CS 675 – Computer Vision – Spring 2022
Instructor: Marc Pomplun

Assignment #2

Sample Solutions

Question 1: Binary Image Processing

(a) Let us now use the size_filter() function to remove the noise in text_image.png. Note that there is positive noise as well as negative noise; we would like to remove both. Do not modify the size_filter() function at all, but write a new function posneg_size_filter(img, threshold) that calls size_filter() to do this task and return the resulting image. It uses a single threshold for both types of noise. Determine the threshold that gives the best result (just by visual inspection).

The easiest way to do this is to filter the image, then invert it so that black pixels turn white and vice versa, and then apply the filter again. This time the filter will remove the (formerly) negative noise. You just need to invert the image one more time to get the correct result, i.e., with black object pixels and white background pixels. A size threshold of 20 works well. See the attached file "assignment2_solutions.py" for a code example.

(b) The flood fill algorithm is actually reasonably efficient, but it is clear that in principle the two-pass algorithm we discussed in class can be implemented to run more efficiently. So, let us write some code that applies this algorithm. There are already some lines of code in the function label_components() for you to start with. You can use them or not, it is up to you. If you use them, note that the commented-out block of code should be used to build the equivalence table, and the code at the end will use that table to relabel all components accordingly so that you get a result of the same format as in label_components_by_flooding().

Unfortunately, there was a small bug in the code that I provided. If two pixels were found to be equivalent, i.e., belong to the same component, but they had also previously been found to be equivalent to other, disjoint sets of pixels, then it could happen that a single component was incorrectly split into two (or possibly even more) components.

Sorry for that; I uploaded a corrected version of both assignment2.py and assignment2_solutions.py. You received full points regardless of whether you fixed or even noticed the bug.

(c) Implement the expanding and shrinking operations as discussed in class. Expand(img) and shrink(img) should return the image that results from the corresponding operation.

Then write a function expand_and_shrink(img) that uses a sequence of expand and shrink operations to remove the positive and negative noise in text_image.png as best as possible.

For text_image.png, this method does not work very well, because there are too many randomly flipped pixels. One possible solution is an expand-shrink-shrink-expand sequence as shown in assignment2_solutions.py. It removes almost all positive and negative noise, but the shape of the characters is significantly distorted.

Question 2: Properties of Connected Components

As you will certainly agree, the following binary image contains two connected components. Let us call them A and B:

r/c	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0										
1		■	■	■			■	■	■	
2		■	■ A	■	■			■	■	
3		■	■	■	■			■	■	
4			■	■	■			■	■	
5								■	■	
6								■	■	
7			■	■	■	■	■	■	■ B	
8		■	■	■	■	■	■	■	■	
9										

(a) Compute the center of gravity in the given row/column coordinate system for each of the connected components.

(b) Compute the compactness of each of the two connected components. Let us define the perimeter of a component as its number of boundary pixels (see definition).

Show every step of your computations.

(a) For component A, its area is 14, and the sum of its vertical coordinates is:

1 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4 + 4 = 35

Therefore, the vertical coordinate of A's center of gravity is 35/14 = 2.5

As it turns out, a is mirror symmetric at the matrix diagonal and thus the sets of its vertical and horizontal coordinates are identical, which means the horizontal coordinate of its center of gravity is 2.5 as well.

In summary, A's center of gravity is at (2.5, 2.5).

For component B, its area is 28, and the sum of its vertical coordinates (let's go row by row this time) is:

3 + 4 + 6 + 8 + 10 + 12 + 49 + 64 = 156

Therefore, its center of gravity has a vertical coordinate of 156/28 ≈ 5.57

The sum of its horizontal coordinates is:

1 + 4 + 6 + 8 + 10 + 18 + 56 + 64 = 167

Therefore, its center of gravity has a horizontal coordinate of 167/28 ≈ 5.96

So B's center of gravity is at (5.57, 5.96), which is actually located outside of B.

(b) By definition, boundary pixels are those that have at least one zero-pixel as their 4-neighborhood.

Component A has 10 boundary pixels, and its area is 14, which means that its compactness is $10^2/14 \approx 7.14$.

Component B has 27 boundary pixels (all of its pixels except the one at (7, 7)), and its area is 28, which means that its compactness is $27^2/28 \approx 26.04$.

Since a higher compactness value actually means that a component is less compact, these results are plausible for the given components. Note that using the number of boundary pixels as the perimeter strongly underestimates the actual perimeter that we could obtain by measuring the sum of the length of all cracks, i.e., edges between 1-pixels and 0-pixels. As a consequence, we can find compactness values below that of a circle, which we claimed to be impossible.

The advantage of using the number of boundary pixels for this computation is that it is simpler to compute and is typically strongly correlated with the crack-based measure.