Milena

EPITA Research and Development Laboratory (LRDE)

October 2008

Outline

Outline

Outline

Outline

Outline

What is Olena?

Olena is the name for

- the project of building some modern image processing tools
- the platform, including
  - a library
  - command line executables
  - some documentation
  - etc.

Milena

Milena is the C++ image processing[a] library of Olena.

[a] In the following, IP is “Image Processing” for short.

Yet Another Image Processing Library (YAIPL) ?

Yes!

- Many libraries exist that can fulfill one's needs.
- If you're happy with your favorite tool, we cannot force you to change for Milena...
- Though, you might have a look at Milena and be seduced!

No!

- Milena is rather different than available libraries.
- A lot of convenient data structures that really help you in developing IP solutions.

A Short History of the Olena Project

- 2000: Start of the project.
- From Nov. 2001 to April 2004: Evolution from version 0.1 to 0.10. The level of genericity we expected from the lib was partially obtained...
- February 2007: Update to conform modern C++ compilers = version 0.11.

During those 3 years we developed a prototype to experiment with genericity and to try to meet our objectives.

- From June 2007 up to now: Re-writing of the library with a programming paradigm that rocks.

Outline

What's In a Library

- algorithms

  procedures dedicated to image processing and pattern recognition

- data types for pixel values

  gray level types with different quantizations, several floating types, color types

- data structures

  for instance, many ways to define images and sets of points

- a lot of auxiliary tools

  they help to easily write readable algorithms in a concise way!

Objectives of Milena as a Feature List

efficiency	ready to intensive computation (large data / sets of data)
genericity	not limited to very few types of values and images
reusability	software blocks are provided for general purpose
composability	coherency of tools ensure software building from blocks
simplicity	as easy to use as a C or Java library
safety	errors are pointed out at compile-time, otherwise at run-time

Getting at the same time genericity with efficiency and simplicity is very challenging.

Definition

A generic algorithm

- is written once (without duplicates)
- works on different kind of input

The Most Dummy Example

Filling an image ima with the value v:

```
// Java or C -like code

void fill(image* ima, unsigned char v)
{
  for (int i = 0; i < ima->nrows; ++i)
    for (int j = 0; j < ima->ncols; ++j)
      ima->data[i][j] = v;
}
```

Note that we really have here an example very representative of an algorithm and of many pieces of existing code.

Comments

There are a lot of implicit assumptions about the input:

- the input image has to be 2D;
- moreover its definition domain has to be a rectangle...

  ...starting at (0,0);
- data cannot be of a different type than “unsigned char”;
- image data need to be stored as a 2D array in RAM.

For instance this routine cannot work on a region of interest of a 2D image having floating values.

Comments

Actually these are constraints that limit the applicability of this routine.

This routine is definitely not generic.

Furthermore, those implementation details appearing in code obfuscate the “actual” algorithm.

Generic algorithm translation

Algorithm:

Procedure **fill**
  ima : an image (type: any type I)
  v : a value (type: value type of I)
begin
  for all p in ima domain
    ima(p) ← v
end

// Milena code:

```
template <typename I>
void fill( I& ima,
           mln_value(I) v )
{
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

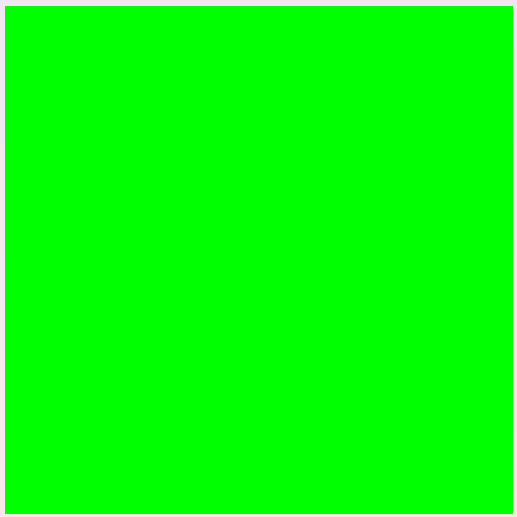
Example

The basic (common) run:

```
fill(lena, literal::green);
```

before:

after:



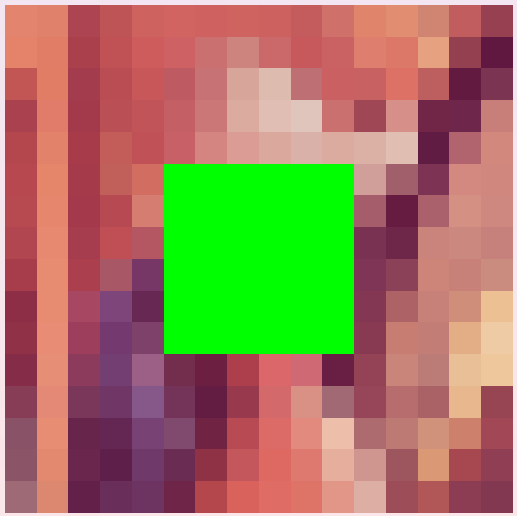
Example

Filling only a region of interest (a set of points):

```
box2d roi(5,5, 10,10);
fill(input | roi, green);
```

before:

after:



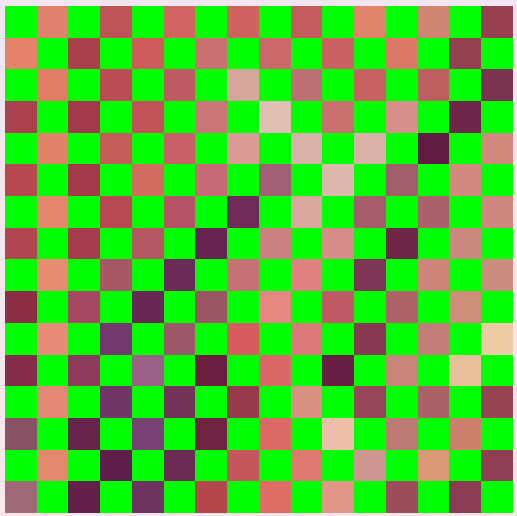
Example

Filling only points verifying a predicate:

```
fill(input | chess, green);
```

before:

after:



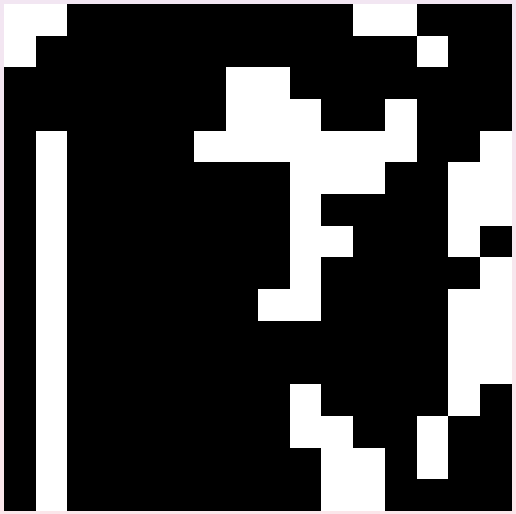
Example

Likewise, the predicate being a mask image:

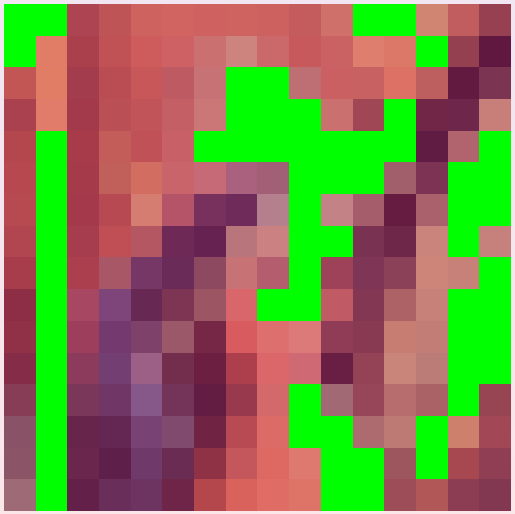
```
fill(input | pw::value(mask), green);
```

before:

mask:



after:



Example

Likewise, relying on an image of labels:

```
fill(input | (pw::value(label) == 3), green);
```

before:

label:

after:

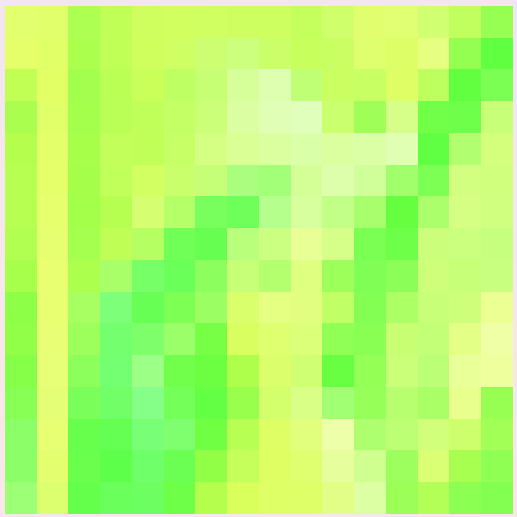
Example

Filling only a component:

```
fill(fun::green(input), literal::max);
```

before:

after:



Example

Mixing several “image views”:

```
mln_VAR(object_3, pw::value(label) == 3);
fill(fun::green(input) | object_3, literal::max);
```

before:

label:

after:

About the Former Example

```
// Milena code (actually running):

template <typename I>
void fill( I& ima,
           mln_value(I) v )
{
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

```
// equivalent "more classical" code
// that you will never have to write:

void fill( image2d_rgb_3x8* ima,
           image2d_unsigned& label,
           unsigned char& v )
{
  for (unsigned row = 0; row < ima->nrows; ++row)
    for (unsigned col = 0; col < ima->ncols; ++col)
      if (label->data[row][col] == 3)
        ima->data[row][col].green = v;
}
```

Conclusion about Genericity

The generity applies on:

- structures of images
- values of images
- neighborhoods
- etc.

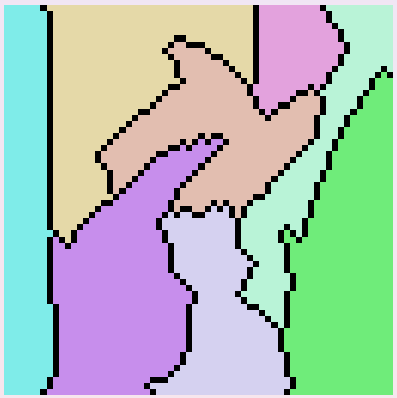
and

- algorithms are written once...
- ...even complex ones

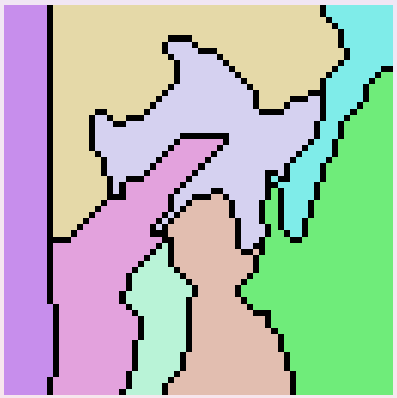
When Different Neighborhoods

input:

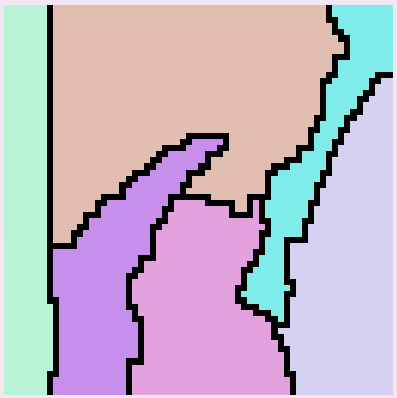
with c4:



with c6:



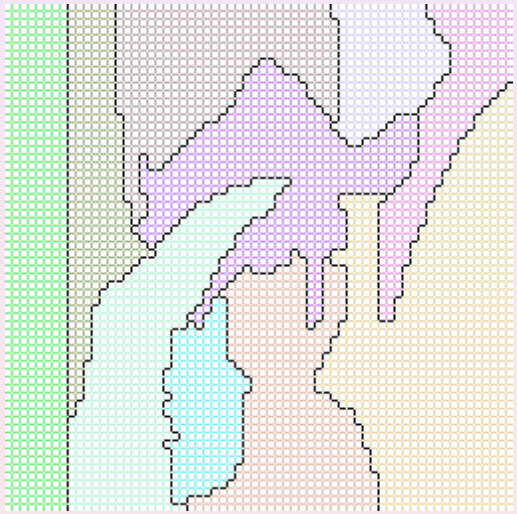
with c8:



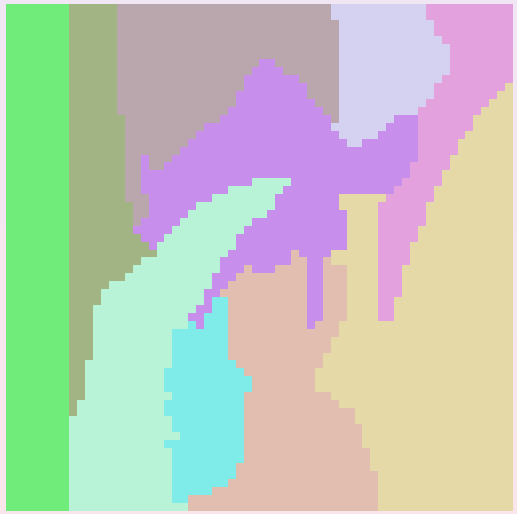
On Edges

gradient:

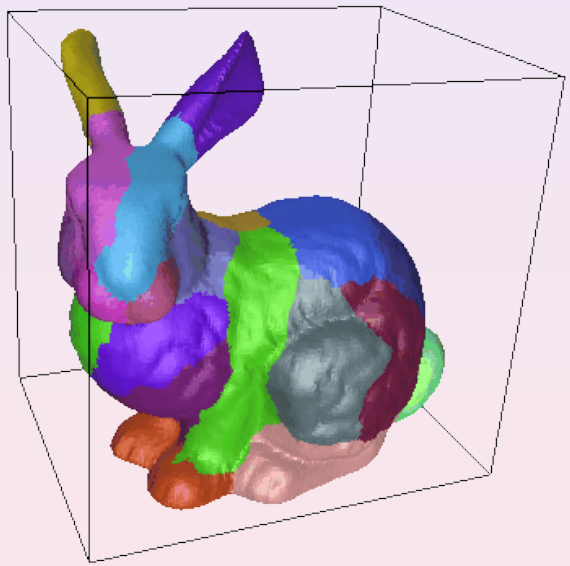
watershed:



output:

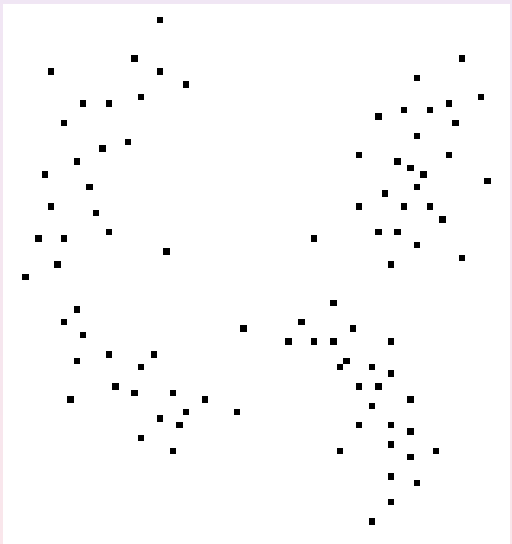


On Surface Edges

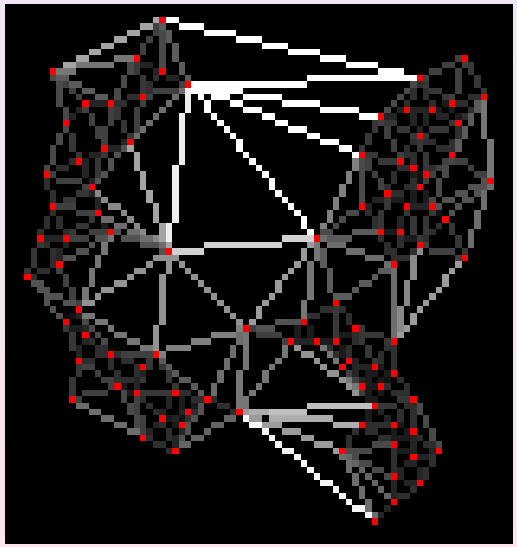


Input

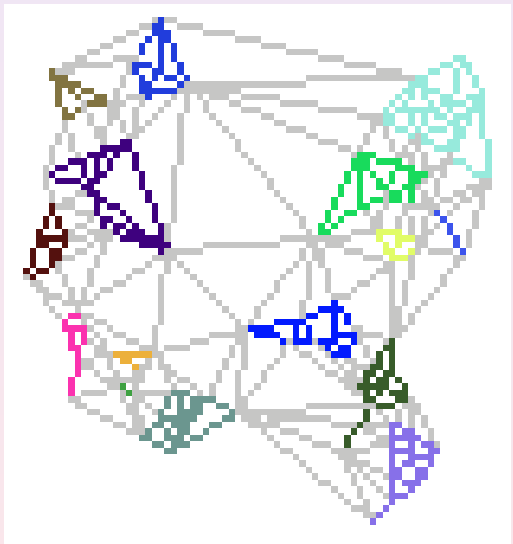
vectors:



distance:

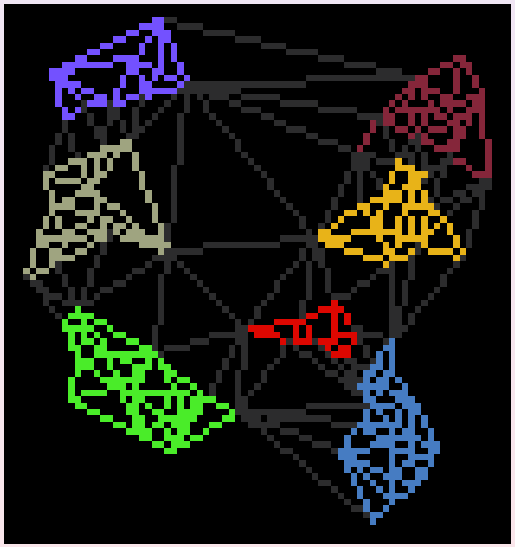


regional minima:

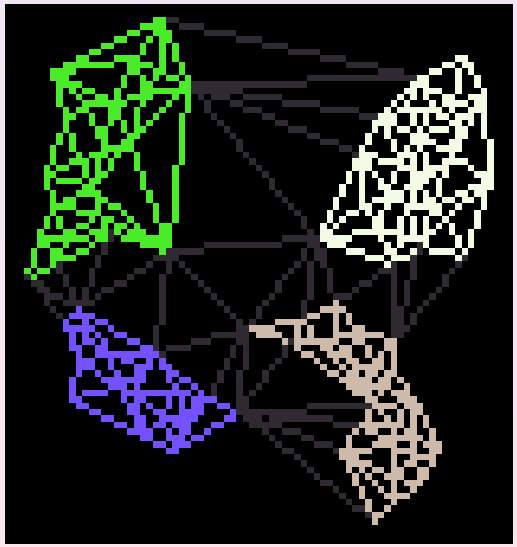


Output

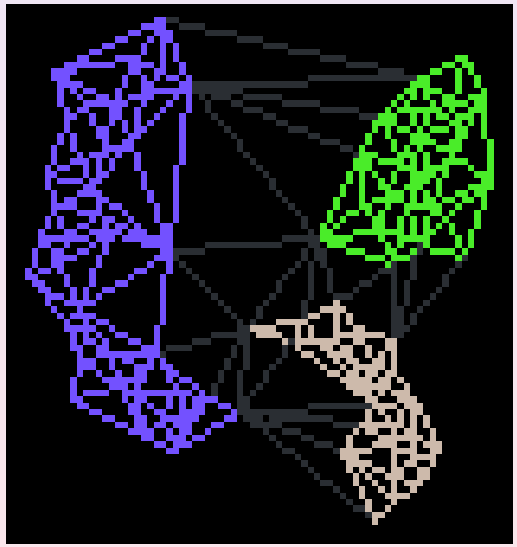
light:



medium:

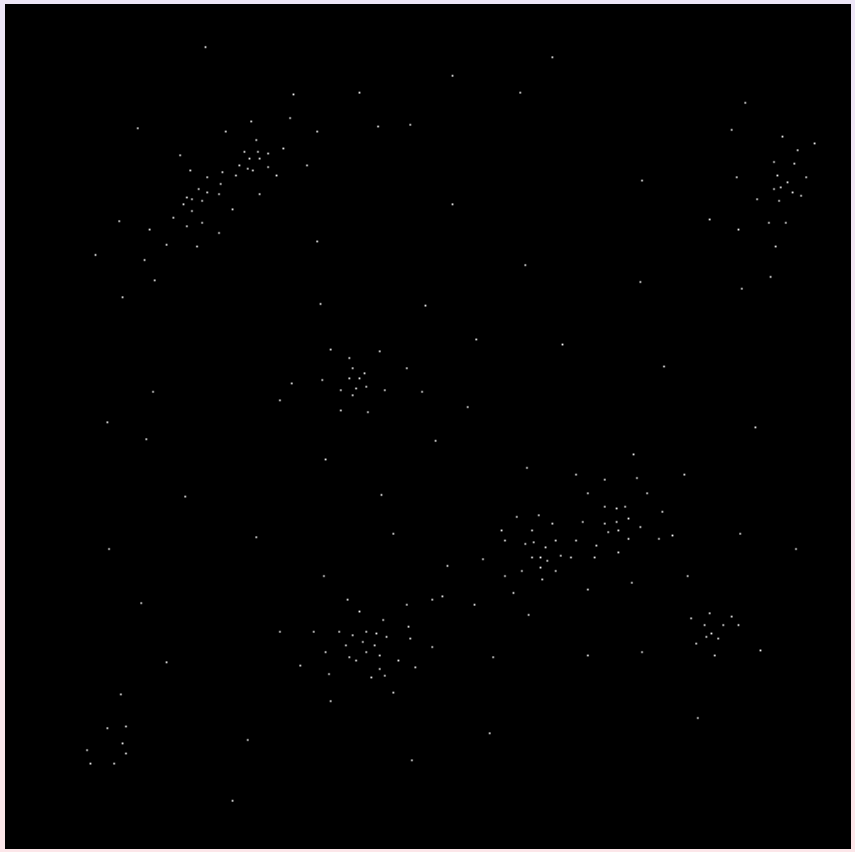


strong filtering:

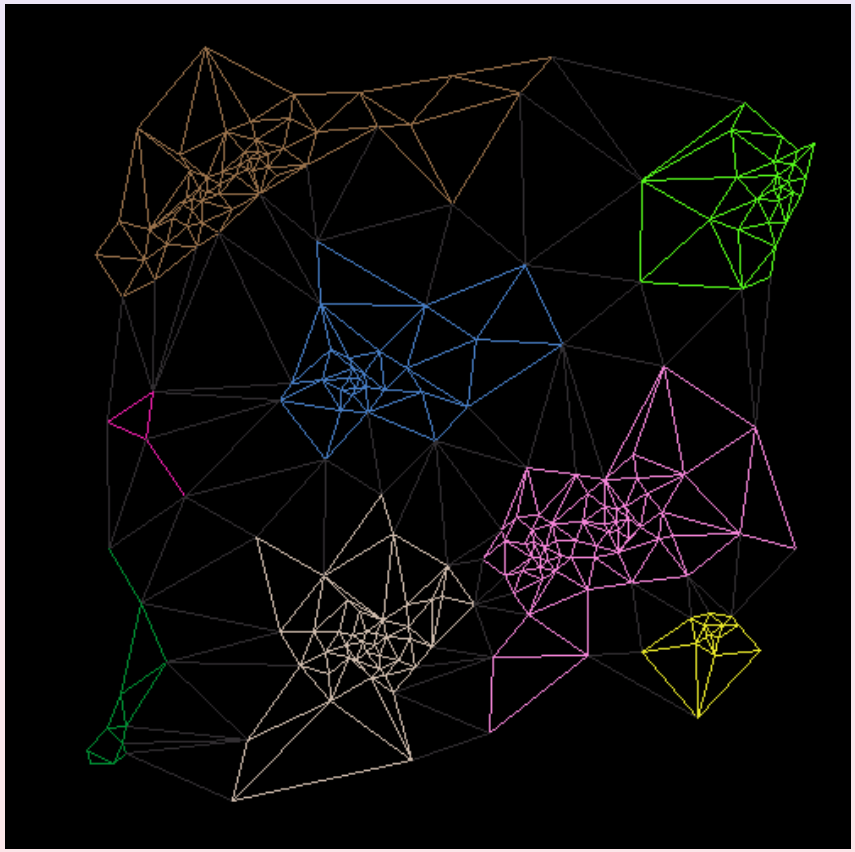


Output

population:



result:



Always the Same Code Running!

```
template <typename I, typename N>
mln_ch_value(I, rgb8)
segmentation(const I& ima, const N& nbh, unsigned area)
{
  mln_concrete(I) filtered = morpho::closing_area(ima, nbh, area);

  unsigned nbasins;
  mln_ch_value(I, unsigned) wst = morpho::wst_meyer(filtered, nbh, nbasins);

  return level::transform(wst, colorize(nbasins));
}
```