Topological Watershed

Alexandre Abraham

Technical Report n^o0821, June 2008
revision 1830

Dividing a picture into areas of interest is called picture segmentation, it is useful in particular to point out cancerous cells in medical imaging. The *Watershed Transform* provides such a segmentation and can be implemented in many ways. Here we will focus on the *Topological Watershed*, an efficient algorithm producing results with nice properties. In this report, we will explain all the mechanisms of this algorithm and then show that thanks to Milena, the C++ generic image processing library of the Olena platform, developed at LRDE, it can handle classical image format as well as trickier ones like pictures mapped on general graphs.

Segmenter une image consiste à en extraire les régions d'intérêt, par exemple pour séparer des cellules cancéreuses en imagerie médicale. L'approche par transformation de la ligne de partage des eaux (LPE) ou *Watershed Transform* permet d'obtenir une telle segmentation. Il en existe de nombreuses définitions, ainsi que diverses implémentations, dont certaines sont à la fois performantes et produisent un résultat avec de bonnes propriétés, comme le *Topological Watershed*. Ce rapport présentera
l'implémentation d'un algorithme calculant cette LPE au sein de Milena, la bibliothèque C++ générique de traitement d'images de la plate-forme Olena, développée au LRDE. Nous nous intéresserons tout d'abord aux les formats d'images "classiques", puis la généralisation à des formats d'images plus inhabituels (images à support de graphe généraux, etc.).

Keywords
image segmentation watershed Najman Couprie Meyer

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
abraham@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Copyright © 2008 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

1 State of Art 5
- 1.1 Watershed Algorithm 5
- 1.2 Classical Immersion Watershed 5
- 1.3 The Need for a Topological Watershed 6
- 1.4 Watershed in Milena 7

2 Topological Watershed 8
- 2.1 The idea 8
- 2.2 The algorithms 8
- 2.3 The Component Tree 9
- 2.4 The Least Common Ancestor 10
  - 2.4.1 A Naive Implementation 10
  - 2.4.2 A Naive Implementation With Preprocessing 11
  - 2.4.3 LCA Seen as a Range Minimum Query (RMQ) 12
  - 2.4.4 Fast Preprocess for the RMQ Problem 12
  - 2.4.5 From LCA to Highest Fork 13
- 2.5 Characterisation of Points 13
  - 2.5.1 Goal 13
  - 2.5.2 Preliminary 14
  - 2.5.3 W-destructible Points 14
  - 2.5.4 M-destructible points 16
  - 2.5.5 W-constructible 17
- 2.6 M-watershed 17
- 2.7 W-Watershed 20
- 2.8 Topological Watershed 21

3 TikZ Export 23
- 3.1 The Need for an Image Output 23
- 3.2 From Milena to LaTeX 23
- 3.3 Tikz Color Conversion Algorithm 23
- 3.4 Examples 25

4 Conclusion 27

5 Bibliography 28

A Example 29

Introduction

Milena is a generic and efficient image processing library. As it is aimed to be used by people working in medical imaging, it needs some image segmentation algorithm in order to fulfill the needs of its users.

Image segmentation is a process used to disclose areas of interest. It has many applications in biology or medecine, for exemple to locate tumors, to measure tissue volume... But also for some tasks like face or fingerprint recognition, traffic regulation, site localisation in satellite images...

Many approaches had been implemented to locate the limits between image regions:

- **edge detection** that focuses on the characteristic lines between regions;
- **region detection** that focuses on the detection of the regions (region growing, watershed...);
- **hierarchic fragmentation** that clusters the image in order to get a stack of growing regions (histogram-based methods...);
- **stochastic algorithms** like neural networks.

This paper focuses on watershed algorithms. In fact these algorithms are easy to apprehend and are very flexible, one can choose to use a more efficient algorithm at the expense of information. A quasi linear topological watershed and its implementation in Milena is presented here.

After a brief reminder on the principle of a basic watershed, we will present some common watershed algorithms. Then we will describe step by step a topological watershed algorithm. Finally, a little tool designed to display exotic image formats is presented to conclude.

Acknowledgements

I would like to thank Thíerry Géraud, Alexandre Duret-Lutz, William Caldwell and Caroline Vigouroux for their help on this report.

Chapter 1

State of Art

1.1 Watershed Algorithm

The watershed transformation is a segmentation algorithm mainly used because of its simplicity. It allows a user to slice a greyscale image by filling an image with water. Let us consider a 2D image as a heightmap: each point's altitude is given by its gray level. Consequently, lighter areas are seen as mountains and darker ones as basins.
The relief is now filled with water through holes made in the minimas. Each time that water from two **different** basins meet, a dam is built. Once the whole image is flooded, the image is partitioned into basins separated by the dams, called watershed lines.

Many implementations exist and some very efficient algorithms have been proposed.

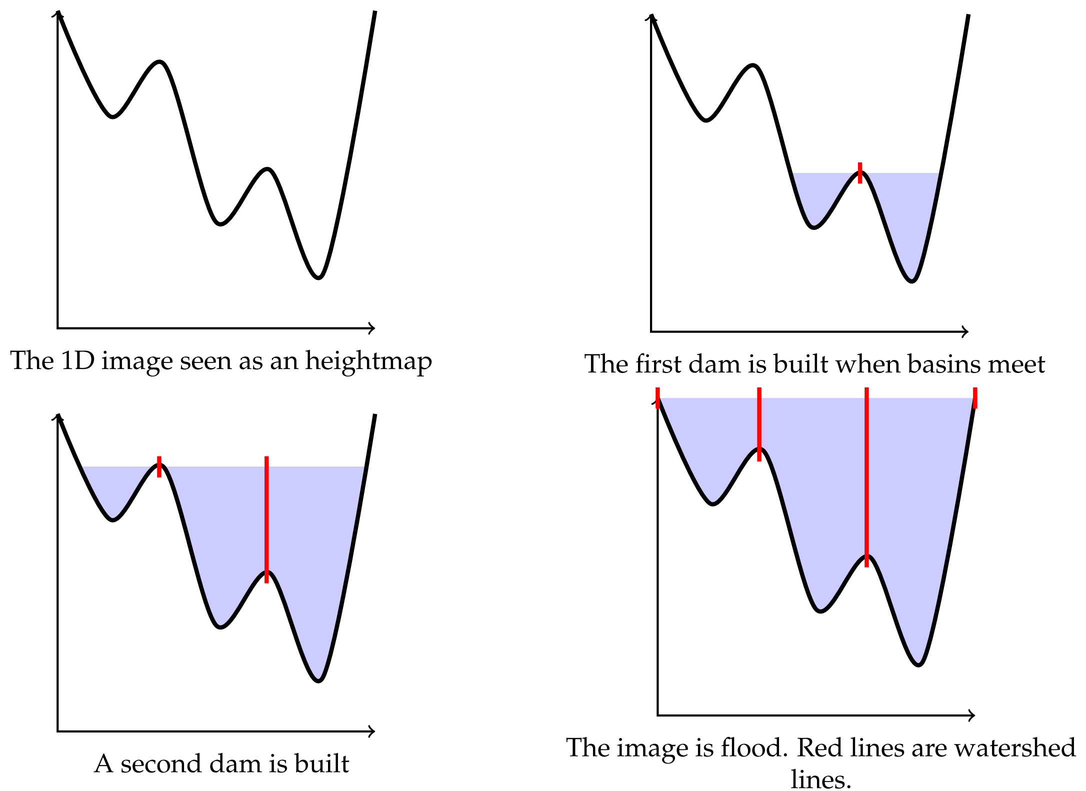
1.2 Classical Immersion Watershed

A commonly used algorithm of watershed is the immersion algorithm : it consists in flooding the image with water. These are the main steps of the algorithm :

- Take the image as an heightmap[1]
- Cut holes in the local minima (darkest points)
- Fill the image through the holes
- Each time two basins meet, build a dam at the top of the crest
- Once the image is flood, the dams are the watershed lines

To illustrate this algorithm, let's take a 1D image and represent it as a function. Then, apply the steps described above.

[1] the grey level of a pixel is taken as its altitude



The 1D image seen as an heightmap

The first dam is built when basins meet

A second dam is built

The image is flood. Red lines are watershed lines.

1.3 The Need for a Topological Watershed

Unfortunately, much information is lost during a classical watershed, essentially contrast information. In fact, most of the time the watershed is in the first stages image processing. It is commonly followed by a region merging in order to obtain the desired granularity. In order to select the region to merge, a binary partition of the image is not enough. The altitude of basins and edges are essential information needed to take a decision on which components to merge. In fact, computing each component's minimum can be done afterward, but it requires a lot of useless operations.

We want a Watershed that preserves more information on the image. A topological watershed has some nice properties and gets rid of major defaults of classical watersheds:

- thickness: if a limit between two basins is delimited by several pixels of the same height, a classical watershed would leave a thick limit between the basins. This behavior is a pain because the pixels in this limit don't belong to any component. The Topological Watershed ensures that all limits are 1 pixel thick;
- contrast: as said before, keeping the contrast of the picture is essential (Najman and Couprie, 2003). In a topological watershed, the height of the edges represents the height of the lowest mountain between the components.

1.4 Watershed in Milena

There is, for the moment, only one watershed algorithm in Milena: Meyer's Watershed which is one of the simplest. Many image processing libraries do not implement any watershed and the one who do implements Meyer's algorithm most of the time.

Few implement a topological watershed – without giving their complexity – like Vigra or Wolfram and others have chosen to propose a classic watershed with the ability to choose the level of segmentation like Insight. It is in fact implemented in Pink, because the algorithm creators are also the authors of this library.

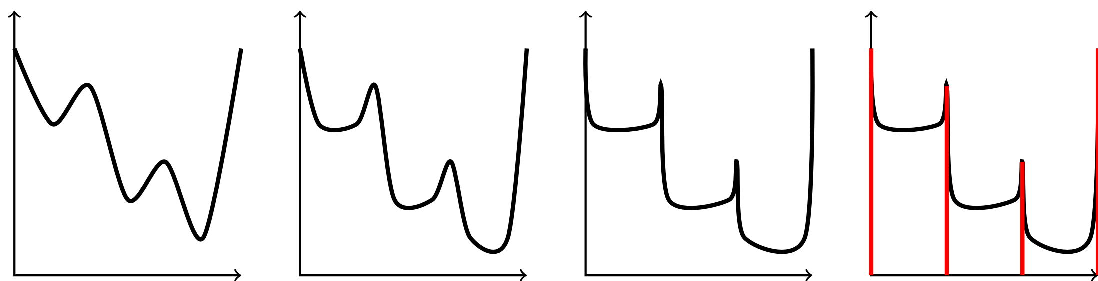
Chapter 2

Topological Watershed

In 2005, Michel Couprie, Laurnet Najman and Gilles Bertrand proposed a quasi-linear implementation for the topological watershed in (Couprie et al., 2005). The reader is highly encouraged to read their paper for a full comprehension of the algorithm. Here is a graphical example of the algorithm in order to give a general idea of it to the reader. Detailed mechanisms are not described here, please refer to the paper.

2.1 The idea

The idea of the topological watershed is to let the water "erode" the relief of the picture. This operation is called "lowering" it consists in reducing the altitude of a point to its minimum.



The processus of topological watershed

2.2 The algorithms

Before describing the general layout of the algorithm, three little tools need to be introduced :

- **the component tree** is a segmentation of the image;
- **the Least Common Ancestor** of two points a and b that gives the furthest node from the root that is an ancestor of both points;

- **the characterisation of points** that uses the tools described above to let us know if a point can be lowered or not.

After this preprocessing, two sub-algorithms are applied to the image :

- M-watershed;
- W-watershed.

In fact, the topological watershed can be done by applying successively the W-watershed until stability (when the image stays the same after two following iterations) but this repetition is costly. That's why we need the M-watershed : applying the M-Watershed before guarantees that a single iteration of the W-Watershed is needed to obtain the topological watershed.

We will also introduce an algorithm that merge both M and W watersheds. This algorithm is faster but offers less liberty to the user.

2.3 The Component Tree

The component tree can be seen as a segementation of the image in layers of increasing height. This structure is needed for the characterisation of points.

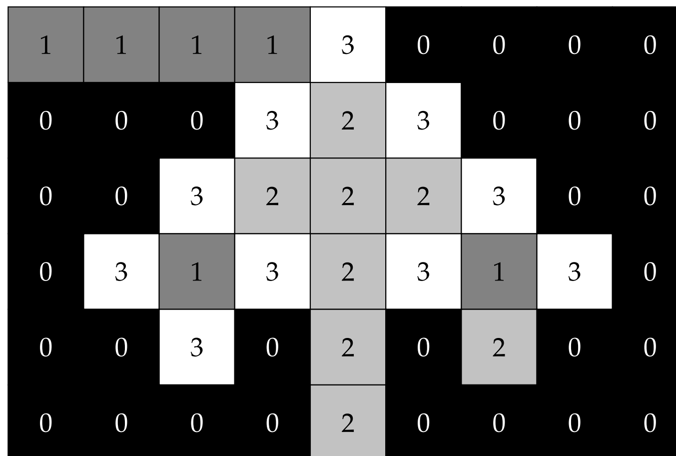


Figure 2.1: example image to process

This algorithm, usually named max-tree (or min-tree) consists in building a hierarchy of the connected components of the image (represented by a tree).

Let I be an image and $k \in \mathbb{K}$. Let's define a binary threshold of the image such as:

$$\forall p \in I, I_k(p) = \begin{cases} 1 \text{ if } I(p) \leq k \\ 0 \text{ otherwise} \end{cases}$$

By overlaying the thresholds, we obtain a tree of the components (2.2).

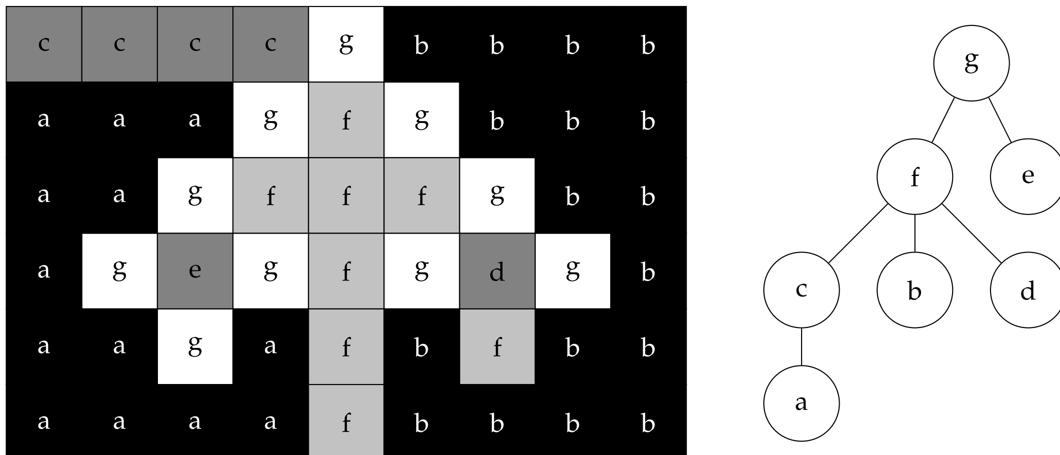


Figure 2.2: The component mapping and its tree

Since this algorithm was already implemented in Olena 0,11, I have just imported it and made a few corrections. This tree will be used later to determine the type of a point.

2.4 The Least Common Ancestor

We defined by the LCA of two nodes a and b the furthest node from the root which is an ancestor of a and b. Even if this algorithm seems unrefined, computing it in a linear time is harder than it seems. Furthermore, this algorithm is extensively called in the watershed and is one of the main culprits for the complexity of the algorithm. In fact, thanks to it and to the component tree, one can determine if a set of points belongs to a shed or not.

2.4.1 A Naive Implementation

When looking at the problem, one would code this basic algorithm by intuition:

```
1  Node LCA (Tree t,     /* the component tree */
2            Node a,     /* the first node */
3            Node b)     /* the second node */
4  begin
5    /* The level of the node is known, first climb up at the same level */
6
7    while (a.level ≠ b.level)
8      if (a.level < b.level)
9        a ← a.father
10     else
11       b ← b.father
12
13   /* This is not the best way to do this but this is not important */
```

```
14
15    while (a ≠ b)
16      a ← a.father
17      b ← b.father
18
19    return a
20
21  end
```

The complexity of this algorithm is $O(log(n))$ but, since the tree will not change during the process, it is easy to see that if the LCA of two nodes is computed one time, we can store the result for a future request. Moreover, we can preprocess the tree in order to answer every request in a linear time.

2.4.2 A Naive Implementation With Preprocessing

The first idea for this implementation is to build a $n \times n$ matrix that gives, for each pair of nodes, the LCA.

```
1  Matrix lca[n][n];
2
3  Preprocess (Tree t)
4  begin
5    for each p in t
6      for each q in t
7        lca[p][q] = LCA(p, q) /* call to our previous LCA function */
8  end
9
10 Node LCA2 (Tree t,    /* the component tree */
11            Node a,    /* the first node */
12            Node b)    /* the second node */
13 begin
14   return lca[a][b]
15 end
```

This implementation raises two problems:

- **the huge size of the matrix:** let's take a 1000×1000 image in which every pixel is a component. The tree has $n = 1000 \times 1000 = 1000000$ nodes, so the matrix size is $s = 1000000 \times 1000000 = 10^{12}$. It is impossible for a computer to handle such an amount of data.
- **the LCA processing time:** in fact, this implementation still relies on the naive LCA implementation described above.

Even if the preprocessing can help to improve the LCA speed, we need to look at the problem closer to find an efficient way to improve it. In fact, this idea is not new. D. Harel and R.E. Tarjan D. Harel (1984) have proposed an algorithm to answer this question in linear time. Let's see how to do so and propose an implementatable algorithm.

2.4.3 LCA Seen as a Range Minimum Query (RMQ)

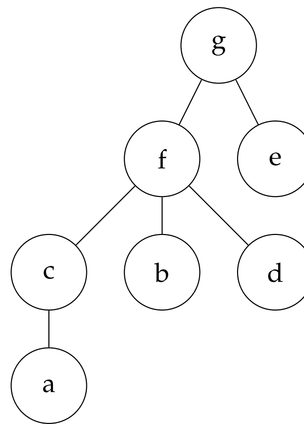
The RMQ problem consists in finding the local minimum in a subarray:

Given a length n array A and two points indices i and j in $[1..n]$, $RMQ_A(i, j)$ returns the index of the smallest element in the subarray $A[i..j]$.

In M.A. Bender (2000), complete description and proof of the following affirmations is given. The step between LCA and RMQ is given in the Observation 4:

Observation 4 *The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a depth first search traversal of T*

This observation is obvious with an example. Let's take the example component tree computed above and build an Euler Tour[1] of it (2.3).



Node	g	f	c	a	c	**f**	b	f	d	f	g	e	g
Depth	0	1	2	3	2	**1**	2	1	2	1	0	1	0

Figure 2.3: Example of LCA computing through RMQ resolution. The LCA of a and d is the RMQ of a and d in the depth array coresponding to the Euler Tour. The minimum (in bold) is the minimum of the subarray (in blue).

This problem is also called ± 1 RMQ because the difference between two values in the depth array is always 1. Now that the LCA problem is reduced to a RMQ, we can focus on a more efficient way to preprocess the datas.

2.4.4 Fast Preprocess for the RMQ Problem

There are many ways to solve this problem. The chosen solution is a good compromise between speed and memory consumption. The idea is to precompute all 2^k long requests only (with $1 \leq k \leq n\, log\, n$). This can be done in $O(n\, log\, n)$ thanks to dynamic computing. Then, to answer the RMQ(i, j) request, one just has to take two overlapping 2^k long subarrays that cover the

[1] The Euler Tour of T is the sequence of nodes obtained by writing down each node encountered during a deprh first search traversal. Each internal node is encountered $c + 1$ times (with $c =$*number of children*) and each leaf only one time. If T has n nodes, the Euler Tour is $2n + 1$ long.

$[i..j]$ subarray (with $k = \lfloor log_2(j-i) \rfloor$) and return the minimum of these two blocks. Let's see how it works on our previous example (2.4).

Node	g	f	c	a	c	**f**	b	f	d	f	g	e	g
Depth	0	1	2	3	2	**1**	2	1	2	1	0	1	0

Position	Size 2	Size 4	Size 8
0	0 (g)	0 (g)	0 (g)
1	1 (f)	1 (f)	1 (f)
2	2 (c)	5 (f)	5 (f)
3	4 (c)	5 (f)	10 (g)
4	5 (f)	5 (f)	10 (g)
5	5 (f)	5 (f)	10 (g)
6	7 (f)	7 (f)	
7	7 (f)	10 (g)	
8	9 (f)	10 (g)	
9	10 (g)	10 (g)	
10	10 (g)		
11	12 (g)		

Example for a (pos. $i = 3$) and d (pos. $j = 8$):

1. determine k, the size of our 2 overlapping subarrays:

$$k = \lfloor log_2(j-i) \rfloor = 4$$

2. identify the k long subarrays:
   - $[i..i+k] = [3..7]$
   - $[j-k..j] = [4..8]$

3. find the RMA thanks to our matrix:

$$RMA(i,j) = min(M[i, log_2 k], M[j-k, log_2 k]) = 5\,(f)$$

Figure 2.4: An exemple of RMQ request processed in linear time

2.4.5 From LCA to Highest Fork

In order to process our image, we would like to apply the LCA on all the neighbours of a point. The HF (Highest Fork) of a set of points V is the LCA of all points of V that doesn't belong to V (the highest fork may not exist). Thanks to this definition, we can now go further in the algorithm with the characterisation of the points.

2.5 Characterisation of Points

2.5.1 Goal

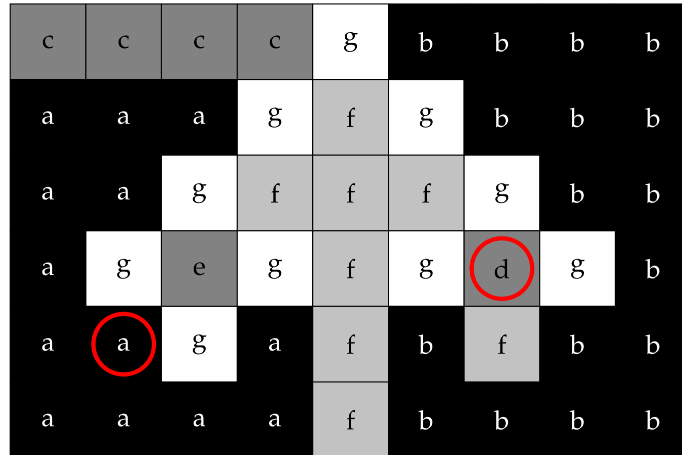
The topological watershed algorithm consists in lowering points. But to lower a point, one has to know the new value to give to this point, that's the purpose of the characterisation. There are many kinds of interesting points in a picture (see Couprie et al. (2005)) but we will focus on two of them in particular :

- **W-destructible points** that belong to a shed and can be lowered;
- **M-destructible points** that belong to a basin and can be lowered too;
- **W-constructible points** that are neither M-destructible, neither W-destructible.

Moreover, the following algorithms can indicate if a point is M-destructible or W-destructible but if they are, they also give the component at which they can be attached (and therefore the height at which the point can be lowered).

2.5.2 Preliminary

Given two points $p, q \in I$, we say that p and q are k-separated in I if they belong to the same component in I_k and to different components in I_{k-1}. We can also write $I(p,q) = k$. This operation is algorithmically equivalent to a highest fork.



Component mapping

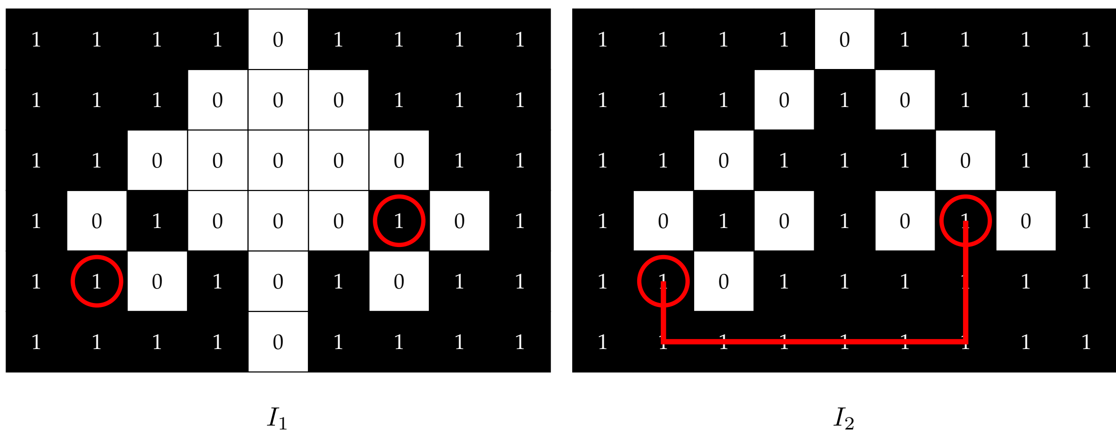


Figure 2.5: Example with points in a and d: they belong to separated components in I_2 and to the same component in I_1. They are 2-separated, which is also the height of their highest fork (f).

2.5.3 W-destructible Points

A point p is W-destructible if, for any q and r neighbours of p, $I(q,r) < I(p)$.

Our function not only returns whether a point is W-destructible or not but also the component to which the point should be attached.

```
1  component W-destructible (point p)
2  begin
3    set_of_points V ← neighbours of p lower than p
4    if (V is empty)
5      return not_W-destructible
6
7    point hf = highest_fork(V)
8
9    if (not (hf exists))
10     return component(min (V))
11
12   if (hf.height < p.height)
13     return component(hf)
14
15   return not_W-destructible
16 end
```

Interpretation. As we can see in the algorithm, there are two types of W-destructible points:

- **crest:** if the highest fork exists and is lower than p, then p is on a watershed line and can be lowered to the value of the lowest crest around it.

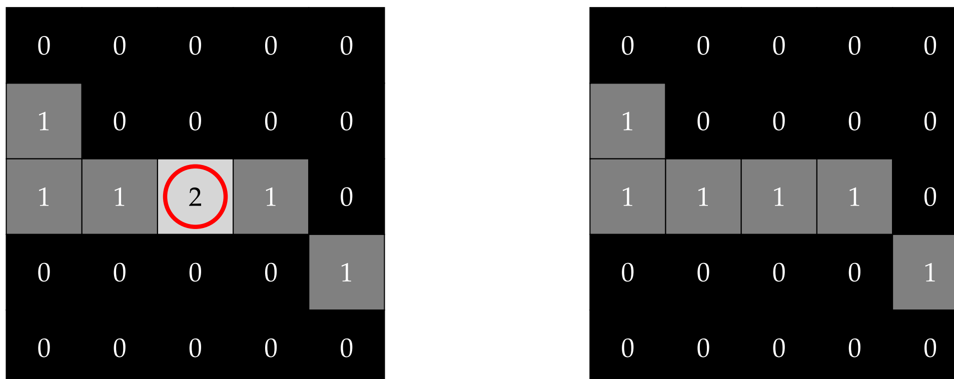


Figure 2.6: The point is W-destructible. Its neighbours are 1-separated, so it is lowered to 1.

- **gradient:** if the highest fork does not exist, p is in a slope and can be lowered to the value of the lowest of its neighbours.

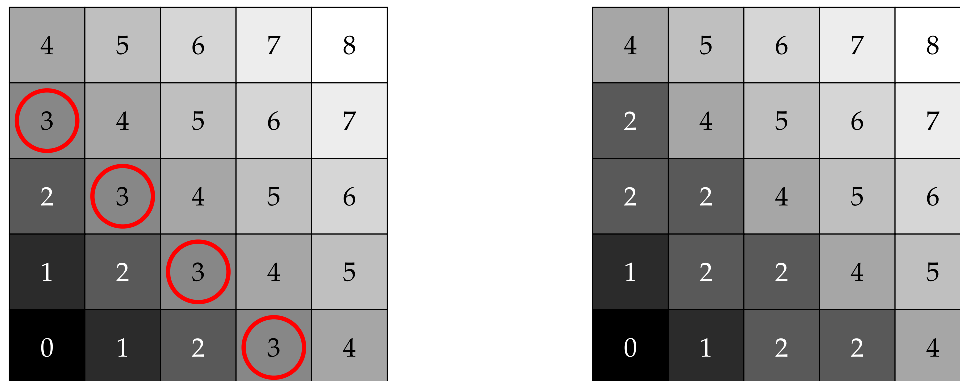


Figure 2.7: These points are W-destructibles. Their lowest neighbours are at 2, so they are lowered to 2.

One can see here that lowering the points in 2.7 is pointless because the will be lowered again to 0 further on in the algorithm. In order to avoid this behaviour, we want a stronger condition on our points in order to lower each point once and for all.

2.5.4 M-destructible points

An M-destructible point is a W-destructible point which becomes part of a minimum when lowered. The interest of this characterisation is obviously to spot the points which can be lowered once and for all. Let's take our previous example and see the M-destructible points.

```
1  component M-destructible (point p)
2  begin
3    set of points V ← neighbours of p lower than p
4
5    if (V is empty)
6      return not_m-destructible
7
8    if (min(v) is not a leaf)
9      return not_m-destructible
10
11   point hf = highest_fork(V)
12
13   if (not (hf exists))
14     return component(min (V))
15
16   return not_m-destructible
17 end
```

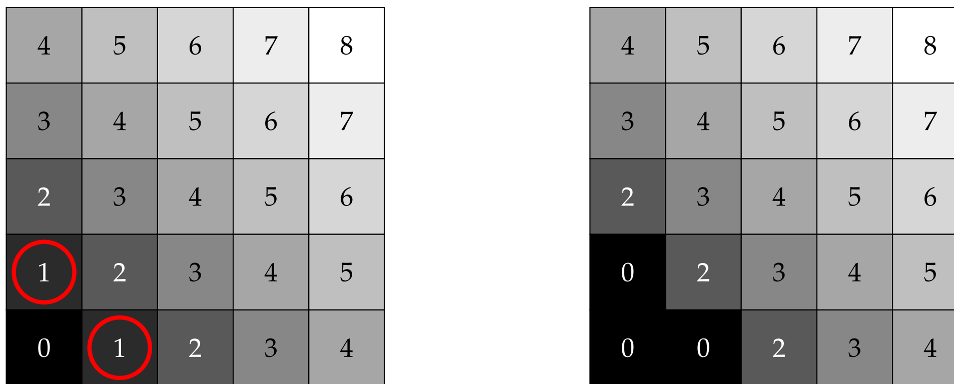


Figure 2.8: These points are M-destructibles. Their lowest neighbours are at 0, so they are lowered to 0. They can't be lowered anymore.

2.5.5 W-constructible

A W-constructible point is a point which is neither W-destructible, neither M-destructible. The algorithm differs from the previous ones because it should be used on an inverted image.

```
1  component W-constructible (point p)
2  begin
3    set of points V ← neighbours of p greater than p
4
5    if (V is empty)
6      return not_w-constructible
7
8    if (size(V) == 1)
9      return V[1]
10
11   point hf = highest_fork(V)
12
13   if (hf.height ≤ p.height)
14     return not_w-constructible
15
16   return hf
17 end
```

2.6 M-watershed

The M-Watershed is a full-fledged watershed and is the first step of the topological watershed : it prepares the image for the W-watershed. Indeed, the W-Watershed tries to lower W-destructible points, but, as seen before, once that some points have been lowered, some other points become W-destructible according to the new image and need to be lowered again, these points are M-destructible points.

This watershed's goal is to remove all M-destructible points. Since a point needs to have a minimum near it in order to be M-destructible, the algorithm takes image minima as starting points. To process all points in importance order, the points are enqueued in a priority queue depending on their height. Then they are all lowered one by one until the algorithm reach the crests.

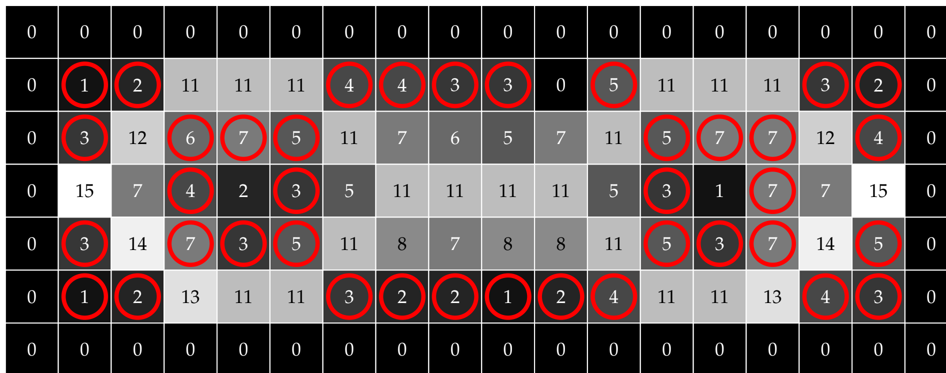
A strong property of this algorithm is given in Couprie et al. (2005):

> **Property 13** *Whatever the chosen priority function, the output of Procedure* **M-watershed** *is an M-watershed of the input.*

This means that the user could change the priority function if he wants to get another watershed.

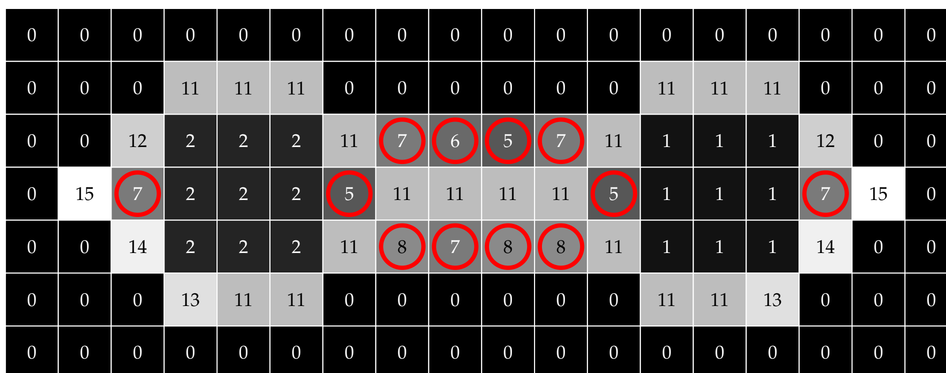
Let us now see the pseudo-code of this algorithm. Then a step by step example is given, on a more explicit image than the one used before.

```
1  M-Watershed (image I,
2                component mapping C)
3  begin
4    priority queue L
5
6    for all p in I
7      c ← M-destructible(p)
8      if (c ≠ not_m-destructible)
9        L.enqueue(p)
10       mark(p)
11
12   while (L is not empty)
13     p ← L.pop()
14     c ← M-destructible(p)
15     if (c ≠ not_m-destructible)
16       I(p) ← height(c)
17       C(p) ← c
18       for all neighbours q of p
19         if (q is not marked)
20         c ← M-destructible(q)
21         if (c ≠ not_m-destructible)
22           L.enqueue(q)
23           mark(q)
24 end
```

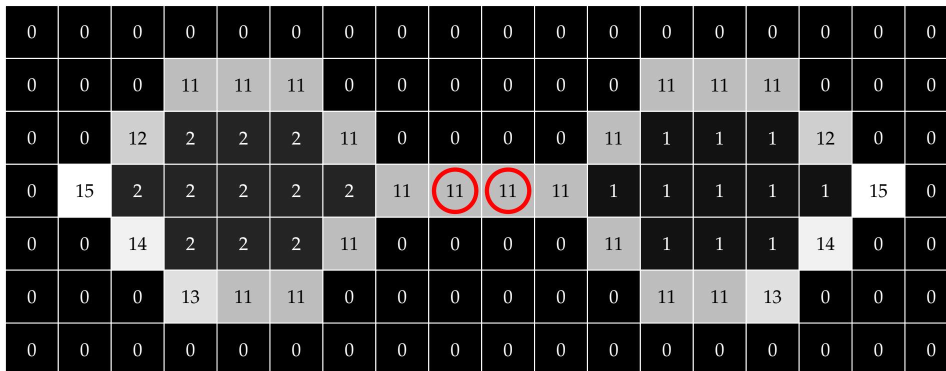


0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	11	11	11	4	4	3	3	0	5	11	11	11	3	2	0
0	3	12	6	7	5	11	7	6	5	7	11	5	7	7	12	4	0
0	15	7	4	2	3	5	11	11	11	11	5	3	1	7	7	15	0
0	3	14	7	3	5	11	8	7	8	8	11	5	3	7	14	5	0
0	1	2	13	11	11	3	2	2	1	2	4	11	11	13	4	3	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Circled points are M-destructible and will be lowered

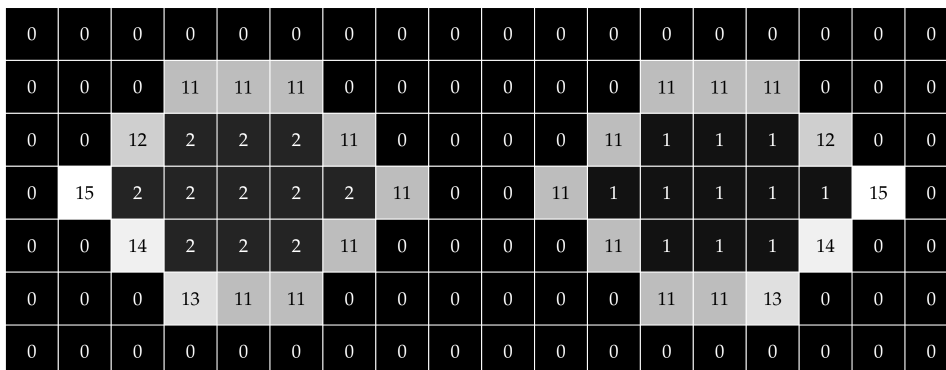


0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	12	2	2	2	11	7	6	5	7	11	1	1	1	12	0	0
0	15	7	2	2	2	5	11	11	11	11	5	1	1	1	7	15	0
0	0	14	2	2	2	11	8	7	8	8	11	1	1	1	14	0	0
0	0	0	13	11	11	0	0	0	0	0	0	11	11	13	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	12	2	2	2	11	0	0	0	0	11	1	1	1	12	0	0
0	15	2	2	2	2	2	11	11	11	11	1	1	1	1	1	15	0
0	0	14	2	2	2	11	0	0	0	0	11	1	1	1	14	0	0
0	0	0	13	11	11	0	0	0	0	0	0	11	11	13	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This reduction can seem unnatural but the 0s on each side of the points belong to the same component.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	12	2	2	2	11	0	0	0	0	11	1	1	1	12	0	0
0	15	2	2	2	2	2	11	0	0	11	1	1	1	1	1	15	0
0	0	14	2	2	2	11	0	0	0	0	11	1	1	1	14	0	0
0	0	0	13	11	11	0	0	0	0	0	0	11	11	13	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Final result

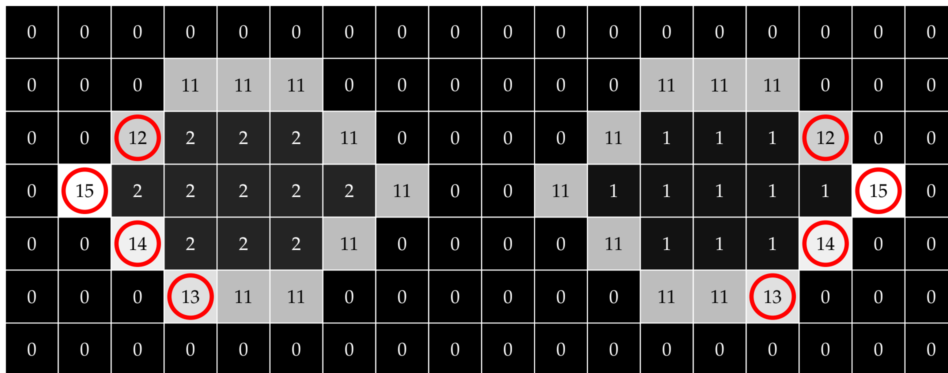
Figure 2.9: M-Watershed step by step

2.7 W-Watershed

The W-Watershed, also called W-thining, is the second step of the topological watershed. It will flatten the edges so that the height of the watershed line between two components will become the altitude of the minimal crest between the two components.

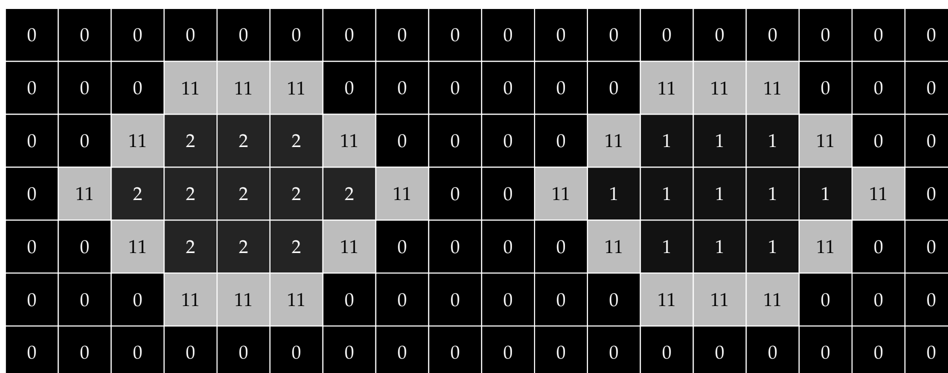
In order to lower W-destructible points, we run through the picture by levels in ascending order and lower the W-destructible points. The algorithm has a little optimization: every time a point is lowered, we look at its neighbours to see if another point is W-destructible (through the variables K and H in the algorithm). So each time a point is lowered, we know that it will not be lowered anymore. In our example, all W-points are lowered in the same loop thanks to this mechanism.

```
1  W-Watershed(image I,
2              component mapping C)
3  begin
4    image K;
5    component mapping H;
6
7    array of sets of points L[number of levels in I];
8
9    for all p in I
10     c ← W-Destructible(p)
11     if (c ≠ not_w-destructible)
12       L[height(c)].insert(p)
13       K(p) ← height(c)
14       H(p) ← c
15
16   for all levels k of I in ascending order
17     while (L[k] is not empty)
18       p = L[k].extract_first()
19       if (K(p) == k)
20         I(p) ← k
21         C(p) ← H(p)
22
23         for all neighbours q of p
24           if (k < I(q))
25             c ← W-Destructible(q)
26             if (c == not_w-destructible)
27               K(q) ← \infinity
28             else
29               if K(q) ≠ height(c)
30                 L[height(c)].insert(q)
31                 K(q) ← height(c)
32                 H(q) ← c
33 end
```



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	12	2	2	2	11	0	0	0	0	11	1	1	1	12	0	0
0	15	2	2	2	2	2	11	0	0	11	1	1	1	1	1	15	0
0	0	14	2	2	2	11	0	0	0	0	11	1	1	1	14	0	0
0	0	0	13	11	11	0	0	0	0	0	0	11	11	13	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Circled nodes are W-destructible points



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	11	2	2	2	11	0	0	0	0	11	1	1	1	11	0	0
0	11	2	2	2	2	2	11	0	0	11	1	1	1	1	1	11	0
0	0	11	2	2	2	11	0	0	0	0	11	1	1	1	11	0	0
0	0	0	11	11	11	0	0	0	0	0	0	11	11	11	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Final result

Figure 2.10: W-Watershed step by step

2.8 Topological Watershed

This algorithm is a merge of the algorithms described above. This is obviously faster but one can't use a customized priority function as in the M-watershed, but as a previous step, it is needed to invert the colors of the image.

```
1 Topo-Watershed (image I,
2                 component mapping C)
3 begin
4   priority queue L
5
6   for all p in I
7     if (C(p) is a leaf)
8       mark p // p is fixed
9
10  for all p in I
11    if (p is marked)
12      enqueue neighbours of p
```

```
13
14   while (L is not empty)
15     p ← L.pop()
16     c ← W-constructible(p)
17     if (c ≠ not_W-constructible)
18       I(p) ← height(c)
19       C(p) ← c
20       if (C(c) is a leaf)
21         mark c
22
23       for all neighbours q of p
24         if (q is not marked and not enqueued)
25           L.enqueue(q)
26 end
```

Chapter 3

TikZ Export

3.1 The Need for an Image Output

Milena is able to manipulate many image types including exotic ones like graph images. Unfortunately, these unusual types cannot be visualized simply. Therefore we need a tool to visualize such images but also to integrate them in reports since their purpose is highly theoretical for the moment. A typical example would be graph images which can be valuated on the nodes, on the edges or on both.

We want to:

- visualize **clearly** various but little images;
- tag images with their values and chosen colors;
- modify generated images after resizing, add annotations...

3.2 From Milena to LaTeX

After a little overview of all existing tools, the PGF LaTeX extension seems to meet all our requirements. It is easy to use thanks to its little programming languages (TikZ), it is easy to customize, integrate in a LaTeX report or to output it in an eps file for an external use.

For the moment, only greyscale or rgb 2D images (and compatible) can be displayed. For other color types, the easiest way is to provide a conversion function to RGB or Greyscale. For other image types, we can imagine a conversion toward 2D coordinates or using the gnuplot compatibility of pgf for 3D images for example.

The only problem encountered to convert Milena images to TikZ was the color format.

3.3 Tikz Color Conversion Algorithm

In TikZ, color rendering is made through the xcolor extension. This extension allows color definitions in multiple formats: RGB, CMYK ...
Colors can also be aliased thanks to the `definecolor` command:

```
\definecolor{alias}{value}
```

Or their color can be defined directly in a TikZ structure:

```
node [color=red]
```

Unfortunately, using definecolor is a pain for TEX capacity and further human reading. Plus, for an obscure reason, it is impossible to use anything else than the mix color format to define colors in TikZ. The most simple workaround for this issue was to convert RGB colors to the color mix format.

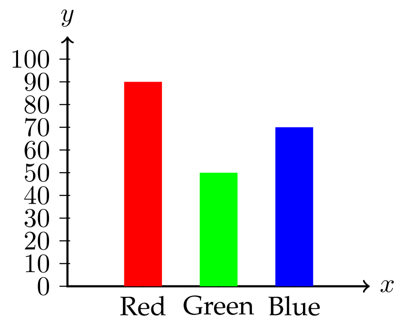
The color mix format is defined as follows (C and C' are predefined names of colors):

$$
\begin{aligned}
&\mid C \\
&\mid C\ !\ p\ !\ C' = p \times C + (1-p) \times C'
\end{aligned}
$$

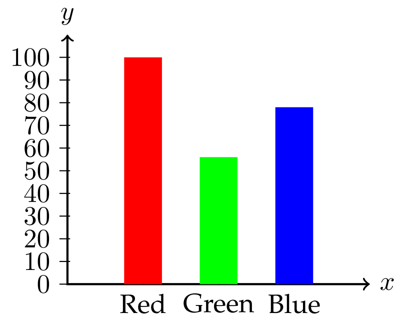
This unusual format is obviously not supported by Milena but a naive algorithm has been implemented to convert RGB value to it.

This algorithm consists in the following simple steps:

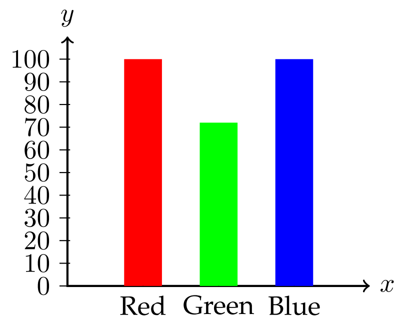
- compute the percentage of black in the color
- normalize the image by removing black
- compute the percentage of the max color
- normalize the image by removing the max color
- compute the percentage of the second color
- normalize the image by removing the last color
- compute the percentage of the last color



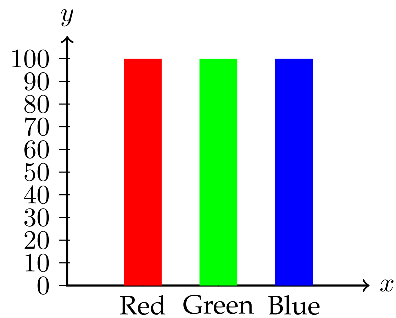
TikZ color code:
`black!10`



TikZ color code:
`black!10!red!22`



TikZ color code:
`black!10!red!22!-green!28`



TikZ color code:
`black!10!red!22!-green!28!white`

Figure 3.1: The RGB to TikZ algorithm step by step

3.4 Examples

All images displayed in this paper have been generated by Milena in TikZ and here is an example of a color image : 3.2.

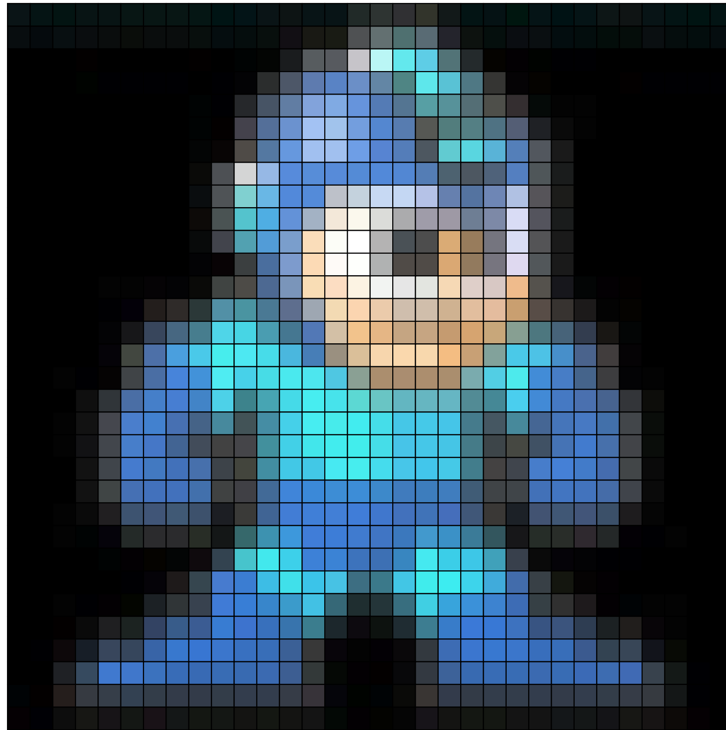


Figure 3.2: TikZ generated colored image

Chapter 4

Conclusion

The implemented algorithm is functional and gives images that fit exactly the requirements. There is still some work left to optimize the transformation by speeding up inner algorithms. A region merging algorithm can also be implemented to increase the interest of the algorithm. A variant of the algorithm can also be done to work on graph image and on coloured ones.

As seen is the state of art, this algorithm is not widespread. This is a real advantage compared to other image processing library, notably in term of efficiency because such an algorithm can be costly. With our quasi-linear implementation, it will be easy to process large images, making Milena a true competitor in this domain.

The TikZ export function can be enhanced in order to support more image types. Many options could be added (like tue ability to highlight some points) in order to make it more user friendly.

Chapter 5

Bibliography

Bertrand, G. (2005). On topological watersheds. *Journal of Mathematical Imaging and Vision*.

Couprie, M., Najman, L., and Bertrand, G. (2005). Quasi-linear algorithms for the topological watershed. *Journal of Mathematical Imaging and Vision*.

D. Harel, R. T. (1984). Fast algorithms for finding nearest common ancestors. *SIAM J Comput.*

M.A. Bender, M. F.-C. (2000). The lca problem revisited. *Proc. 4þ Latin American Symposium on Theoretical Informatics*.

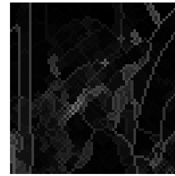
Najman, L. and Couprie, M. (2003). Watershed algorithms and contrast preservation. *DGCI*.

Appendix A

Example

Original image of Lena

Meyer's Watershed



Topological Watershed