Image Processing II
Watershed

Aadil Anil Kumar
Otmane Sabir

29/2/2020

Introduction

The second homework assignment required us to implement the watershed transform while following certain guidelines which could be summarized to the following list:

1. Implement the watershed algorithm as described as pseudo code from the textbook 4-connected and 8-connected neighborhood.
2. Output a single CSV file for the transformed image in the same format and same definition and value domains as the input image 'f'.
3. Do meaningful (motivated from a real-world perspective) watersheds for 3 other images.

Contents

1 Watershed Transform

The watershed transform is a classical segmentation - separating different objects in an image - algorithm in the world of image processing; it initially derives from the the geological watershed which means the elevated terrain that separates neighboring drainage basins. The watershed transform segmentation follows the same logic: starting from defined markers, the watershed algorithm treats pixels values as a local topography (elevation). The algorithm floods basins from the markers until basins attributed to different markers meet on watershed lines. *See Figure 1*

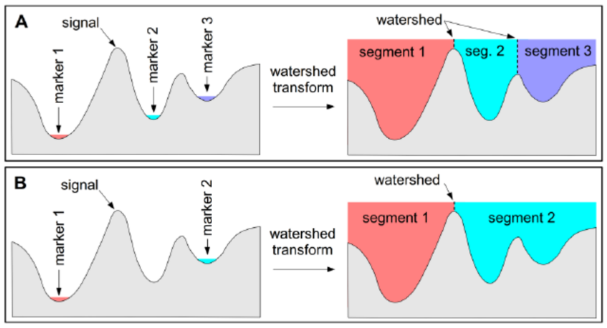


Figure 1: Watershed Visualization

1.1 Formal Definition:

DEFINITION:[2]

Let $f \in C(D)$ have *minima* $\{m_k\}_{k \in I}$, for some index set I. The catchment basin $CB(m_i)$ of a minimum m_i is defined as the set of points $x \in D$ which are topographically closer to m_i than to any other regional minimum m_j:

$$CB(m_i) = \{x \in D | \forall jn \in I\{i\} : f(m_i) + T_f(x, m_i) < f(m_j) + T_f(x, m_j)\}$$

The watershed of f is the set of points which do not belong to any catchment

$$Wshed(f) = D \cap \left(\bigcup_{i \in I} CB(m_i) \right)^c .$$

Let W be some label, $W \notin I$. The watershed transform of f is a mapping $\lambda : D \to I \cup \{W\}$, such that $\lambda(p) = i$ if $p \in CB(m_i)$, and $\lambda(p) = W$ if $p \in Wshed(f)$.

So the watershed transform of f assigns labels to the points of D, such that (i) different catchment basins are uniquely labelled, and (ii) a special label W is assigned to all points of the watershed of f.

2 Algorithmic Definition

In our implementation, we're relying on the simulated immersion approach introduced by Vincent & Soille[3].

Metaphorically speaking, the algorithm pierces holes in every minimum, and the entire relief begins to be flooded with water. Starting from the minimum of lowest height, the water gradually fills up all catchment basins. Watersheds are built in the places where water from different basins unites. The process ends when the water reaches the maximum peak of the relief, and as a result, every catchment basin gets covered by the watershed lines which are easily distinguishable lines from the rest of the output.

2.1 Formal Definition

Definition 2:[2]

Let $f : D \rightarrow$ be a digital grey value image, with h_{min} and h_{max} the minimum and maximum value of f. Define a recursion with the grey level h increasing from h_{min} to h_{max}, in which the basins associated with the minima of f are successively expanded. Let X_h denote the union of the set of basins computed at level h. A connected component to the threshold set T_{h+1} at level $h+1$ can be either a new minimum, or an extension of a basin in X_h: in the latter case one computes the geodesic influence zone of X_h within T_{h+1}, resulting in an update X_{h+1}. Let MIN_h denote the union of all regional minima at altitude h.

Let us now define the following recursion:

$$\begin{cases} X_{h_{min}} = & \{p \in D | f(p) = T_{h_{min}}\} \\ X_{h+1} = & MIN_{h+1} \cup IZ_{T_{h+1}}(X_h), h \in [h_{min}, h_{max}) \end{cases}$$

The watershed $Wshed(f)$ of f is the complement of $X_{h_{max}}$ in D :

$$Wshed(f) = DnX_{hmax}$$

Let's use Figure 2 from [2] as an example of the algorithm defined above. Assuming that A and B are basin labels and that W is the watershed pixels. Then the algorithm will proceed as shown in the figure (a) is the original image and the (b-e) are steps following the algorithm definition above.

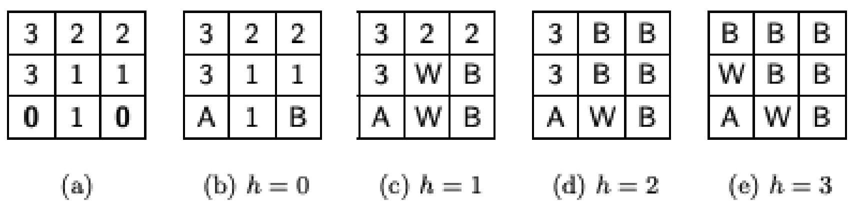


Figure 2: Immersion algorithm on the 4-connected grid

2.2 Implementation

As we previously discussed, we attempted to implement the immersion approach. The pseudo code below was provided by Soille [3].

2.2.1 Pixel Mapping

One of the main challenges we encountered during the implementation of the algorithm was mapping the pixels using their indices to their specific locations and gray values. First, we tried getting all the pixels using the image itself but then found a better solution which doesn't require any information from the image besides its height and width - we'll refer to these values as *H* & *W* respectively. The idea is to generate a matrix of the same H and W and assign it x, y values the same way they would be assigned to a grid. The idea is creating two matrices, one where we create a row ranging from 0– and duplicate it over H number if times. We then create a second matrix where rows are a constant value *C* repeated width times and the columns increment *C* by one after each iteration ranging from 0–. We then take these matrices, reshaped them in the format of a list of 2 element arrays and transpose it to get our [y, x] mapping.

2.2.2 Neighbouring Pixels

The neighbouring pixels follows the same logic as the previous section but with a different way of interpreting the range. We decided to use a combination of the pixel mapping logic but also by introducing a different way to limit how far we go. We first get the maximum and minimum values of the given pixel. This allows us to first find whether the current pixel is at a border and also allows to avoid having an issue with a negative value. As all of our output is initialized with a -1 value. We then generate two matrices in a similar fashion to the previous section 2.2.1 and perform similar matrix handling operations in order to obtain a mapping of these pixels. We also allowed the user to freely pick a number of pixels which we divide by 2 in order to make sure we're not giving more than what the user is asking for. This also made it easier for us to experiment with differently sized number of neighbors in our future sections.

Algorithm 1 Watershed using flooding simulations

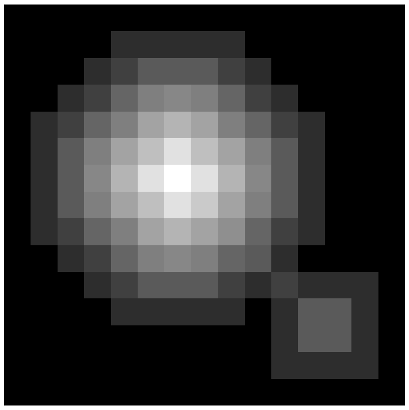
```
procedure WATERSHED
    mask ← -2 ; initial value of a threshold level
    wshed ← 0 ; value of pixels belonging to watersheds
    inqueue ← -3 ; value assigned to pixels put into the queue
    init ← -1 ; initial value of f0)

    Input : $f_i$, grey tone image (non negative integers):
    Output : $f_0$, image of labelled catchment basins:

    procedure INITIALISATIONS
        Value $f_i$ is assigned to each pixel of $f_0$:
        current_label ← 0:
        flag: Boolean variable:
        $N(p)$ is the set of neighbours of P:

    Sort the pixels of $f_i$ in the increasing order of their grey values.

    for $h \leftarrow h_{min}$ to $h_{max}$ do    ▷ geodesic SKIZ of level h - 1 inside level h
        for pixel p do                                   ▷ such that $f_i(p) = h$
            $f_0(p) \leftarrow$ mask
            if ∃ p' ∈ N(p) — $f_0(p') > 0$ OR $f_0(p')$ = wshed then
                $f_0(p) \leftarrow$ inqueue                       ▷ fifo.add(p);
    while fifo.empty() = false do
        p ← fifo.retrieve();
        for pixel p' ∈ $N(p)$ do
            if $f_0(p') > 0$ then  ▷ i.e., p' belongs to an already labelled basin
                if $f_0(p)$ = inqueue OR ($f_0(p)$ = wshed & flag = true) then
                    $f_0(p) \leftarrow f_0(p')$
                else if $f_0(p) > 0$ AND $f_0(p) \neq f_0(p')$ then
                    $f_0(p) \leftarrow$ wshed                       ▷ flag ← false
            $f_0(p')$ = wshed
            if $f_0(p)$ =inqueue then
                $f_0(p)$ ← wshed;
                flag ← true;
            $f_0(p')$ = ,ask
            $f_0(p)$ ← inqueue;
            $f_0(p')$ ← mask;

        for pixel p do                              ▷ such that $f_i(p)$ = mask
            if $f_0(p)$ = mask then                      ▷ check for new minima
                current_label ← current_label + 1;
                fifo.add(p);
                $f_0(p)$ ← current_label;
                while fifo.empty() = false do
                    $p'$ ← fifo.retrieve();
                    for p" ∈ $N(p')$ do
                        if $f_0(p'')$ = mask then
                            fifo.add(p");
                            $f_0(p'')$ ← current_label;
```

3 Experiments & Results

Please find all the experimental outputs in the outputs folder of the directory.

3.1 Test Input 1

f1_dinv is a signed image blob. We compute the distance transformation of f1 to derive the appropriate graylevel intensities of pixels inside the foreground region to show the distance to the closest boundary from each pixel.


(a) Distance transformation of f1

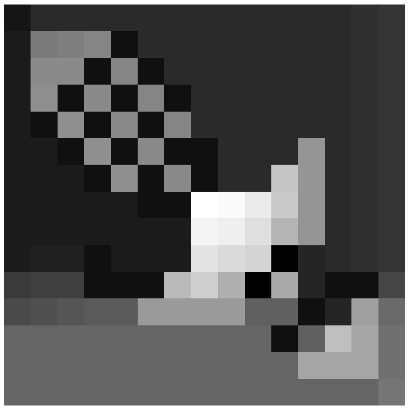

(b) Watershed with 4 Neighbours of f1

Figure 3: Results of experiments on f1

Figure 3b show us the result of applying the watershed algorithm to Figure 3a, we arrive at an image that depicts a 8-bit sword.

3.2 Test Input 2

f2 is also a signed blob. By saving it as an image, figure 3 shows us that f2 is an image containing different circles.

We proceed to apply the watershed algorithm to the image.

3.2.1 Interpretation

Figure 5a shows us that using 4 neighbours for water-shedding results in a over-segmented image. On the other hand, figure 5b shows us that using 8 neighbours for water-shedding results in an image having fewer regions. In general, the more neighbours that you have the lower the amount of regions that are present in the resulting image. This occurs due to the algorithm simultaneously creating larger regions beginning at the initial regional minima. For each regional minimum, each iteration of the 8-connected algorithm uses a

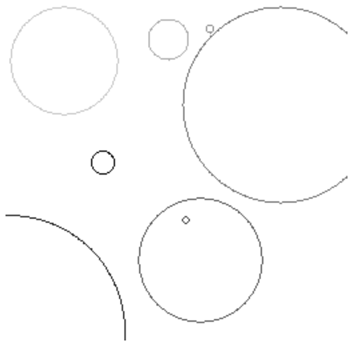
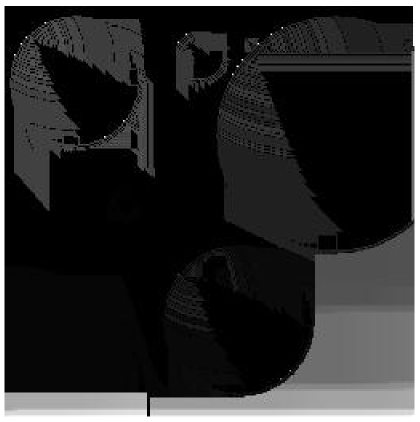
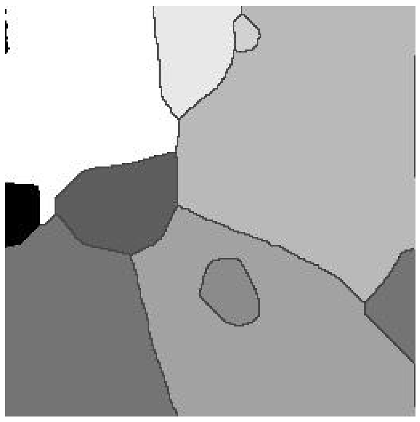


Figure 4: f2 as jpg



(a) Watershed with 4 Neighbours of f2



(b) Watershed with 8 Neighbours of f2

Figure 5: Results of experiments on f2

bigger region to classify the minimum's neighbours, resulting in larger but fewer regions.

3.3 Nuclei Segmentation

One of the many uses of segmentation is with nuclei. Cell nuclei are important indicators of cellular process and diseases and power of image processing has been able to automate this process in order to gain further insight into cells, their features, and functionalities. Therefore , we decided to also test the efficiency of our immersion watershed in splitting heavily clustered nuclei.

3.3.1 Results

The results we obtained were after heavy processing and show how the combination of different filters and algorithms can help us obtain the needed results.

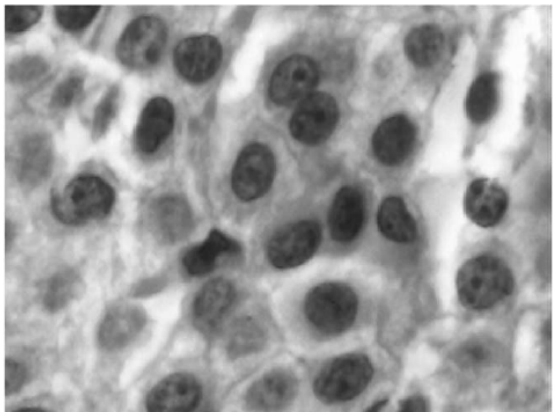
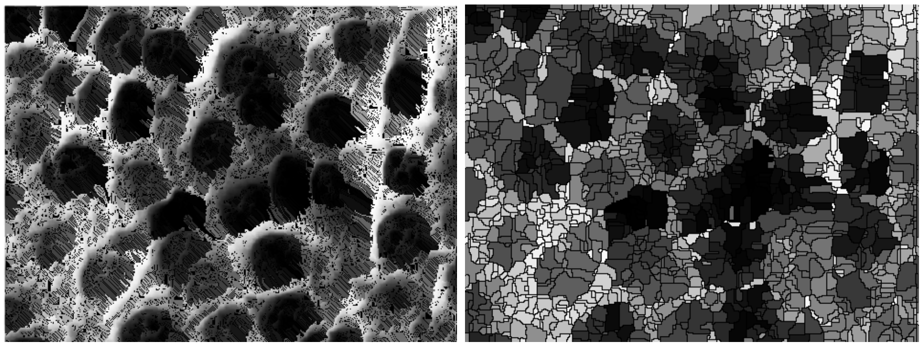


Figure 6: Original Cell Nuclei Image.

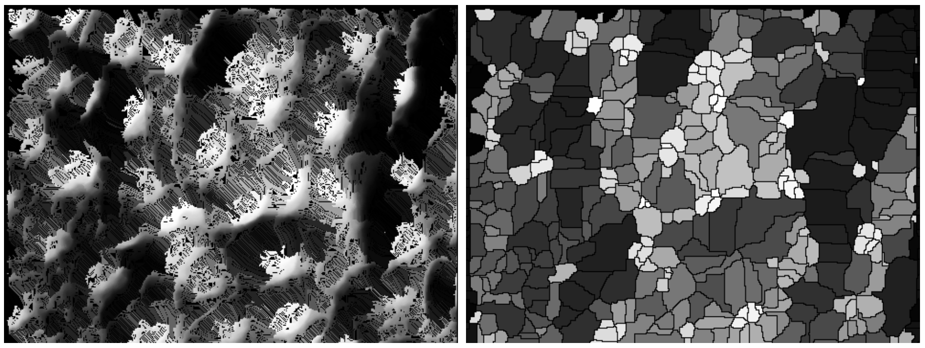
We first started by directly applying the watershed transform to this image without any sort of prepossessing or filters, see Figure 7. One of the first things we noticed is how the segmentation works inversely. As it's trying to remove the nuclei but maintain the background; therefore, the first step was to apply an inverse (bit-wise not) filter in order to invert the image - this allows us to shift focus on the nuclei. We also noticed how the image noise and nuclei cell clusters are causing the image to over segment regardless which made us think of different filters which will reduce this noise and also maintain the edges to a certain extent and the best fit was the blur filter.


(a) 4 Neighbors

(b) 8 Neighbors

Figure 7: Applying watershed only

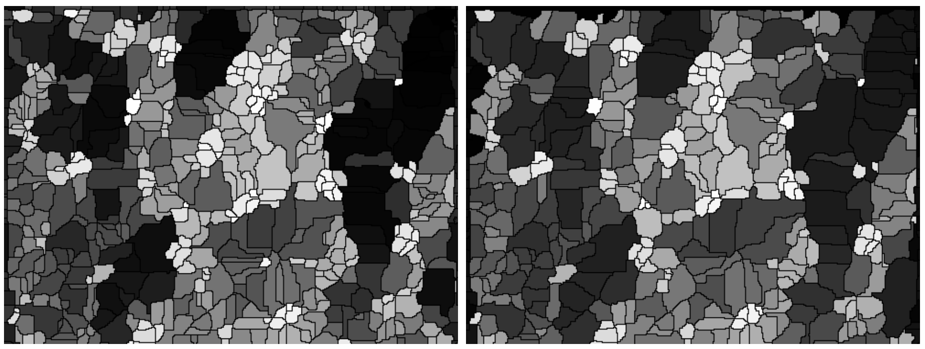
After applying both the blur & the inverse, see Figure 8. These results showed less segmentation and undesired effects but allowed us to come up with two conclusions. First, that the 4-neighbor watershed is not very helpful and its results are not valid; therefore, we decided to mainly rely on the 8-neighbors watershed as it was showing more prominent results. Second, the average blur filter was very helpful in getting rid of over-segmentation but we now wanted to compare its results to a different filter the bilateral filter.



(a) 4 Neighbors (b) 8 Neighbors

Figure 8: Watershed with a median blur filter.

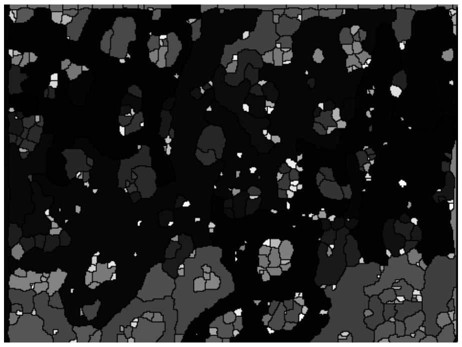
After applying both filters, see Figure 10, we noticed that while the bilateral filter does define basins better it also over segments the image a little more than the median filter and since these results were not conclusive enough for us to pick one of these filters we decided to carry our tests with both of them until one proves to give better results the other. Since the segmentation was still not producing the results we were expecting, we decided to add more processing and adaptive threshold was the next step.



(a) Bilateral Blur Filter. (b) Median Blur Filter.

Figure 9: 8-neighbor watershed using different blur filters.

The adaptive threshold was maybe one of the most important steps towards getting better results. The algorithm determines the threshold for a pixel based on a small region around it. So we get different thresholds for different regions of the same image which gives better results for images with varying illumination. The results were a big improvement and this also where we started noticing big differences between the median and bilateral filters. As we can see in figure **??**


(a) Bilateral Blur.

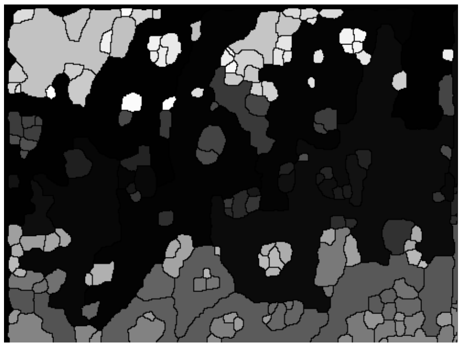
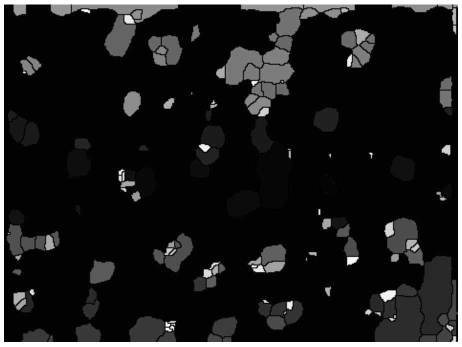

(b) Median Blur.

Figure 10: 8-neighbor watershed with adaptive threshold.

We now decided to give it one more change and increase the image contrast. That would allow us to completely dim out the dark areas and make the other ones more visible. Surprisingly, this gave better results in general, see Figure 11 and we were now finally able to completely view these segmented cells. The nuclei is now marked in a different color and a lot of the information that we don't need is filtered out. This left us with a lot better results generally and also concluded that a median blur is what we need in this case.


(a) Bilateral Blur.

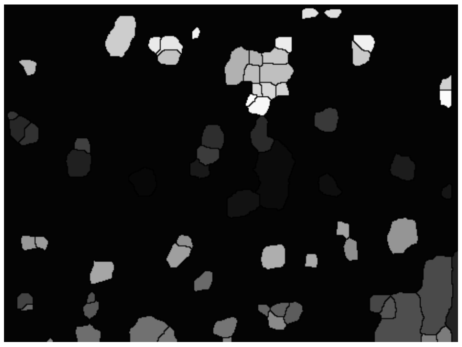

(b) Median Blur.

Figure 11: 8-neighbor Watershed with all filters.

3.3.2 Conclusion

This serves as a good example, see Figure 12 to show how much pre-processing an image can give much better results.

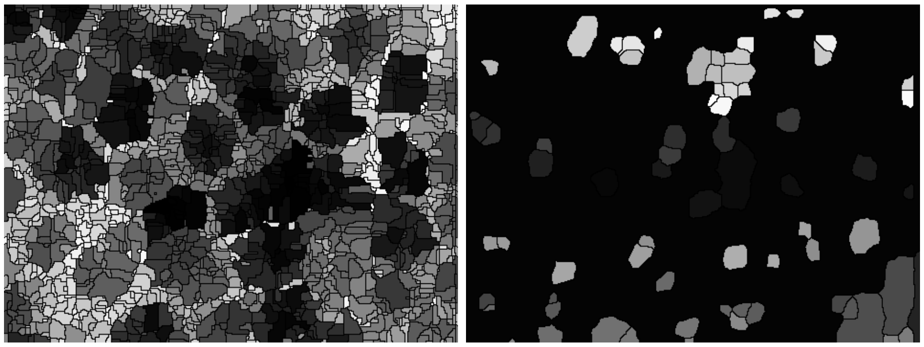

(a) Initial 8-Neighbour Segmentation. (b) Final result.

Figure 12: 8-neighbor Nuclei Watershed.

3.4 Brain MRI Slice Segmentation

Another valid use of segmentation functions in the realm of medical image processing is segmenting MRI slices of the brain. This is done in order to isolate different structures and tissues within the brain to check for the existence of any abnormalities pertaining to an individuals medical condition. These abnormalities could be identified by tracking changes in volume, shape and regional distribution of brain tissue during by comparing it to previous scans.

3.4.1 Results and Interpretations

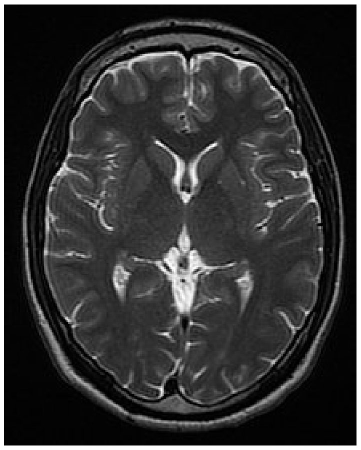
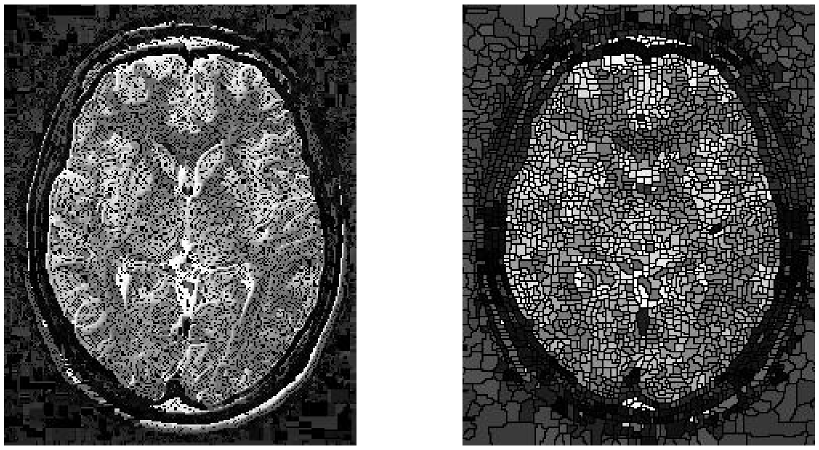


Figure 13: Original image of the brain.

The MRI Slice is a gray scale image. We begin by first directly applying the watershed transform to the image without the use of any morphological pre-processing, figure 14 depicts these results.



(a) Watershed (4-Neighbours) of image. (b) Watershed (8-Neighbours) of image.

Figure 14: Direct application of watershed to the image.

The original image, figure 13, shows the presence of some noise. Therefore, we proceed with applying a filter before segmenting the image. In terms of filter choice, since the brain contains a lot of peaks and troughs within it, we thought that it would be best to use a edge-preserving filter. The two options that we conducted experiments with were the median and bilateral filters, figure 15 depicts the results.

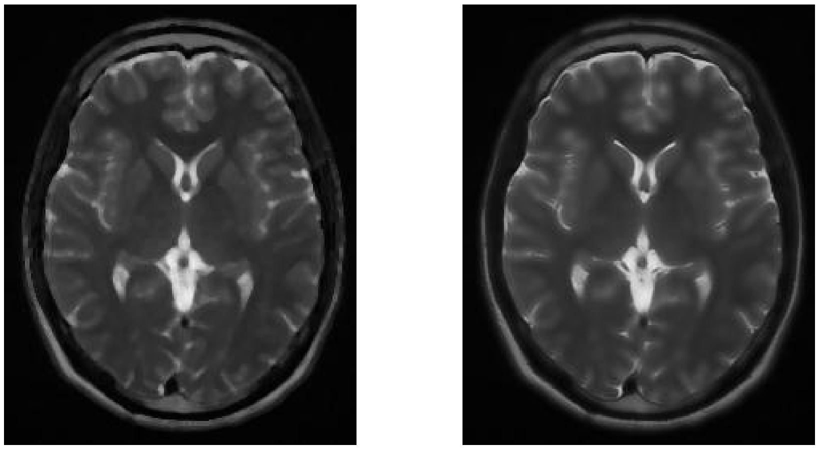
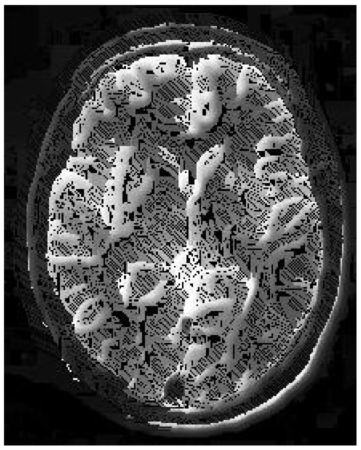

(a) Median filter (b) Bilateral filter

Figure 15: Application of filters to the original image.

Figure 15 shows us that both filters result in an edge-preserved image. However, the bilateral filter results in a less noisy image, therefore, we shall now segment this image.

Figure **??** shows us the results of segmenting the bilaterally filtered image. When comparing these results to figure 14 we can see that we obtain a much better segmentation of the brain. This would allow for better time interval based comparisons of a person's brain scans.

Additionally, figure 17 shows us the result of using 12 neighbours for the segmentation. Our implementation of the algorithm allows us to pick the amount of neighbours we want for each run. Although we are not exactly sure of how it works for neighbours > 8, we assume that algorithm gets arbitrary neighbours of the neighbours of the current pixel being referenced.


(a) 4-Neighbours

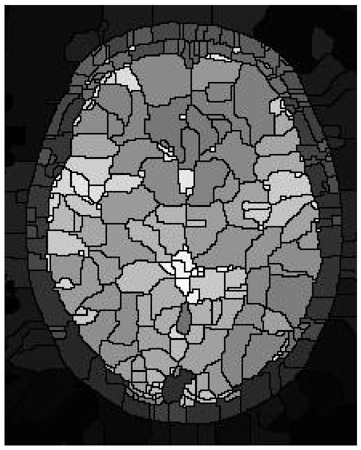

(b) 8-Neighbours

Figure 16: Applying watershed to the filtered image.

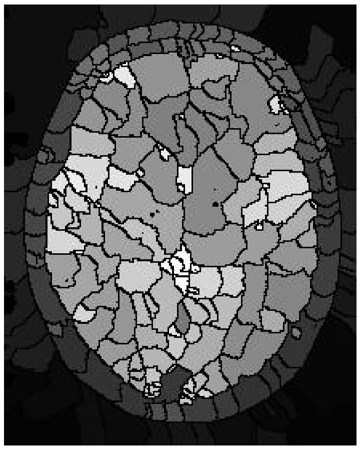


Figure 17: 12-Neighbours

3.5 Animal Segmentation for Classification

Image segmentation plays a key role in machine learning, particularly in image classification applications. In this experiment we try to segment a zebra (Larry) from its background of the Savannah. A segmented Larry could help deep learning algorithms classify him as a zebra.

3.5.1 Results and Interpretations

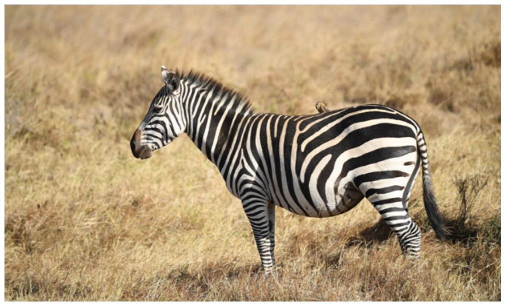


Figure 18: Original image of a zebra

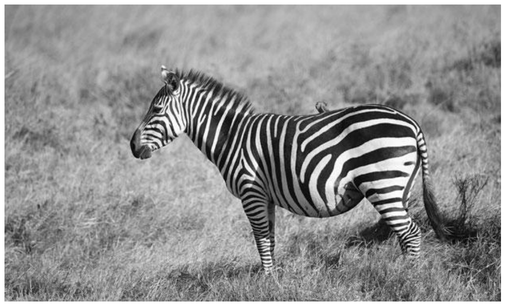
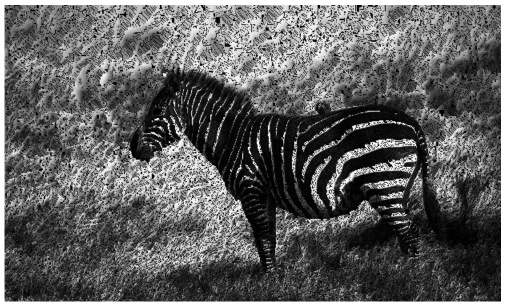
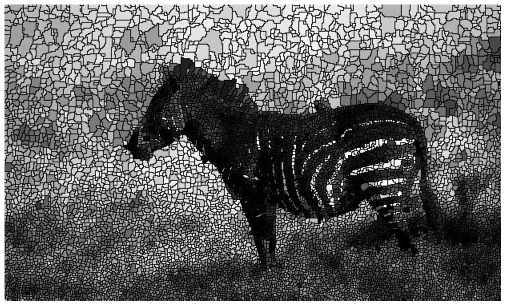


Figure 19: Gray scale image

We first convert the image to gray scale. Then we try to directly apply the watershed transform.



(a) 4-Neighbours



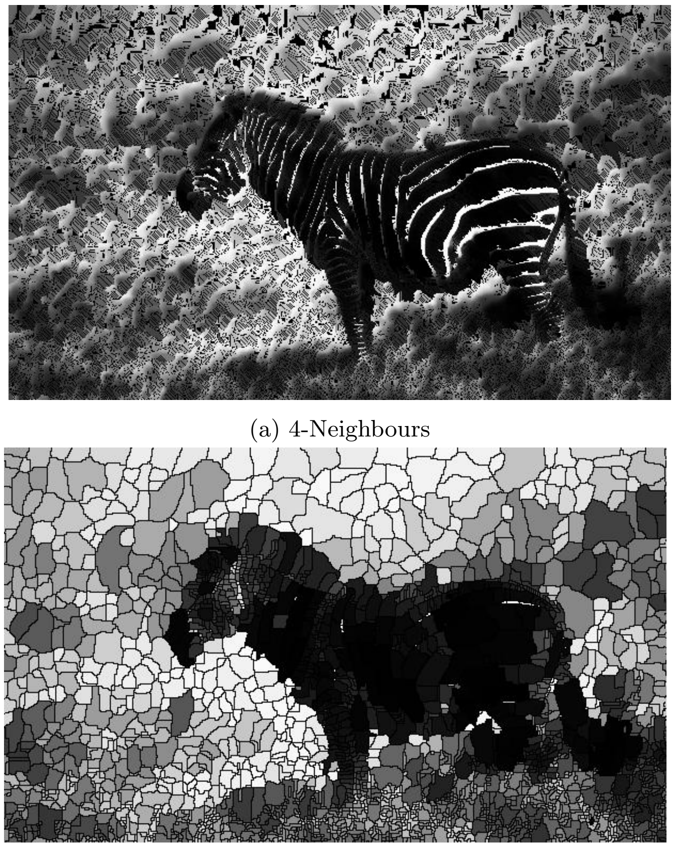
(b) 8-Neighbours

Figure 20: Directly applying watershed to the image.

Once again, Figure **??** shows us that the direct application of the watershed transform results in a heavily over-segmented image. In order to combat the over-segmentation we once again make use of the bilateral filter to preserve the edges while reducing the noise.

Figure **??** shows us that performing the segmentation after applying the filter results in fewer regions being formed in comparison to Figure **??**. The foreground object (zebra) is much clearer in comparison to the background. This would ideally allow for segmenting the animal from its environment for classification.

However, the drawback of the immersion based watershed algorithm is evident by the persisting presence of many regions.


(a) 4-Neighbours

(b) 8-Neighbours

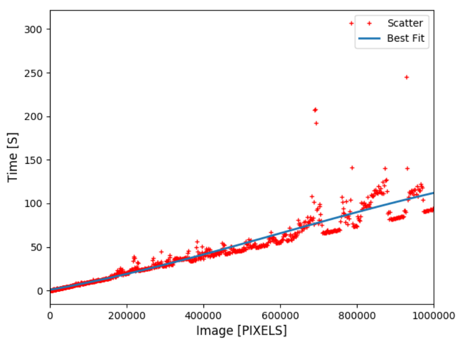
Figure 21: Directly applying watershed to the image.

4 Comparison

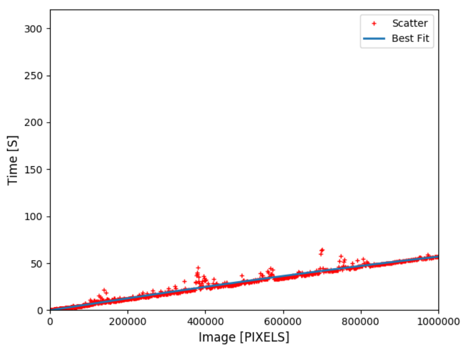
The breadth-first implementation that we're discussing uses a FIFO (First in First Out) queue to find the level of pixels of constant grey value as we have seen, c.f Algorithm 0. For each of these thresholds a pixel is stored in the empty queue, followed by the 'flooding' which runs until the queue is empty.

4.1 Time Complexity

We ran some tests on the algorithm which was presented in order to visualize the time complexity of this algorithm. We can clearly see that the time complexity is linear in the number of pixels in the image; however, the theoretical time complexity would change from linear to quadratic due to the repeated processing of the watershed pixel [1].



(a) 8 - Neighbours



(b) 4 Neighbours

Figure 22: Scatter Plot & Best Fit curve of results

4.2 N-Neighbors Influence

After the previous tests, we noticed how the number of neighbors scanned impacts the time the algorithm; therefore, we decided to run different test with various neighbor sizes (2, 4, 8, 16) and visualized the differences as seen in Figure 24. We ran a test on random graphs of size [n , n] with n ranging from 1–500. We also ran a test, c.f 23 using 2, 4, 6, 8, 16, 32, and 64 to be able to determine how exactly the amount of neighbors influences the overall run time of the algorithm on a 512*512 image.
This helps us explain how the algorithm heavily depends on its neighbors and why, as previously mentioned, the time complexity can vary from linear to quadratic. We almost always see a doubling the overall time until we reach 64 neighbors and notice a much more significant jump.

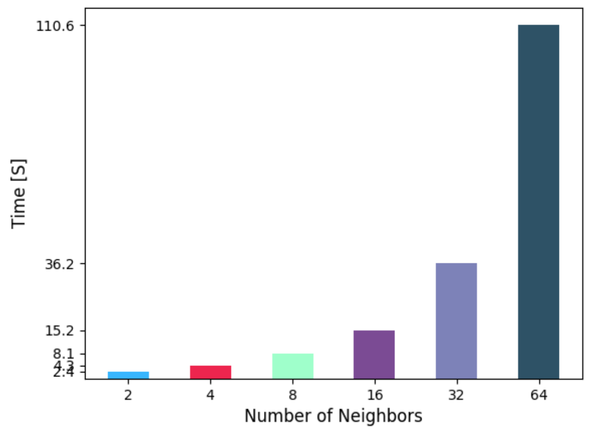
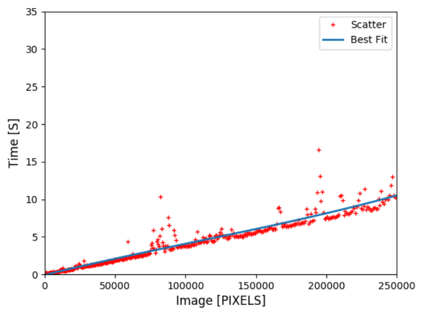
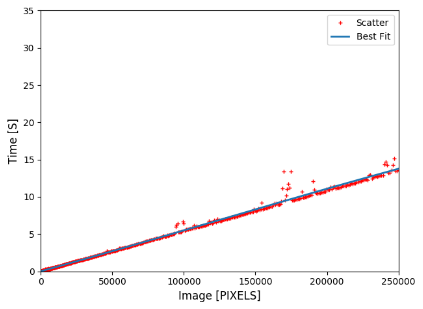


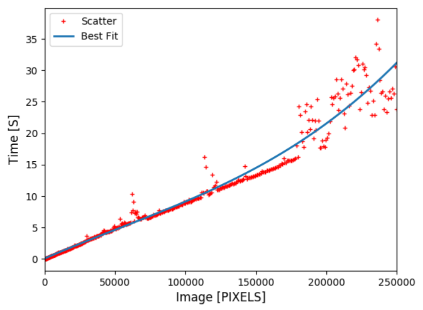
Figure 23: Different neighbors bar chart.



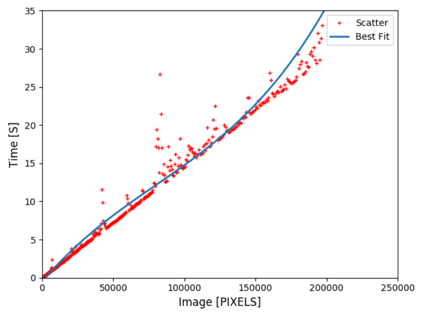
(a) 2 - Neighbors



(b) 4 - Neighbors



(c) 8 - Neighbors

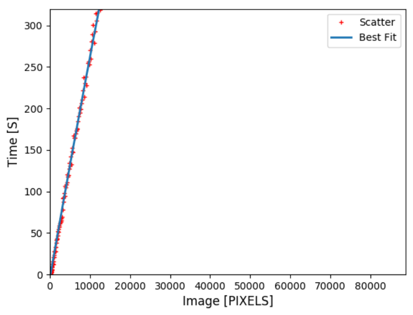


(d) 16 - Neighbors

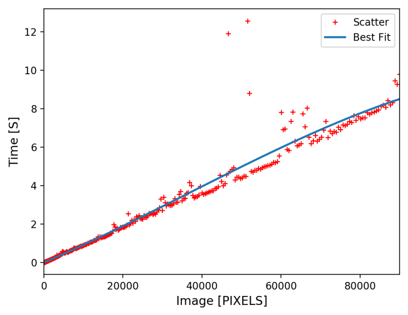
Figure 24: Scatter Plot & Best Fit curve for different neighbors

4.3 Disadvantages of the Queue

One of the main advantages of this algorithm is how it utilizes the queue to reduce time complexity and keeps all our operations feasible in linear time but this also comes at a cost. When running our initial test, we tried running multiple processes at the same time in for optimal performance; however, we ran into many issues because of the nature of the algorithm. The required size of the queue is not known in advance, and memory is addressed in a very unstructured and "blind" manner, causing performance degradation on virtual memory and especially on parallel computing, since it requires a lot of synchronization and tends to leave with many imprecise results. In the figures below you can clearly see the difference when running the processes concurrently vs. running them in normal linear manner.



(a) Concurrent Testing



(b) Consecutive Testing

Figure 25: Scatter Plot & Best Fit Curves for Consecutive and Concurrent testing.

5 Task Distribution

As we have previously worked together, we were more familiar with task distribution. We also decided to both take up on more tasks to share as much of our results and progress as possible. The list below summarizes all of our progress.

- Aadil Anil Kumar :
  - Wrote the GetPixels/Watershed/DistanceTransform function.
  - Ran experiments and their respective explanations.
  - Wrote parts of the report.
  - Report proof reading.

- Otmane Sabir :
  - Wrote the getNeighbors/Test script and partially the Watershed function.
  - Ran one experiment (the nuclei expirement)
  - Ran the time complexity and the algorithm performance tests.
  - Wrote parts of the report.

References

[1] AS Kornillov :
An Overview of Watershed Algorithm Implementations,
`https://www.mdpi.com/2313-433X/4/10/123/pdf`

[2] Jos B.T.M. Roerdink and Arnold Meijster :
Institute for Mathematics and Computing Science,
`http://www.cs.rug.nl/roe/publications/parwshed.pdf`

[3] Soille, P. (2010). Morphological image analysis: principles and applications.
`Berlin: Springer. Samarin`.