c) waves (d) peppers (e) particles (f) aircraft.*

image	minima	256	512	1024
blobs	4	17.2	74.8	313
chess	67	43.2	178	716
waves	20	36.3	165	720
peppers	44426	37.0	170	712
particles	359	41.6	182	756
aircraft	19053	38.7	174	724

The image 'blobs' is a binary image with very large minima plateaus (more than half of the pixels are in a minimum), resulting in short root-paths in the DAG. This explains the significant shorter running time for this image compared to the other ones. By doubling the image dimensions, the number of pixels increases by a factor of 4. Since all phases of the algorithm are performed in (nearly) linear time with respect to the image size, we expect to find this reflected in the timings. On average we find that the running time increases by a factor of 4.2, which is quite close to linear behavior. The images 'blobs', 'chess', and 'waves' are artificially generated images, with relatively few minima, while the other images are camera-made, containing a lot of noise and minima. We see that the number of minima has little effect on the total running time. The timings we find are comparative with the ones found in the literature for other algorithms ([2, 6]). The main interest of our algorithm is the second stage, since it can be parallelized on shared memory computers with very little synchronization overhead, while most other algorithms are difficult to parallelize as a result of global dependencies. Another approach to

deal with these global dependencies is by modifying the definition of the watershed through a locality assumption, as is done in [2].

References

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

[2] Bieniek, A., Burkhardt, H., Marschner, H., Nölle, M., and Schreiber, G. A parallel watershed algorithm. In *Proc. 10th Scandinavian Conference on Image Analysis (SCIA'97), Lappeenranta, Finland* (1997), pp. 237–244.

[3] Meijster, A., and Roerdink, J. B. T. M. A proposal for the implementation of a parallel watershed algorithm. In *Computer Analysis of Images and Patterns*, V. Hlaváč and R. Šára, Eds., vol. 970 of *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, 1995, pp. 790–795.

[4] Meyer, F. Topographic distance and watershed lines. *Signal Process. 38* (1994), 113–125.

[5] Tarjan, R. E. *Data Structures and Network Algorithms*. SIAM, 1983.

[6] Vincent, L., and Soille, P. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Trans. Patt. Anal. Mach. Intell. 13*, 6 (1990), 583–598.