Tiny Image in JavaScript : Mathematical Morphology : Euclidean Distance Map

Master 2 Bioinformatics
Group : Vincent DEJONGHE, Adrien MENDES-SANTOS and Rodolphe TWOREK
Subgroup :Vincent DEJONGHE
Delivery date : 02/22/2018
GitHub link : `https://github.com/ReiNoIkari/Morphological-Image-Processing`

1 Introduction :

Nowadays, scientists have access to huge image databases which can make studying organisms easier. One problem still persist : those collected data are worthless if they are not interpreted and the concern is that a lifetime won't be enough for a scientist to follow the flow. That's were informatics come in order to help to analyse those huge amount of raw data, and in our case more precisely Image Processing. Indeed, the developement of some Image processing tool made that easier to computerize and analyse [1],[2].

Mathematical morphology has been invented in 1964 by Georges Matheron and Jean Serra in the MINES ParisTech's laboratories. Its development was always motivated by industrial application. At the begining, the main purpose was to answer issues in the mining exploitation field. Then this purpose diversified itself to biology, medical imagery, material science, industrial vision, multimedia, teledetection, geophisic, etc. It consist in a mathematical and informatical theory and technique which is linked with algebra, the lattice theory, topology and probabilities SCH1993 .

Currently, one of the mathematical morphology's main field is Image Processing. It particularly allows to use filtering, segmentation and quantification tools. Since it's emergence in 1964, it knows a growing success and constitutes a part of many Image Processing softwares yet.

For the purposes of object identification required in industrial vision applications, the operations of mathematical morphology are more useful than the convolution operations employed in signal processing because the morphological operators relate directly to shape.

For the third part of our project, our group was split in order to implement two morphological operators : skeletonize and Euclidean Distance Map (EDM). This report focus on the latter.

Firstly, the Euclidean Distance Map principal will be introduced. Then it's implementation from the given Java source code in JavaScript and functional JavaScript will be presented. Then, performances comparaisons will be made and our results will be described. The function using gpu acceleration could not be implemented due to lack of time.

2 Method :

Distance maps can be used for several purposes. Among these are expansion/shrinking of the objects S (by thresholding L(S)/L(S)), construction of shortest paths between points, skeletonizing and shape factor computation [3].

the EDM throughout the case of binary images, background pixels are distinguished from the object's pixels. Thus, a map of the original picture is made. This map indicates the shortest distance to the nearest opposite pixel (background in the case of an object to background approach and object in the case of a background to object one) for each pixel of the original binary picture. This distance is measured with the Euclidean relation.

Let (E, d) be a metric space and S⊂E. To each element x of E is associated the DMS(x) value defined by :

$$\mathrm{DM}_S^d(x) = \min_{y \notin S} d(x, y)$$

The set of all DMS(x) for all x of E is called the DMS distance map of S.

2.1 Algorithm

Our EDM function has been implemented in JavaScript after being inspired by the *makeFloatEDM* method (findable at the following link : `https://github.com/imagej/imagej1/blob/master/ij/plugin/filter/EDM.java`) and takes three main parameters as input: an image (or in our case, an image's raster), the background value (set to 0 by default) and the EdgesAreBackground parameter that is used to determine if the "out-of-image pixels" are considered with background values (set to false by default).

Moreover, this function comports two subfunctions. Firstly, the *edmLine* subfunction handles a line which can be traveled in two different directions: from the left to the right, and the reverse. Then the *minDist2* function calculates the squared minimum distance between two points x and y using the nearest point found for the same line and same column, the nearest point found for the same line and previous column, and the nearest point found for diagonal (previous line, previous column). If the *distSqr* value (which represents the distance from an edge) parameter is lower than the squared distance, it is used. this function returns the minimum squared distance obtained. The global EDM function returns a modified raster for whom the pixels values are representative from the EDM mathematical morphology.

2.2 Benchmarks

Benchmarks were set up in order to compare ImageJ and JavaScript implementations efficiencies. To do so, a warmup phase of 100 image's computations is firstly executed for each implementation. Then, for each implementation, and for different image sizes, 1000 computation per image are do, and the global execution time for each size in milliseconds (ms) is returned. We created 6 different sizes of images by using the *concat* method from JavaScript.

3 Results :

the following section shows what kind of results our implementation gives by processing a binary image. The image used is a sample available in ImageJ : *Blobs* (sized 256x254).

3.1 Algorithm

The Euclidean Distance Map morphology made using ImageJ and its *Distance Map* function on a binary image (*Blobs (256x254)*) is shown below :

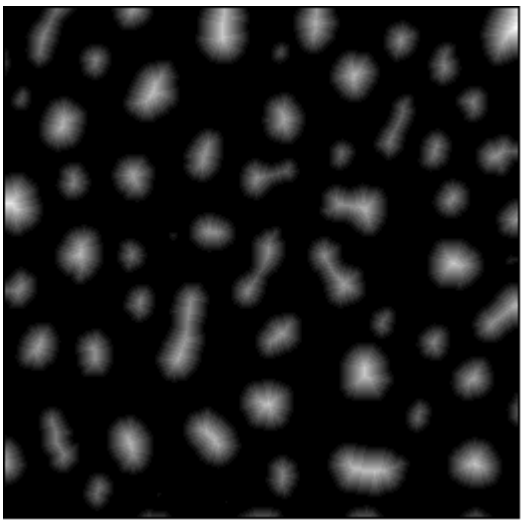

Figure 1: EDM made with ImageJ on the Blobs' binary image

The resulting map after a Java to JavaScript code translation can be found as follows :

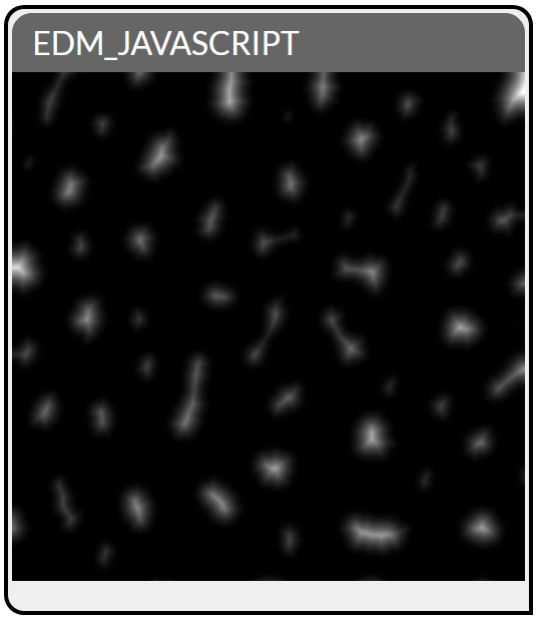

Figure 2: EDM made resulting from the translation from Java to JavaScript on the Blobs' binary image

Finally, the EDM made with functional JavaScript is shown below

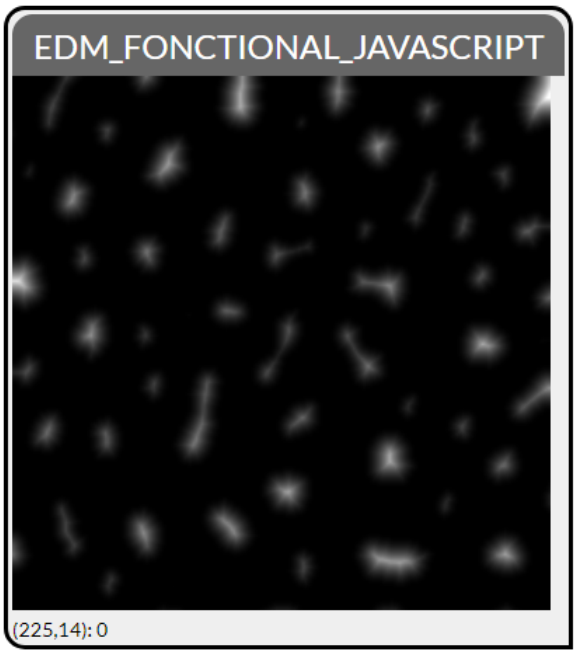

Figure 3: EDM made resulting from the functional JavaScript on the Blobs' binary image

3.2 Benchmarks

As told earlier, we used the *concat* method in order to shape our different sizes. Thus, a *x1 sized* image corresponds to 256x254 pixels, when a *x2 sized* corresponds to a 256x508 pixels, and so on. Following Benchmarks results can be found as follows :

Image size	JavaScript	Fonctional JavaScript	ImageJ
x1 sized	4,847565	14,545665	21,36693
x2 sized	14,091755	45,02358	23,71532
x3 sized	28,508885	90,09537	24,30515
X4 sized	47,803635	149,53868	23,91521
X5 sized	112,50133	270,997305	30,26359
X6 sized	184,57502	411,30332	34,22732

Figure 4: Results of the benchmarks in ms for the JavaScript, the functional JavaScript and the ImageJ implementation of the EDM algorithm

Graphically, those results can be shown has following in order to visualize the evolution of processing time :

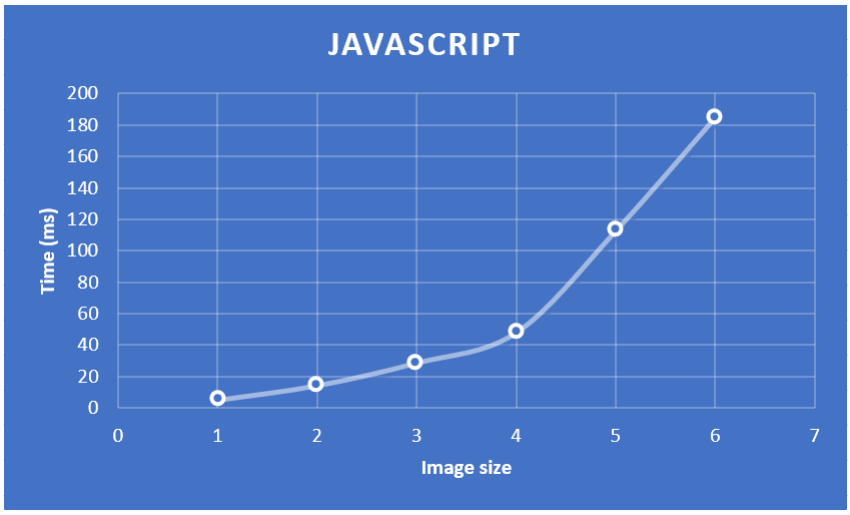

Figure 5: Graphical representation of the Javascript's Benchmark

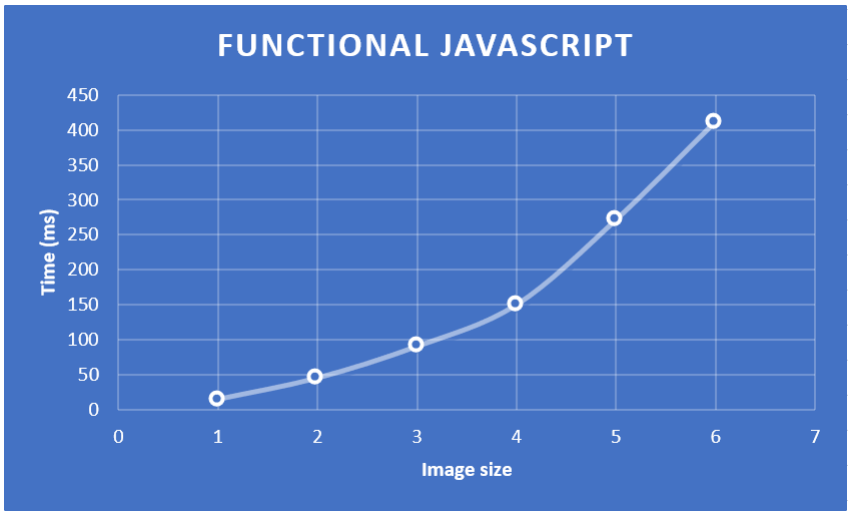

Figure 6: Graphical representation of the Functional JavaScript's Benchmark

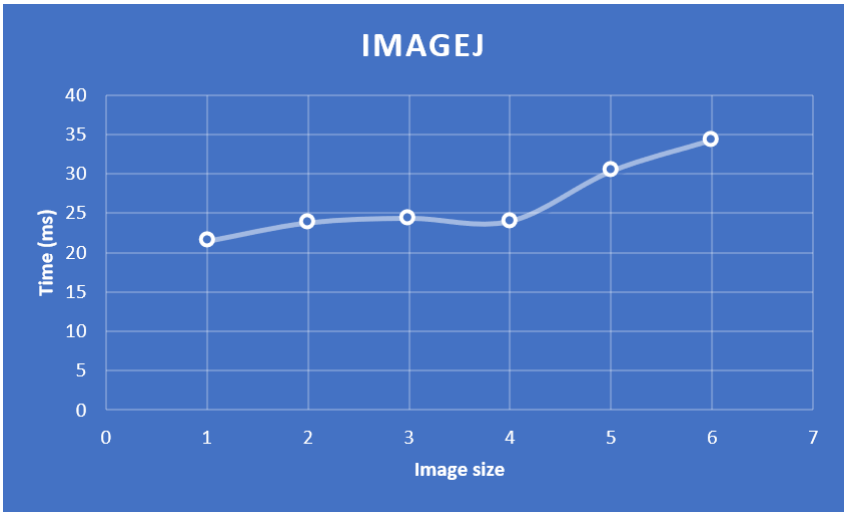

Figure 7: Graphical representation of the ImageJ's Benchmark

At least, those results can be merged in a comparison purpose :

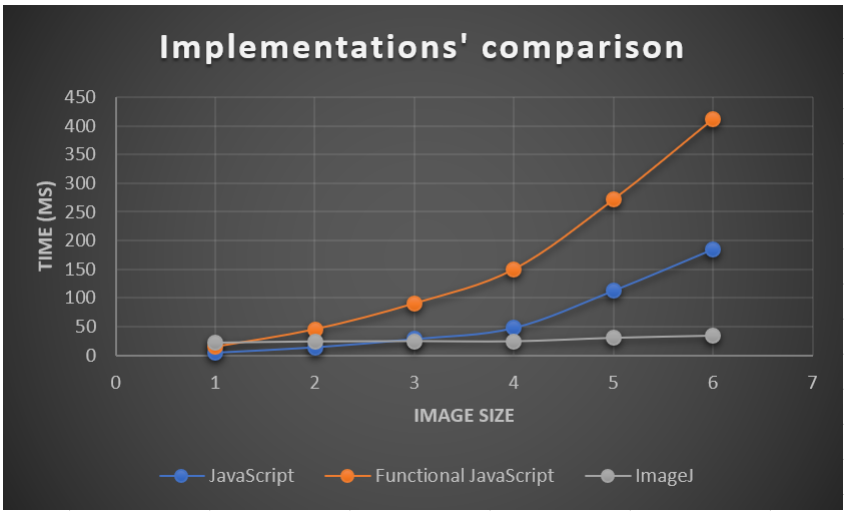

Figure 8: Comparison of the Benchmarks' result for each implementation

4 Discussion

According to the three first figures, the general output between the two JavaScripts implementations and the ImageJ given one seems to looks globally similar on the same binary image. Indeed this output shows a light gradient higher in the center of the blobs' cells and going darker and darker by approaching the background. However, it is to notice that the ImageJ's output seems to consider a higher number of the blobs cell's pixel with its *makeFloatEDM*.

However, according to our benchmarks, whether JavaScripts functions seems to process more efficiently smaller images than ImageJ (4.847557 ms and 14.54567 ms versus 21.36693 ms for a 256x254 pixeled image), ImageJ seems to overcome them for higher sizes (overcoming functional JavaScript from a double sized image and simple JavaScript from a triple sized one).

ImageJ also seems more stable (processing from 21,36693 ms for a 256x254 image to 34,22732 ms for a 254x1524 one versus 4,84757 ms to 184,57502 ms for JavaScript and 14,54567 ms to 411,30332).

5 Conclusion :

During this project, we developed one or more image processing features in a classical image toolbox. We also enhanced our skills in JavaScript programming and to were introduced to an other paradigms : functional programming. Furthermore, this project enhanced some side skills such as our adaptation to a pre-existing global work as well as the information retrieval in order to implement our own functions. During the first step of this project, we had to implement the ImageJ *Distance Map* morphological operator. Whether our results look at least quite similar, ImageJ seems to remain performing better in the processing or more higher sized images. Using webGL would be interesting in order to see the difference.

6 Sources

References

[1] R. Gonzalez and R. Woods. Digital Image Processing, Addison-Wesley Publishing Company, 1992.

[2] A. Jain. Fundamentals of Digital Image Processing, Prentice-Hall, 1989.

[3] E. Danielsson. Euclidean Distance Mapping, Computer Graphics and Image Processing, 14, 227-248, 1980.