CSC3104 Computer Vision

Lab 1: Advanced Morphological Operations

A. Boundary Extraction

There are several ways to extract the boundary of a region by using morphological operations. For example, you can first apply erosion to the region, and then use it to subtract from the original version of the region, as shown in Eq. (1). In this case, A is used to represent the region, and SE is used to represent the structuring element. You can try SE with different shapes, but please ensure that you are using a symmetric SE. The other way is to first apply dilation to the region, and then subtract the original version from the dilated version of the region, as shown in Eq. (2).

$$Boundary = A - (A \ominus SE) \quad \text{Eq. (1)}$$

$$Boundary = (A \oplus SE) - A \quad \text{Eq. (2)}$$

Exercise

Create the binary image as shown in Figure 1 by using '0' for the unshaded region and '1' for the shaded region. Try the two different boundary extraction approaches mentioned in the previous paragraph and observe the difference(s) between the two.

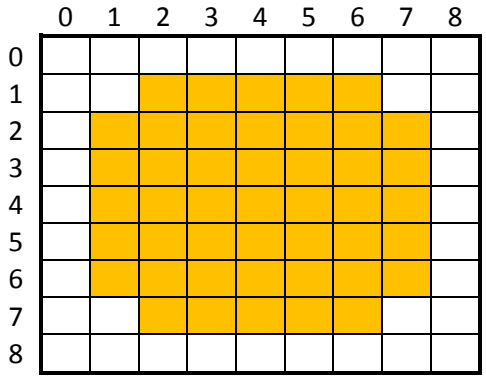


Figure 1

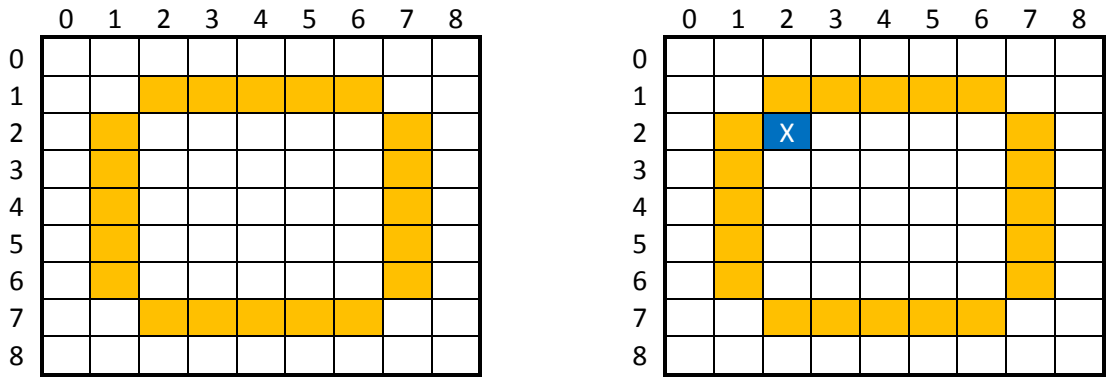
B. Region Filling

If you would like to fill up the empty part within a region, you can use try to complete the task by using dilation. However, it might not work in certain situations, such as when the size of the empty part is larger than the boundary. For example, you can try to create the binary image as shown in Figure 2(a), and then try to fill up the empty part within the shaded boundary using SE with various sizes.

In this case, we have to resolve the issue by using a series of morphological operations (or repeating morphological operations) as listed below.

1) Get the complement version of the binary image.
2) Get the original version of the binary image, select a pixel within the boundary as the starting point.

3) Apply dilation to the selected point with a symmetric SE.
4) Logical AND the output from (3) with the complement version of the binary image.
5) If there is no difference between the output from (3) and (4), stop the process.
6) Otherwise, use the output from (4) as the starting point and repeat step (3) and (4).



(a) Binary Image (b) Starting Point [2,2]

Figure 2

This series of operations can be mathematically represented by using Eq. (3). In this case, A represents the original binary image, A^C represents the complement version of the binary image, and SE represents a symmetric SE. We will repeat the operations starting with k = 1, until X_k is the same as X_{k-1}, where X_0 is the starting point within the boundary.

$$X_k = (X_{k-1} \oplus SE) \cap A^C \text{ for } k = 1, 2, 3, \ldots \quad \text{Eq. (3)}$$

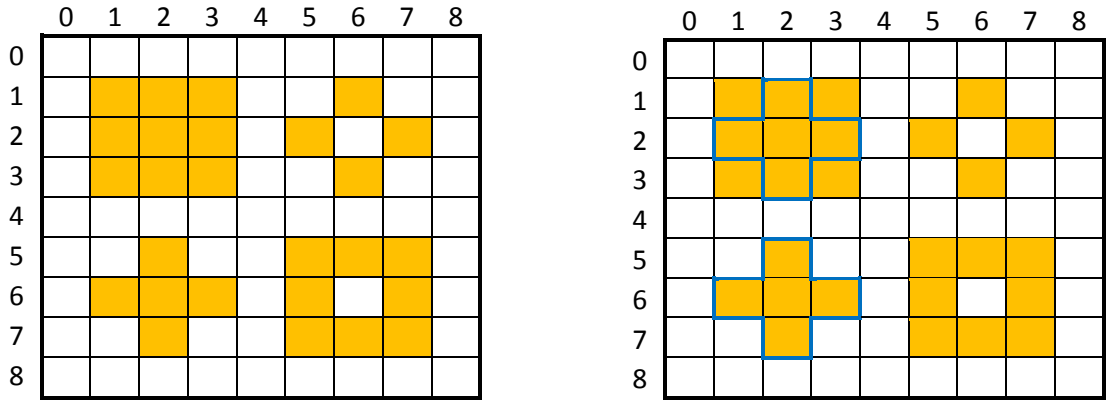
However, this is not the end of the entire operation. The operations above only get you the inner region within the boundary. You still need to union (or add) the inner region that you obtained with the boundary to get the final answer.

Exercise

By using point (2,2) as your starting point as shown in Figure 2(b), write a piece of code to fill up the empty part within the boundary by using the series of morphological operations mentioned above. In this case, please use a cross-shaped (or plus-shaped) SE with the size of 3x3 instead of a square one. You can also try with a square-shaped SE to see what would happen.

C. Hit-or-Miss Transformation

By using the binary image shown in Figure 3(a) as an example, if the purpose is to find the number and location of cross-shaped (or plus-shaped) region, using opening (erosion then dilation) will return 2 regions instead of 1. The another one is coming from the square-shaped region located near the top left corner, because we can also find a cross-shaped region inside this square-shaped region as shown in Figure 3(b).



(a) Binary Image (b) Cross-shaped Regions

Figure 3

If we do not want to consider the cross-shaped region that exists within the square-shaped region, we can use the hit-or-miss transformation (sometimes also known as hit-and-miss transformation). It can be used to find regions that fulfill certain configuration and it is much more robust when compared to using just opening.

Different from the other morphological operations that we have tried previously, we need to have two different SE to complete the hit-or-miss transformation. The first SE defines the foreground pixels (those pixels with value of '1' in the binary image) that it will try to match, whereas the second SE defines the background pixels (those pixels with value of '0' in the binary image) that it will try to match.

In this case, to match the cross-shaped region that we are looking for, we need to first create a cross-shaped SE with the size of 3x3. Another SE that we need can be just the complement version of the cross-shaped SE you created (not compulsory). Assume that A is used to represent the original binary image, cSE is used to represent the cross-shaped SE, and cSE^C is used to represent the complement version, apply the following operations to complete the hit-or-miss transformation.

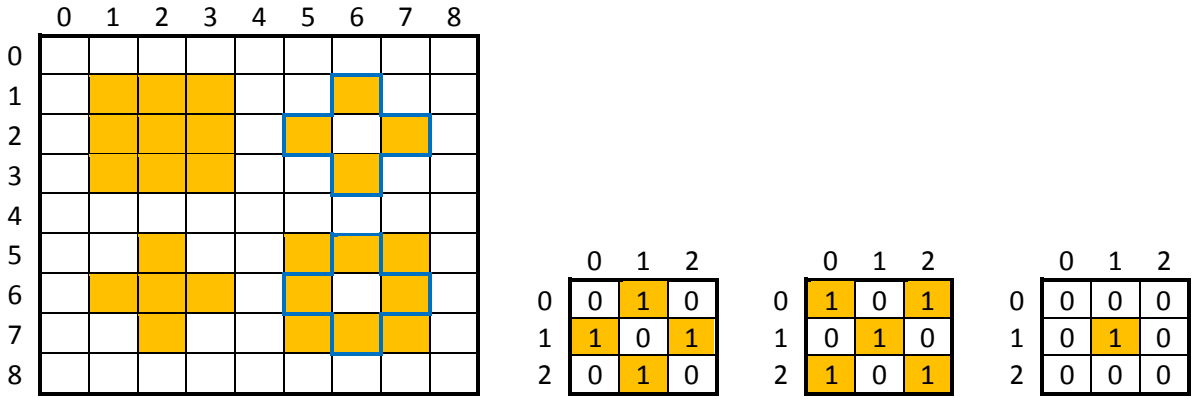
1) Get the complement version of binary image A and save it as A^C.
2) Apply erosion to A with cSE.
3) Apply erosion to A^C with cSE^C.
4) Logical AND the output from (2) and (3) to find the location of each cross-shaped region.

We can then count the number of '1' to determine the number of regions that fulfills the requirement or look at the location of '1' to know the center point of the region that we are searching for. Overall, the hit-or-miss transformation of A by a SE can be mathematically written as shown in Eq. (4), where the $\circledast$ symbol is used to represent the hit-or-miss transformation.

$$\text{Hit-or-Miss}(A, SE) = A \circledast SE \qquad \text{Eq. (4)}$$

Exercise

Based on the steps given, create the binary image shown in Figure 3 and write a piece of code that can be used to count the number of cross-shaped region.



(a) Binary Image (b) SE^4 (c) SE^{4C} (c) SE^{4CO}

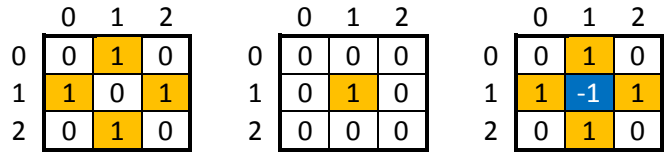
Figure 4

So far, we have been only using '0' and '1' to define the SE. Those with '0' are ignored or we just do not care (usually we just called it don't care) about them during the operation, whereas those with '1' are used to indicate the pixels that we will need to check. Up to this point, we are only concerned about the foreground pixels in the binary image. But in the case of hit-or-miss transformation, it is also trying to match the background. Hence, the '0' in the SE we created is no longer ignored during the operation. In the previous example, the purpose of taking the complement of the binary image and the cSE is in fact to match the background pixels.

Take the binary image shown in Figure 4(a) as an example. If this round we are to find all cross-shaped regions with a hollow center, and we would like to also include the one covers by the square-shaped region. Do you think we can achieve this by using hit-or-miss transformation with the SE shown in Figure 4(b)?

The answer can be YES or NO, depends on what you approach this. If you are using SE^{4O} as your first SE and SE^{4C} as your second SE, then the answer is NO. But if you are using SE^{4O} with SE^{4CO} then the answer is YES. In this case, SE^{4O} ensures that the north, south, east, and west pixels in a region must have a '1' in each of them (or they must be foreground pixels in the region). Then SE^{4CO} ensures that the center of the region must have a '0' (or it must be a background pixel in the region). The northeast, northwest, southeast and southwest pixels will be ignored during the hit-or-miss transformation, which also means that they can take the value of either '0' or '1'. Hence, a SE needs to be capable of defining 3 different conditions, (i) foreground pixels that need to be matched, (ii) background pixels that need to be matched, and (iii) pixels that should be ignored during the operations.

Instead of writing your own code to perform hit-or-miss transformation, we can also choose to use the `cv2.morphologyEx` function from OpenCV. But this function only accepts one SE. So how do we combine SE^4 and SE^{4CO} into one and use it as an input to the function? As shown in Figure 5 is an example on how to combine the two. Remember that SE^4 and SE^{4CO} define the foreground pixels and background pixels they will try to match respectively. When combining the two, those pixels that need to be checked against the foreground pixels will remain as '1', whereas those pixels that need to be checked against the background pixels will change to '-1'. The rest which we do not care (don't care) will stay as '0'. Because the SE has to store a negative number now, we can no longer set the variable type to `uint8`. In this case, you can choose to use `int` instead of `uint8`.



(a) SE^4 (b) SE^{4CO} (c) Combined

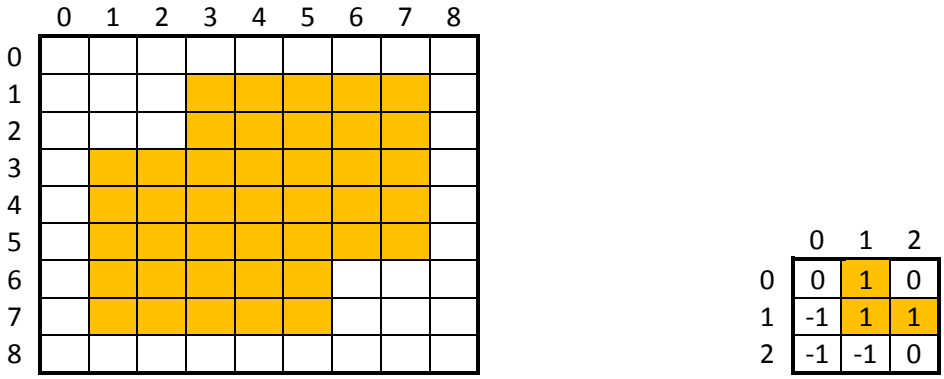
Figure 5

Once everything is ready then you can use the cv2.morphologyEx function to complete the hit-or-miss transformation. As shown below is an example on how to use the function. Please try the following function and compare the result with the one that you obtained from previous exercise.

```
1  sE = np.array([[0,1,0],[1,-1,1],[0,1,0]],dtype=np.int)          #Create the SE.
2  output = cv2.morphologyEx(binary_image, cv2.MORPH_HITMISS, sE)  #Apply hit-or-miss.
```

Exercise

Create the binary image shown in Figure 6(a). Find the location of all the right-angle corner in the binary image using hit-or-miss transformation. To start with, you can use the SE shown in Figure 6(b) to find one type of right-angle corner from the image. In other words, you need to repeat the process four times to cover all four types of right-angle corner. At the end, combine the results to obtain the map that shows the location of all the right-angle-corners.



(a) Binary Image (b) SE

Figure 6

D. Thinning

We will need to use hit-or-miss transformation when comes to thinning. Hence, please ensure that you have completed the previous exercises before attempting this. Thinning is used to remove boundary pixels while preserving the shape of a region (or object). This can be helpful when comes to skeletonization (to find the base shape of a region) and reducing the width of edges extracted from an image. Indirectly, this can also help to simplify the processing (if there is any processing that needs to be carried out after that).

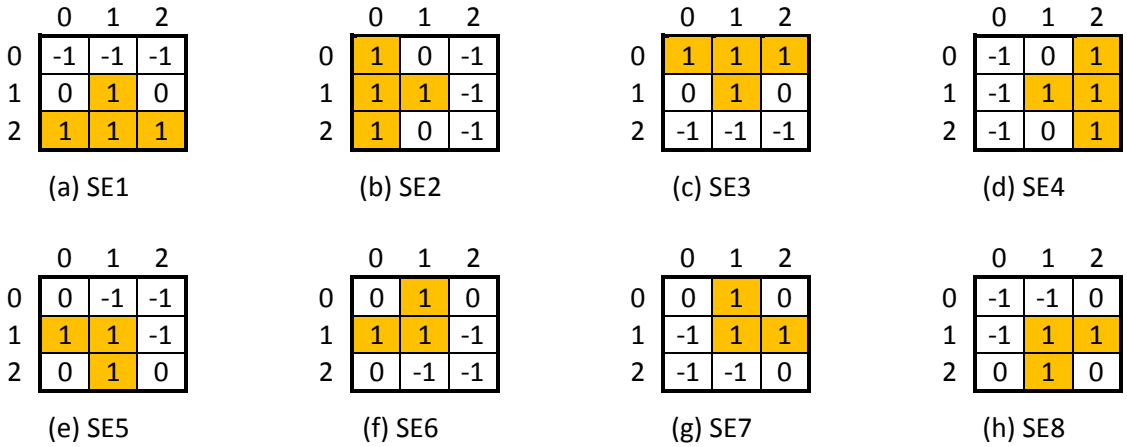


Figure 7

We have to use a set of SE to complete the thinning task. As shown in Figure 7 is a set of SE that can be adopted. This set of SE will try to remove the boundary pixels from eight different directions. Generally, SEs shown in Figure 7(a) to (d) will handle straight line in the north, south, east, and west directions, whereas the SEs shown in Figure 7(e) to (h) will handle the right-angle corners in the northeast, northwest, southeast, and southwest directions. As mentioned earlier, thinning is completed with the help of hit-or-miss transformation and can be mathematically represented by Eq. (5), where A is used to represent the binary image and SE is the set of SE that is used.

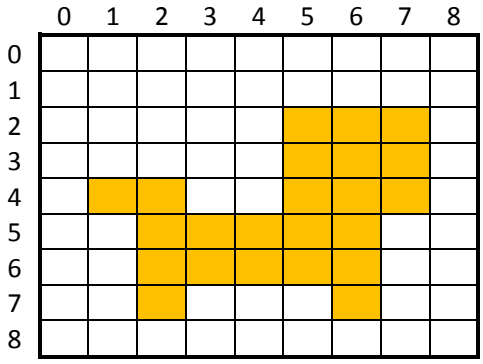
$$\text{Thinning}(A, SE) = A - (A \circledast SE) \qquad \text{Eq. (5)}$$

The general idea of thinning is that, we will take one of the SEs that we have created, for example the one shown in Figure 7(a), and slide it across every single pixel in the binary image. If we found an exact match to the SE, then a '0' will be output at the origin. Otherwise, nothing should change, and we just move the SE to the next pixel in the sequence. This is different from the dilation or erosion, where we always output a '1' at the origin. Hence, this is something that you need to remember when comes to thinning. After we have done it with the first SE, we will move on to use the second SE, for example the one shown in Figure 6(b). The entire operation will continue until there is no more changes to the binary image no matter which SE you apply (until convergence). If you have used all the SEs shown in Figure 7 and yet to reach the convergence state, then we will start again with the first SE and so on.

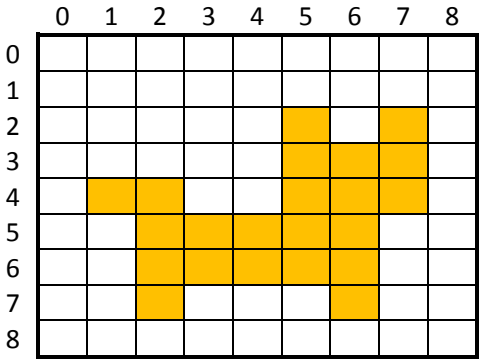
As shown in Figure 8 is a very simple example on how thinning works when using the set of SE shown in Figure 7. There is no effect after SE4 and SE8 were applied and hence they are omitted. We will not be able to reach the convergence state after applying all the SEs for one round. Therefore, we need to apply SE1 to the binary image again. We should reach the convergence state after applying SE5 for one more time.

Exercise

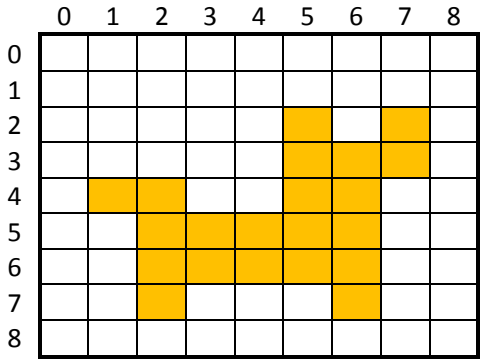
Try to write a piece of code to complete the thinning operations illustrated in Figure 8 to check if my final answer is correct.



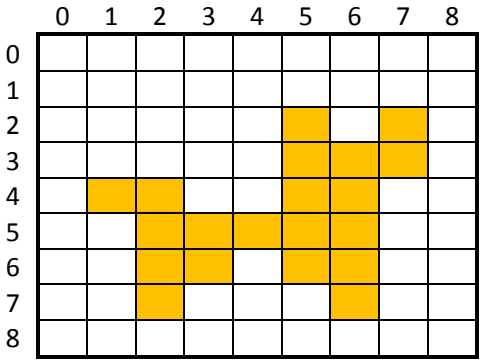
(a) Binary Image



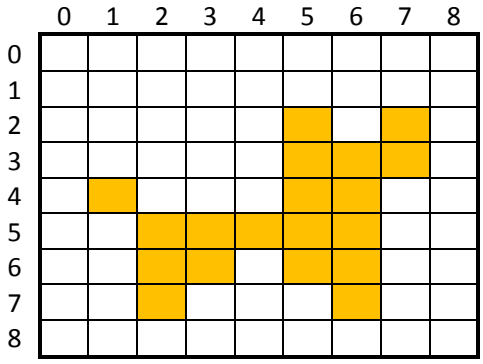
(b) After SE1



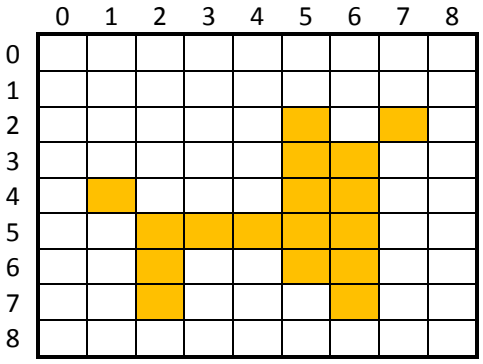
(c) After SE2



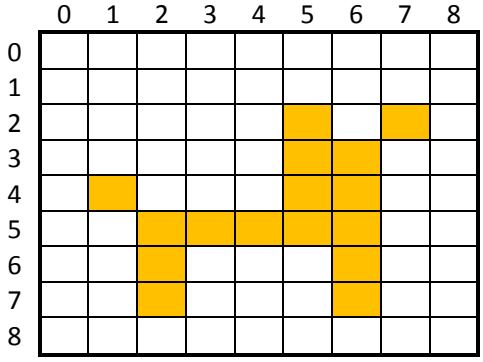
(d) After SE3



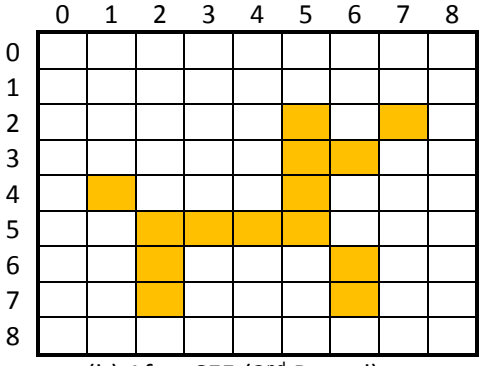
(e) After SE5



(f) After SE6



(g) After SE7



(h) After SE5 (2nd Round)

Figure 8