EE569: Homework #3

Yifan Wang
wang608@usc.edu #3038184983

February 11, 2019

1 Geometric Modification

1.1 Geometric Transformation

Assumption:
1. Holes and patches are rectangle.
2. Which patch to which hole is known.

$Figure 1(a)$ shows my processing procedure.

Step 1: Corner detection on $lighthouse\#.raw$ is different from $lighthouse.raw$. For $lighthouse\#.raw$, corner detection is inspired by FAST from [1][2]. Here a 4 point simplified version is used. Then, suppressing other corner points only keep 4 on image borders.

While considering $lighthouse.raw$, due to the fact that corner points need is all on carefully placed white square patches. For simplicity, template matching method is used, to find points in a 3*3 window which have 3 connected totally white pixels. Templates look as follows:

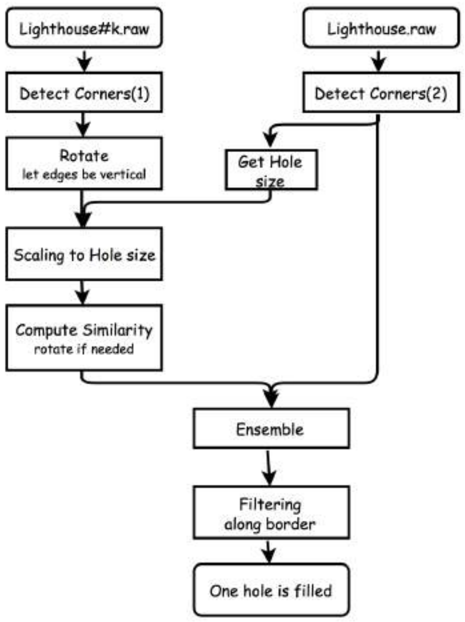
$$\begin{bmatrix}1&1&0\\1&1&0\\0&0&0\end{bmatrix},\begin{bmatrix}0&1&1\\0&1&1\\0&0&0\end{bmatrix},\begin{bmatrix}0&0&0\\1&1&0\\1&1&0\end{bmatrix},\begin{bmatrix}0&0&0\\0&1&1\\0&1&1\end{bmatrix} \tag{1}$$

Which can be realized by binary tree search used in Problem 2 basic morphological process implementation or just $if - else$. All corner points' coordinate (pixel coordinate system) are shown in $Table 1$

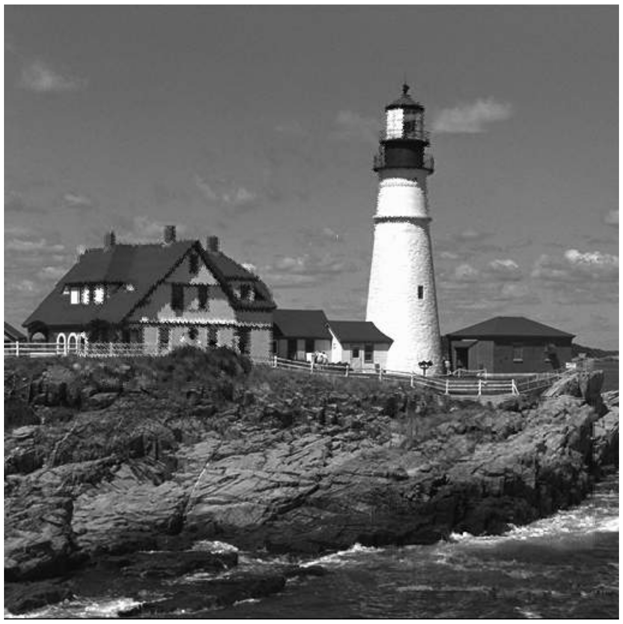
Image	Corner Point (u, v)			
$lighthouse.raw$	$(31, 278)$	$(31, 437)$	$(157, 62)$	$(157, 221)$
	$(190, 278)$	$(190, 437)$	$(316, 62)$	$(316, 221)$
	$(328, 326)$	$(328, 485)$	$(487, 326)$	$(487, 485)$
$lighthouse1.raw$	$(223, 183)$	$(118, 221)$	$(80, 116)$	$(185, 78)$
$lighthouse2.raw$	$(211, 109)$	$(117, 218)$	$(8, 125)$	$(101, 15)$
$lighthouse3.raw$	$(74, 146)$	$(39, 244)$	$(26, 210)$	$(61, 26)$

Table 1: Corner Points
visualied on images in $Appendices A.1$ $Figure 8$

Step 2: a rotation is applied to $lighthouse\#.raw$ to make its borders become vertical or horizontal ($Appendices A.1$ $Figure 9$). Rotation angle is calculated from slope driving from corner points. Interloping is done on rotated image for pixel do not have values by choosing medium value among neighbors. At this step, whether rotated to right angle that fit the hole perfectly matters less, what I need is let image patches become vertical or horizontal for convenience of scaling.



(a) Procedure



(b) Result

Figure 1: Geometric Transformation

Step 3: using scaling method to scale image patch to fit hole size exactly. Scaling factor is calculated by corner points on patches and holes as well. If patch is not square, scaling factor would be calculate among longer borders (less error comparing with the one calculated on short borders). ($Appendices A.1\ Figure 10$)

Step 4: to filling holes, problem that image patch rotated for rectangle image (0, pi), for square image (0, $\pi/2$, pi, $3\pi/2$) all fit perfectly in size, but it may occur that tower would be put in wrong direction, in that case right angle should be decided. So that similarity is calculated among one border of the hole with each matched border (same length) of image patch to decide angle each patch should be rotated. Similarity is calculated by using L_2 norm among gray values, and choose the smallest one. ($Appendices A.1\ Figure 11$)

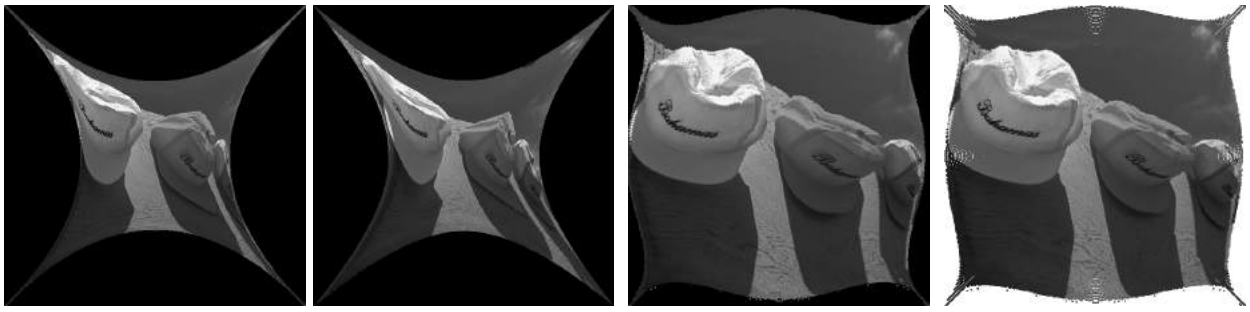
Step 5: ensemble patches to hole, inconsistency would occur on borders ($Appendices A.1\ Figure 12$). To make it look nicer, an averaging process is used ($Figure 1(b)$). Averaging process is done only on vertical direction while dealing with a horizontal borders, and horizontal direction when dealing with a vertical borders. Mask dealing with borders looks as follows:

$$vertical = \frac{1}{6}\begin{bmatrix}0&0&0&0&0\\1&0&0&0&1\\1&0&0&0&1\\1&0&0&0&1\\0&0&0&0&0\end{bmatrix}, horizontal = \frac{1}{6}\begin{bmatrix}0&1&1&1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&1&1&1&0\end{bmatrix} \tag{2}$$

1.2 Spatial Warping

In Spatial Warping, polynomial provided in discussion section is used to get distortion effect. Firstly, assume origin of 2D image is in center of image, point (x, y) (camera coordinate system), four quadrants are defined as follows:

$$\begin{bmatrix}(-,-) & | & (+,-)\\ ---- & (0,0) & ----\\ (-,+) & | & (+,+)\end{bmatrix} \tag{3}$$



(a) inverse function (b) reverse mapping (c) mapping back (explicitly inverse) (d) mapping back (iterative)

Figure 2: Spatial Warping, arc height = 128

1.2.1 mapping from $Original$ to $SWarpped$

What is given: height of arc is 128 with highly symmetric parameter in x and y direction. Suppose the height of square image $Original$ is H $(H > 256)$.

Using inverse function:

Some special point mapping pairs can be found (from (x, y) to (x_d, y_d)):

$$
\begin{gathered}
(0, -\frac{H}{2}) \longrightarrow (0, -\frac{H}{2} + 128) \\
(\frac{H}{2}, -\frac{H}{2}) \longrightarrow (\frac{H}{2}, -\frac{H}{2}) \\
.....
\end{gathered} \tag{4}
$$

6 specified pairs can be found if dividing image into 4 triangles. 12 unknown with 12 equations (given in homework note), it would be super easy to use basic linear algebra knowledge to get coefficients for one particular triangle. Repeat this process for all triangles to get all coefficients needed. Appling these coefficient to particular triangles in image, final result comes. $Figure 2(a)$

Using reverse mapping:

In this case, mapping should change direction from the one using inverse function:

$$
\begin{gathered}
(0, -\frac{H}{2} + 128) \longrightarrow (0, -\frac{H}{2}) \\
(\frac{H}{2}, -\frac{H}{2}) \longrightarrow (\frac{H}{2}, -\frac{H}{2}) \\
.....
\end{gathered} \tag{5}
$$

Using reversing mapping method ($Algorithm 2$), result is shown in $Figure 2(b)$. From this case, there is no need to worry about aliasing problem caused by reverse mapping, since image becomes smaller. Result is slightly better than using inverse function. especially in image borders.

1.2.2 mapping from $SWarpped$ to $Original$

Considering that exactly which point match to which point is known. Using $Equation (5)$, others remains the same by using 4 triangles. This method needs to compute inverse mapping function explicitly. Result (after interpolation) is shown in $Figure 2(c)$, (no interpolation one is shown in $Appendices A.2$ $Figure 13(a)$).

If more pairs of matching points (none of any 3 is on a single line) rather than 6 are available, a better result can be achieved by making formula loner (by adding $a_6x^3 + a_7x^2y + a_8x_y^2 + a_9y^3...$) (a flavor of truncate effect on Taylor expansion). However, in real world, other matching point would be difficult to find, at least not precision.

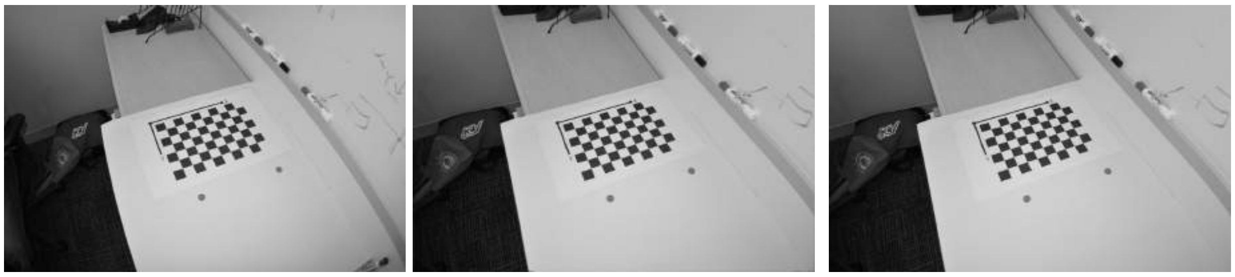

(a) Distorted (b) Corrected ($Algorithm1$) (c) Corrected ($Algorithm2$)

Figure 3: Lens Distortion Correction

Another way to mapping back is to divide the image into more triangles (just like using lots of triangles or regular polygon to approximate a circle), for example 8. In this method, some unknown mapping pairs are need to specified as well. In general this way remains the same flavor as previous one.

Besides, using iterative method like Newton's method ($Algorithm1$) can achieve similar effect ($Figure2(d)$), as well. In this method, using iteration to approach inverse function calculated in above two method, which can be pleasing without compute inverse function explicitly (especially when inverse function do not exist). Intermediate process is show in $AppendixA.2$ $Figure13(b)$.

1.2.3 More

In above method, it can be found that borders shrinks more than center part. If using a equation similar as $Equation(7)$ in next section, Another different result that meets the requirement is generated ($AppendixA.2$ $Figure13(b)$). In this image, center part would shrink more.

1.3 Distortion Correction

Algorithm 1

```
1: procedure
2: FOR EACH PIXEL DO:
3:     $x_{tmp} \leftarrow x_d$
4:     $y_{tmp} \leftarrow y_d$
5: repeat until convergence:
6:     $r \leftarrow x_{tmp}^2 + y_{tmp}^2$
7:     $R_d \leftarrow \frac{1}{1+k_1r+k_2r^2+k_3r^3}$
8:     $x_c \leftarrow x_d R_d$
9:     $y_c \leftarrow y_d R_d$
10:    $x_{tmp} \leftarrow x_c$
11:    $y_{tmp} \leftarrow y_c$
12: return:
13:    $(x_c, y_c)$
```

Algorithm 2

```
1: procedure
2: INPUTS:
3:     Distorted Image $D$
4: Blank Image R
5: for $i$ in range $(0, Height)$:
6:     for $j$ in range $(0, Width$:
7:         convert $(i, j)$ to camera coordinate as
8:           $(i_{cam}, j_{cam})$
9:         $r = i_{cam}^2 + j_{cam}^2$
10:        $R_d = 1 + k_1r + k_2r^2 + k_3r^3$
11:        $x_{cam} \leftarrow i_{cam}R_d$
12:        $y_{cam} \leftarrow j_{cam}R_d$
13:        convert $(x_{cam}, y_{cam})$ back to pixel
14:          coordinate as $(x, y)$
15:        $R(i, j) = D(x, y)$
16: return:
17:    $R$
```

When given a distorted image, distortion correction for radial distortion with k_1 and k_2 like this case is calculated by transferring image into camera coordinate system through camera's intrinsic parameters f_x, f_y,

c_x, c_y. For a distorted image's pixel (u_d, v_d) in pixel coordinate system, its coordinate in camera coordinate system (x_d, y_d) is computed by:

$$x_d = \frac{u_d - c_x}{f_x}$$
$$y_d = \frac{v_d - c_y}{f_y} \tag{6}$$

corrected coordinate (x_c, y_c) is:

$$x_d = x_c(1 + k_1 r^2 + k_2 r^4)$$
$$y_d = y_c(1 + k_1 r^2 + k_2 r^4) \tag{7}$$

where $r^2 = x_c^2 + y_c^2$; final result (u_c, v_c) in pixel coordinate system is:

$$u_c = f_x x_c + c_x$$
$$v_c = f_y y_c + c_y \tag{8}$$

In distortion correction, to find inverse function, $Algorithm1$ is used (based on Newton's Method): let (x_d, y_d) represent pixel at image (x, y) in distorted image D; (x_c, y_c) represent pixel at image (x, y) in undistorted image R.

$Algorithm1$ returns approximation of (x_c, y_c) in regard to(x_d, y_d). Set pixel value $R(x_c, y_c) = D(x_d, y_d)$. Interpolation is used for after get corrected image ($Appendices A.3$ $Figure 14$). $Figure 3(b)$ show final result.

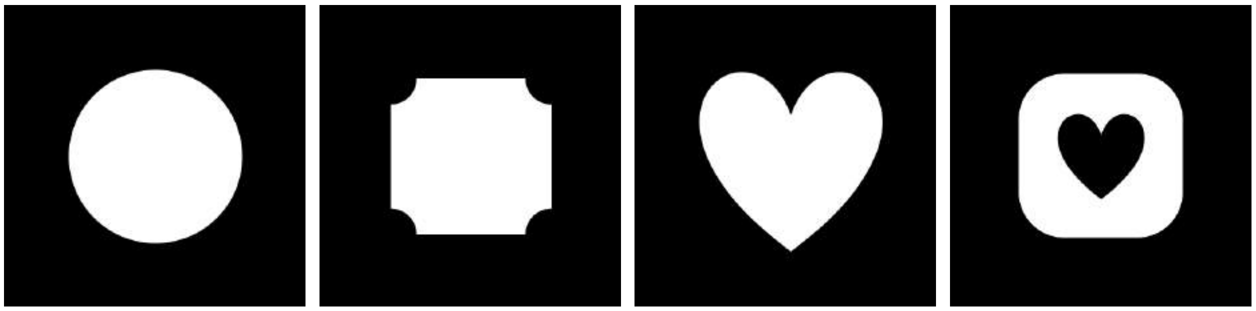
For each pixel, 20 iteration is enough to get nice approximation of (x_c, y_c). For a $H \times W$ image, time complexity is $O(HW)$.

1.3.1 More

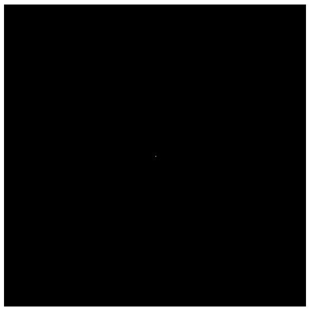
Another tricky but simple solution to this problem is using reversing mapping ($Algorithm2$). Which can be explained as given a blank image R which has same size as distorted one (D). For each location (i, j) in undistorted image using $Equation(7)$ find a coordinate (x, y) in distorted image. Then arranging pixel value $D(x, y)$ to $R(i, j)$ would finish correction task. ($Figure 3(c)$). It's a super simple method which can achieve similar result (but different) as the one using iterative methods. Major difference occurs obviously when image have texture change in a small region, aliasing would occur due to the simple operation of $Algorithm2$.

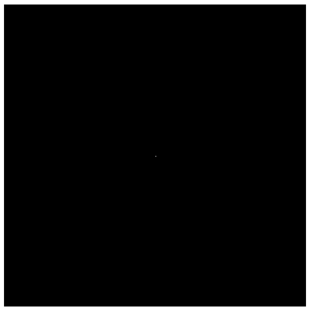
2 Morphological Processing

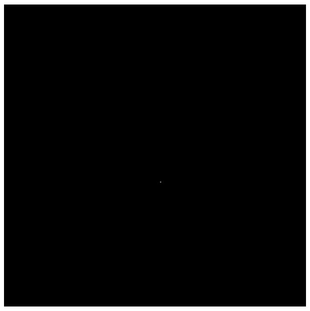
2.1 Basic Morphological Process Implementation

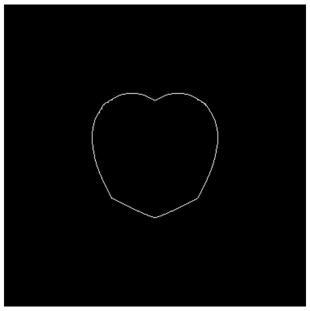

(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

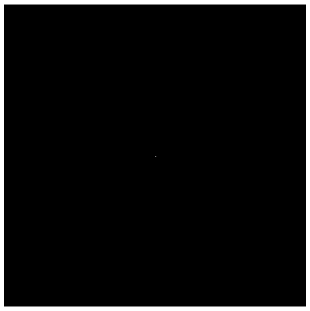
Three basic morphological process is realized in a single function named $MorphologicalProcessing$ with parameter Opt to control which process to use.

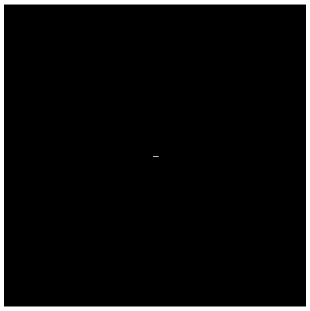

(e) Shrinking: pattern1

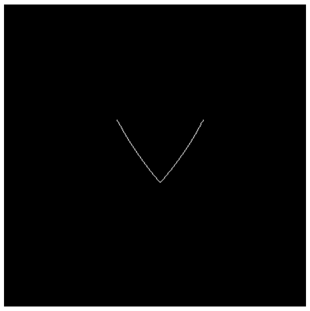

(f) Shrinking: pattern2

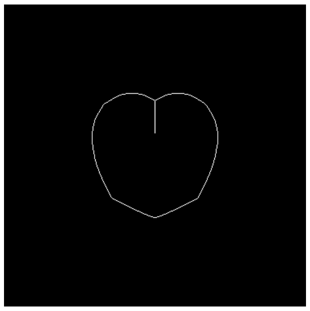

(g) Shrinking: pattern3

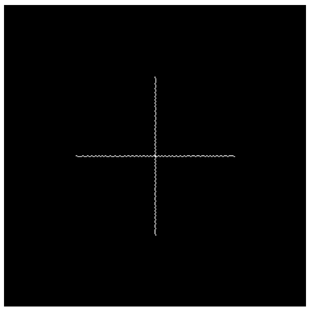

(h) Shrinking: pattern4

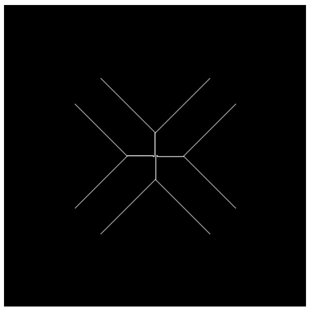

(i) Thinning: pattern1

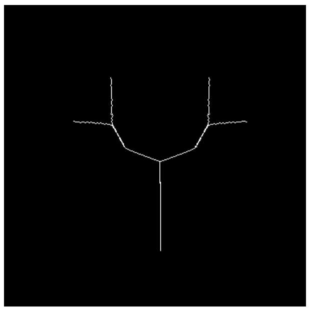

(j) Thinning: pattern2

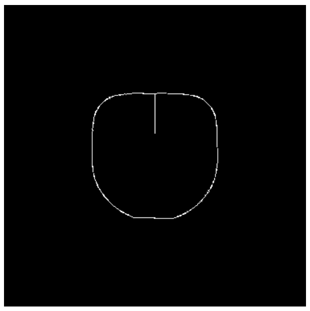

(k) Thinning: pattern3

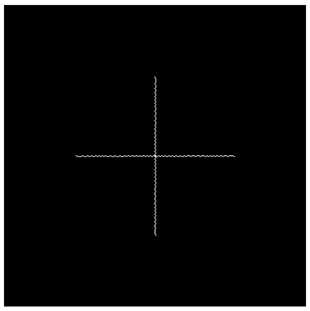

(l) Thinning: pattern4

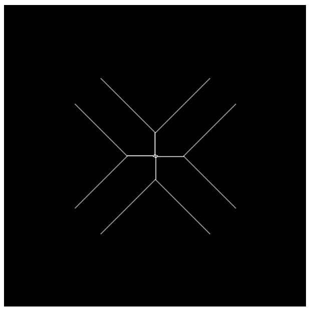

(m) Skeletonizing: pattern1

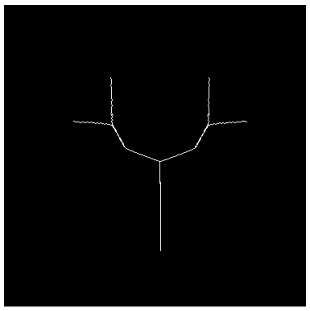

(n) Skeletonizing: pattern2


(o) Skeletonizing: pattern3


(p) Skeletonizing: pattern4


(q) Skeleton+Bridge: pattern1


(r) Skeleton+Bridge: pattern2


(s) Skeleton+Bridge: pattern3

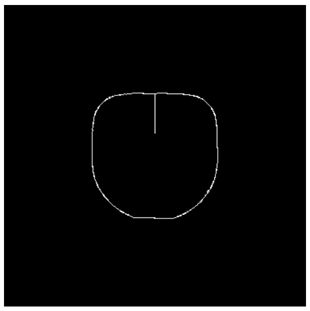

(t) Skeleton+Bridge: pattern4

Figure 4: Shrinking, Thinning, Skeletonizing result on four patterns

2.1.1 Realization

Detail realization is based on binary tree search. All conditional masks are built in one single binary tree and another two tree for unconditional masks (one for ST another for K).

Build a tree for conditional mask:

When building a conditional binary tree, all masks are changed into sequences in sequential order text book provides and given it a string label ("S__", "_T_", "__K" for thinning, shrinking and skeletonizing in my realization) to determine its function. Labels would be combined if several different masks share a same node ("ST_" if this node only has thinning and shrinking mask).

Using binary tree search can achieve faster speed, since all mask are combined, once not matched, it would end search process on this pixel, on the other hand, using other method like $switch$ would just end search on this mask and would do same search on other masks have similar flavor which would use much longer time.

For example, image stack is "1010", image masks are "1100" and "1101", in binary tree search the first node only have right branch, and it miss, it would return immediately, only 2 bit comparison is needed. While non-tree search, the first mask needs 2 bit comparison and second mask needs 2 as well, 4 in total. When masks is large and image is large, this difference would have huge impact on running time.

Build a tree for unconditional mask:

For unconditional cases, M is set to be one which can be derived from logical equation from text book. A "d" is used to represent not care, when meet this case in building tree, recursive method would be used to generate this node in both left and right direction. And for conditions like $A \cup B = 1$, they are separated into 2 conditions: $A = 1, B = d$, and $A = d, B = 1$.

After building up binary tree, all remaining process is the same as what book provided [3].

End of iteration:

To determine whether to end iteration or not, a bool indicator is used to record whether changes have been made on previous iteration, if not, end iteration. Besides a counter is used to end iteration when iterated 2000 times. (for image in general resolution, it would never needs so many iterations).

Using this method achieves a moderate speed with about 0.006 second on each iteration on xcode 10.1 (debug mode, using release mode or cmake with g++ -O3 would give much faster speed, about 0.1sec for $pattern1$ image, $OpenMP$ is supported to make it faster as well). And it would generally need need about 100 iterations ($pattern1$) before finishing. Results are shown in $Figure4$. Result of 20 iterations is shown in $AppendicesB.1$ $Figure15$.

Time complexity is $O(HW)$, for the worst case (all white) it would reach about $O(log_2(HW)HW)$.

Result from this method have slightly difference with $skimage.morphology.skeletonize$ in $Python$ or $bwskel()$ in $Matlab$, their results are thinner more approximate to $thinning$ which can be explained that they have more optimizations.

2.1.2 Explanation

These 3 operation can be regard as using some edge detector to get edges, then based on some requirement to decide whether an edge pixel should be removed (or erosied). The difference comes from different edge detectors and keeping criterions.

Major difference among these 4 patterns between shrinking and thinning comes from $Bond1, 2$ which would shrink a line ($pi/2$ and 0). This explains why for shrinking both pattern 1 and 2 only left a point; while for thinning, they become $+$ or $-$ shape. The reason why pattern 2 does not become a vertical line in thinning is decided by definition of $CornerCluster$ in unconditional mask. If can be a vertical line if changing this mask (this mask serves as decide which pixel to keep when 4 pixel cluster in square, text book provides the one keeps right bottom pixel).

Major difference among these 4 patterns between thinning and skeletonizing comes from $Bond5$ which would shrink a line ($pi/4$ and $-pi/4$). Which leads to the difference in pattern4 where bottom of hearts become almost a horizontal line.

For Skeletonizing, result for pattern4 contain undesirable short spurs which might comes from none symmetric shape which comes from this template matching method is just an approximation of medial axis skeleton.

2.2 Defect Detection and Correction

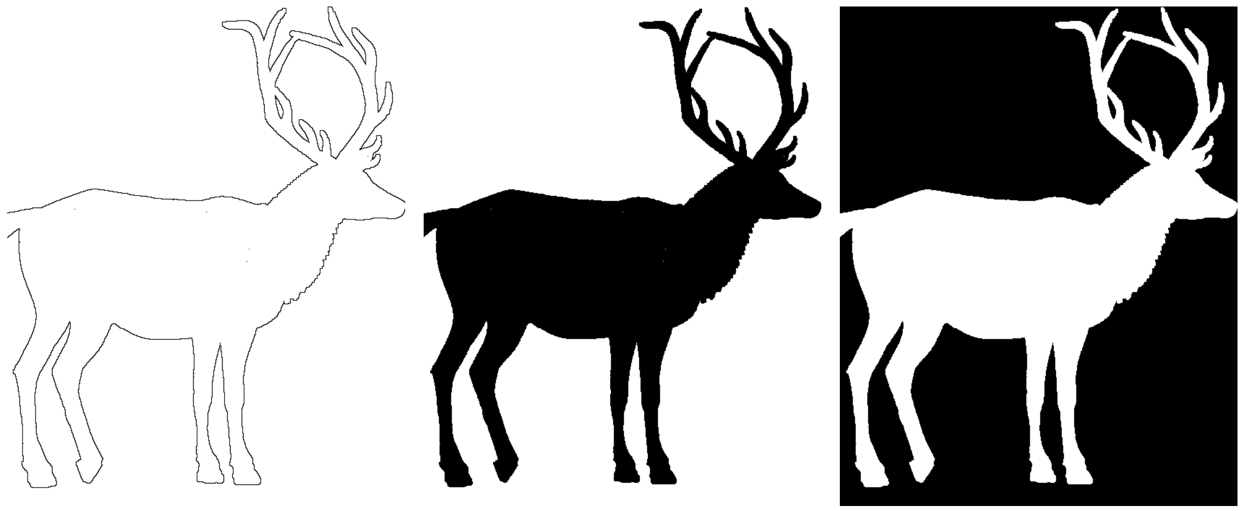

(a) intermediate image (b) labeled defects (yellow part) (c) deer with defects corrected

Figure 5: Defect detection

Assumption:
1. All defects are smaller than needed small, separated background.

First idea comes when I saw this problem is to use board first search (BFS) algorithm or depth first search (DFS) algorithm to find all black parts and using a threshold to remove regions whose area are less than threshold (based on my assumption that defects would not be larger than small backgrounds). Using basic morphological processing methods which might be sensitive to noise and lose some coordinate information. While using DFS or BFS can get both size and location information.

However, due to the size of image, using either method directly would lead to stack overflowing. And even not overflow, would require lots of time. In that case, preprocessing is required (remove some point that is not defect certainly).

Step 1: A 3×3 all 0 mask is used to scan whole image, if hit, remove center pixel (all black pixel would mean less what I want is only contours which is not as thinning as that from $thinning$ but it keeps location information). Intermediate image of this process is in $Figure4(b)$, objects becomes contours.

Step 2: Apply 8-connected DFS on intermediate image search for black parts. Size of defects' contour would be smaller than objects' contour need. In this case, $threshold = 50$ is used to separate. Using this method can deal with defects whose area is at least less than $threshold$ pixel.

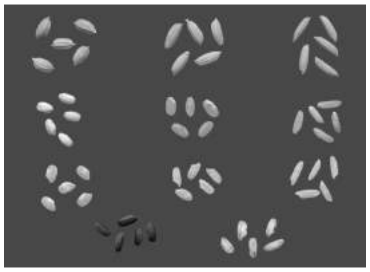
Result: there are 5 defects in deer image, they are all labeled with yellow color in $Figure4(c)$ (inverted for better visualization). $Figure4(d)$ is final result of defect-less image.

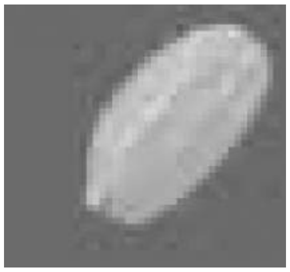
Time complexity is $O(HW)$, since step 1 and step 2 both are $O(HW)$. With the help of preprocessing, it would take far less stack when using DFS which otherwise needs lots of recursive and memory.

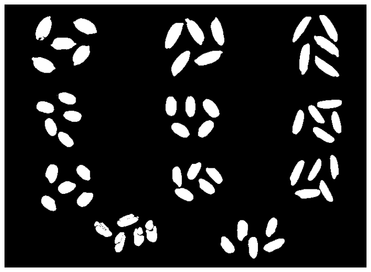
Besides, for this particular case, a super super simple way to get all defects is by using a mask shown in $Equation(9)$. But it makes no sense for general cases and would not be robust to defects larger than 1 pixel.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{9}$$

To make this method more robust, and using morphological possessing as question required. It is feasible to use $shrinking$, instead of let whole image converge, a iterator number is set for example 3 (let defects


(a) Rice Count Step 1: grayscale image


(b) noise around rice


(c) Rice Count Step 2: binarized image

(d) noise

(e) noise 2

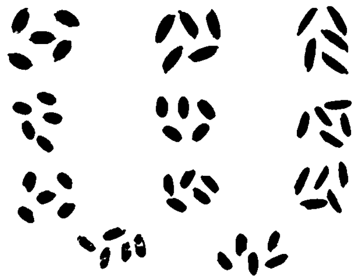

(f) Rice Count Step 3

Figure 6: Process of Rice

converge on shrinking based on $Assumption 1$). After shrinking for 3 times ($Appendices B.2(a)$ $Figure 16$), then using template in $Equation (9)$ to find the defects and their location (based on shrinking's function, all connected part would shrink to a single pixel when converging). With these location found. DFS is used again on original image, start from defects' location recorded.

By using this method with iteration 3, defects whose size less than 27 pixel (ideal circle, if defects is a line 3 iteration can only deal with a 7 pixel line) can be all discarded. More iteration, lager defects can be found. Since in each iteration, one pixel boundary can be erased. Same result can be achieved as $Figure 5(c)$.

2.3 Object Analysis

Assumption:

1. Background accounts for the majority of whole image.
2. Background does not vary dramatically in color.
3. (For question (2) only) shape of rice is convex.
4. (For question (2) only) type number of rice is known (11 in this case).
5. (For question (2) only) each type of rice are geometrically clustered.

2.3.1 Rice Count

To count the number of rice, idea is to binarize image, then using DFS to search whole image with the same flavor as previous section. Tough part is how to binarize image. Here it is separated into several steps:

Step 1: convert color image to grayscale (using function from HW2). Then using a Non-Local Mean filter (from HW1) to get rid of noise around rice ($Figure 5(b)$). Due to NLM filter's property of keeping and enhancing edges, it would give better and more accurate result for binarize.

Step 2: based on $Assumption 1, 2$ that background account for the majority and varies little. Using $Otsu$ [7] method to find threshold and binarize image. ($Figure 5(c)$).

Step 3: after these steps, binarized image looks good. While there remains some noise in image $Figure 5(d, e)$. Even though it can be suppressed through using a threshold, it seems better to git rid of them for the sake of

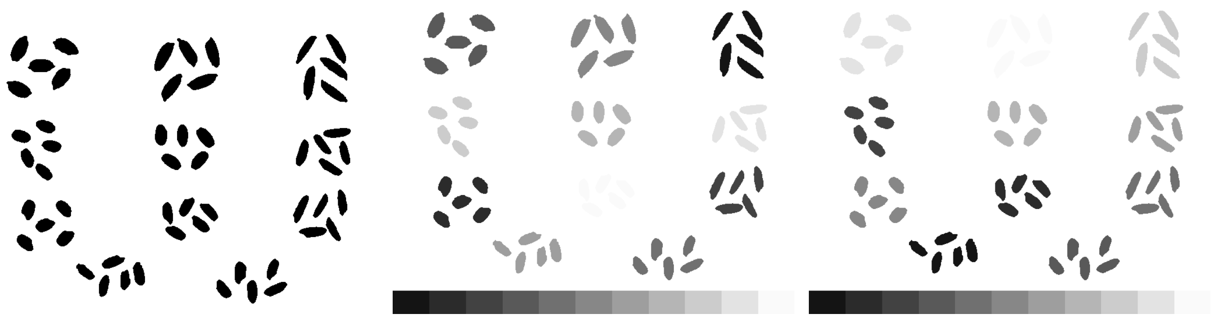

(a) Size Comp Step 2: hole filling (b) Size Comp Step 3: classify rice type (c) Size Comp Step 3: sort

Figure 7: Rice Size Comparsion
different grayscale in (b) means different type
different grayscale in (c) represent different average size, the brighter, the larger

further process. So that a score S is computed for each pixel which based on connectivity:

$$S(i,j) = \sum P(pixel \in 8connected) + 2\sum P(pixel \in 4connected) \tag{10}$$

where all pixel value is $0/1$. Then using a threshold ($T_1 = 7$ used) to decide pixel value $P(i,j)$ based on this score. For the reason of using DFS without overflowing, image is reversed at the same time (search on black part). ($Figure5(f)$).

Step 4: Then using DFS to count the number of black regions. For the sake of counting accuracy, I reject regions area is less than a threshold ($T_2 = 50$), which might caused by residential noise.

Result: final out put of my code is:

```
RiceCount: There are 55 rice.
```

2.3.2 Size Comparison

Size comparison is based on previous section's result. But we should notice there are some holes remaining in rice, which would affect the accuracy of area comparison. So that more process is need before comparing size.

Step 1: get processed image from previous section.

Step 2: based on $Assumption3$ shape of rice are convex, holes are filled in both x direction and y direction on condition that if hole in this particular rice is bounded in either direction. Result of this process is shown in $Figure7(a)$.

Step 3: to classify rice, based on $Assumption4, 5$, how many types of rice exists in this image is given and each type is geometrically clustered. Then, computer the geometric center of each rice (using DFS as usual return rice size as well). K-means algorithm [4][5] is used to cluster rice types. However, initialization is of great importance to final clustering result. Random initialization would perform poorly. In that case, I used the trick from [6] to maximum the distance between each cluster center during initialization. The process are as follows: (1) choose the first rice center as our first cluster center; (2) find a rice who has maximum distance (distance is normalized in each direction) to all cluster center, set it as another cluster center. (3) repeat (2) until get all cluster center is got. Using this method would perform well. Result of clustering is shown in $Figure7(b)$, each different grayscale in rice represent a rice type, same grayscale means same type.

Result: then it would be super easy to compare average rice size (based on number of pixels). Average size of each type output:

```
RiceSize (pixel):
    label: 5,  size: 523.6
    label: 8,  size: 527.4
    label: 9,  size: 537
    label: 2,  size: 549.4
    label: 4,  size: 556.2
    label: 1,  size: 559.6
    label: 10,  size: 565.8
    label: 7,  size: 618.4
    label: 0,  size: 741
    label: 3,  size: 835.6
    label: 6,  size: 872.8
```

where each label refers to grayscale in $Figure 7(b)$ with formula $grayscale = 21 * label$.

Sorting result is show in $Figure 7(c)$, grayscale means the size of rice, the larger grayscale (brighter, as axis below shows, $grayscale = 21 * (11 - sizeRank)$ in this image), the larger average rice size is.

2.3.3 Improvement

After that I consider that if there are no assumption that all types of rice are geometrically clustered which can be found in $RiceSizeFancy$. Using other features like color shape to determine which rice belongs to which type. This problem becomes quite tough due to the variation of color, light and shape among different rice in one single type.

It performs slightly better if I transfer RGB to Lab, however, classification result remains unsatisfactory. The major difficulty is that color is less robust than other features. It is likely to change due to the difference of rice pose and luminance. $AppendicesB.3$ $Figure 17(a)$.

If I regard each rice as a ellipse, then using features like its area, major radius and minor radius, these features work not well due to the different size of one single type of rice.

If combine these features together, it would be difficult to decide the weight of each features in regard to the difference of their measurement (even though it would be relatively easy to normalize major radius and minor radius, but compare them with color to get a good discount weight would be super tough).$AppendicesB.2$ $Figure 16(b)$

Return back to using color as features, it might be feasible to compute color distance with a weight computed by their geometric distance to cluster center (for example, Gaussian). Using this weighted color distance might perform better.

References

[1] E. Rosten and T. Drummond, "Machine learning for high speed corner detection," in 9th Euproean Conference on Computer Vision, vol. 1, 2006, pp. 430–443

[2] [online]`https://en.wikipedia.org/wiki/Features_from_accelerated_segment_test`

[3] William K Pratt, Digital Image Processing 4th Edition pp.432

[4] Hartigan J A, Wong M A. Algorithm AS 136: A K-Means Clustering Algorithm[J]. Journal of the Royal Statistical Society, 1979, 28(1):100-108.

[5] Kanungo T, Mount D M, Netanyahu N S, et al. An efficient k-means clustering algorithm: analysis and implementation[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2002, 24(7):881-892.

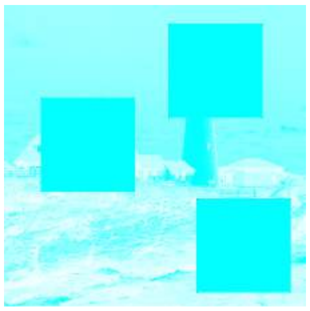
[6] Katsavounidis, I., Kuo, C.-C. J., & Zhang, Z. (1994). A new initialization technique for generalized Lloyd iteration. IEEE Signal Processing Letters, 1(10), 144–146.

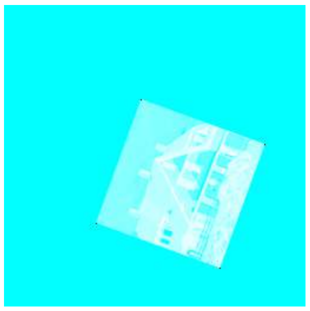
[7] Otsu N. A threshold selection method from gray-level histogram. IEEE Trans,1979;SMC-9;62-66

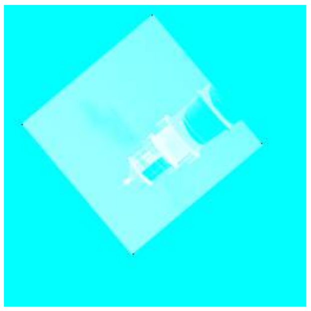
Appendices

A Probelm #1

A.1 Geometric Transformation


(a) 512 × 512


(b) 256 × 256


(c) 256 × 256

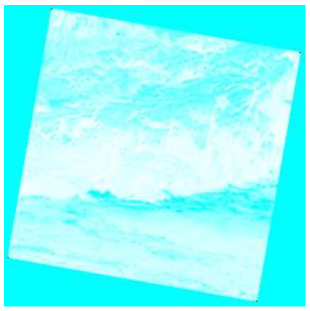
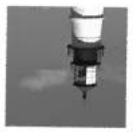
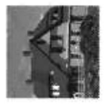

(d) 256 × 256

Figure 8: Geometric Transformation Step 1: corner detection (black point)


(a) 362 × 362


(b) 362 × 362

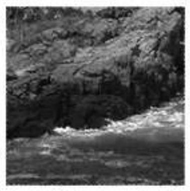
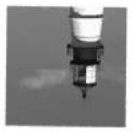
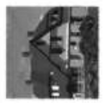

(c) 362 × 362

Figure 9: Geometric Transformation Step 2: rotation 1


(a) 414 × 414


(b) 529 × 529

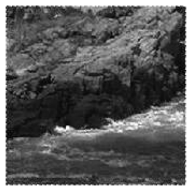
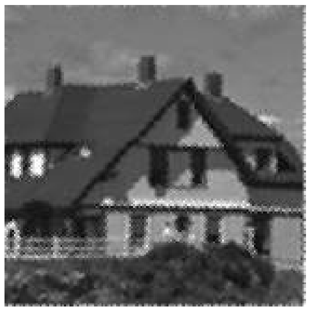
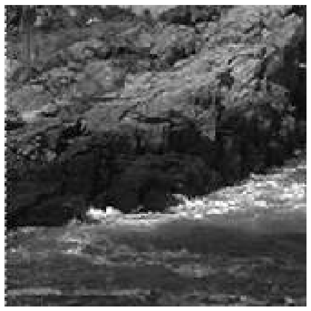

(c) 281 × 281

Figure 10: Geometric Transformation Step 3: scaling

(a) 161×161



(b) 162×162



(c) 164×164

Figure 11: Geometric Transformation Step 4: rotation 2

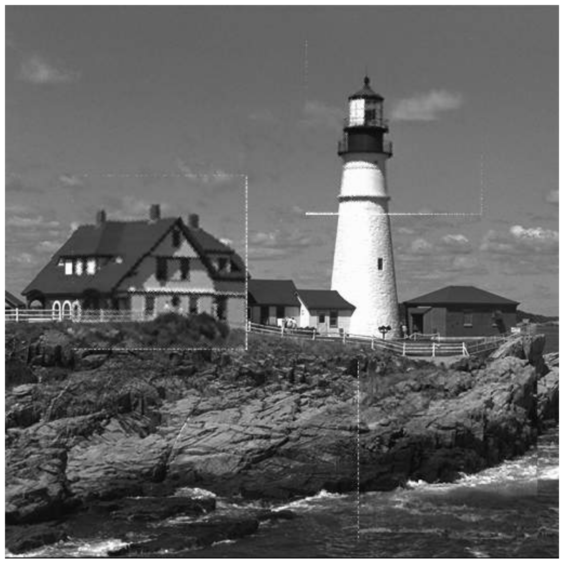
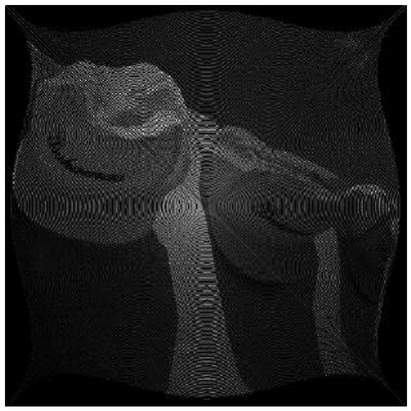
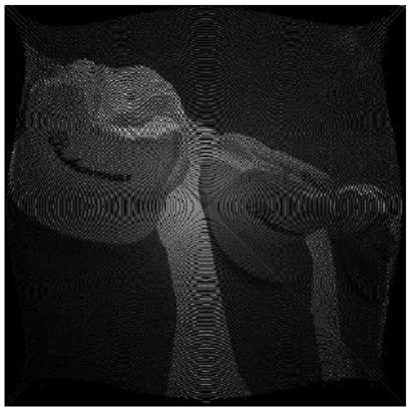


Figure 12: Filling hole without smoothing

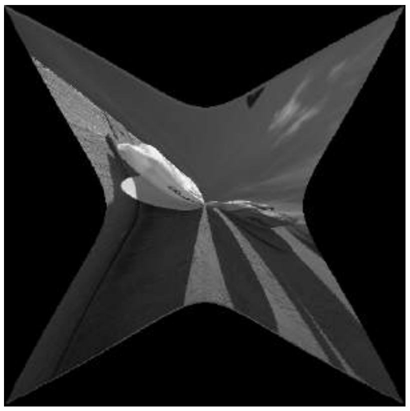
A.2 Spatial Warping



(a) Mapping back (explicit inverse)



(b) Mapping back (iterative)



(c) Using $Equation(7)$

Figure 13

A.3 Distortion Correction

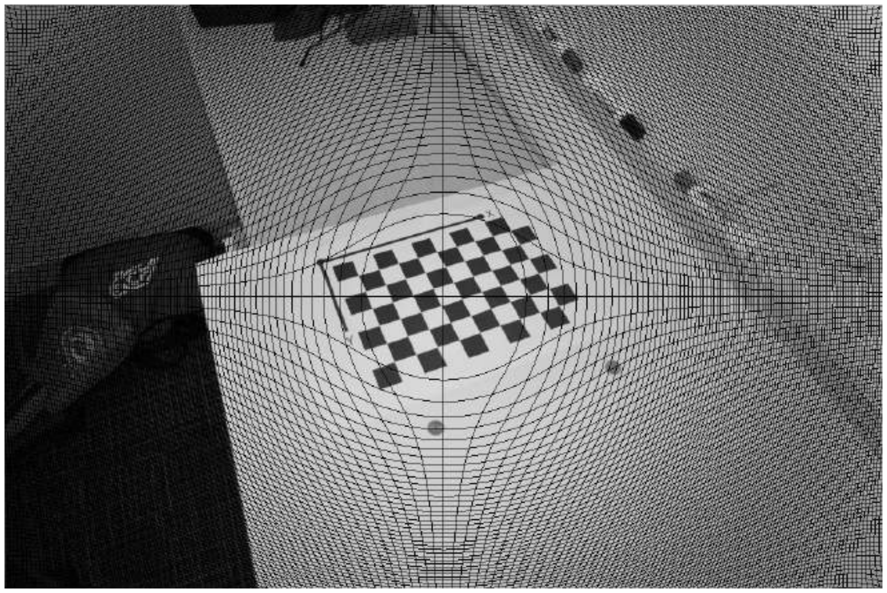
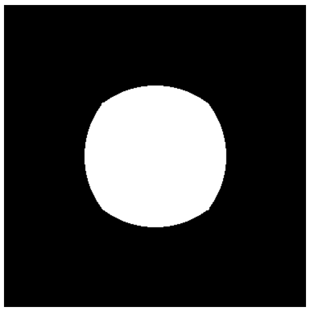


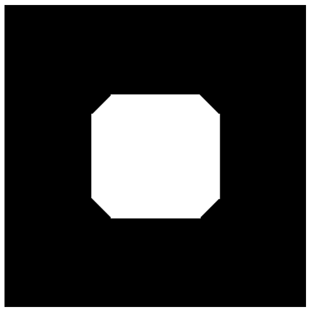
Figure 14

B Probelm #2

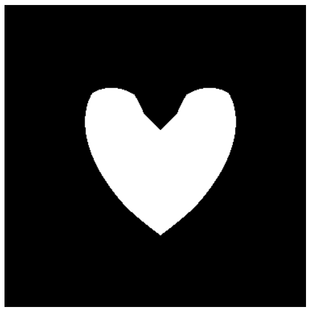
B.1 Basic Morphological Process Implementation



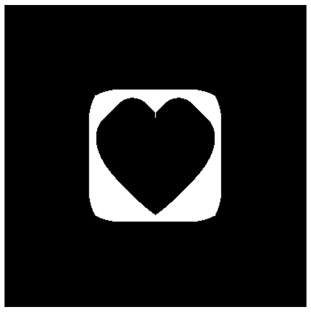
(a) Shrinking: pattern1



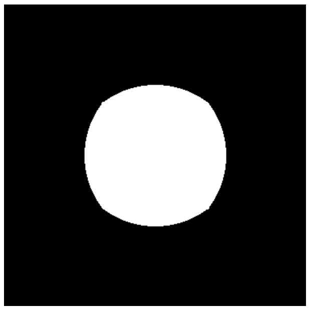
(b) Shrinking: pattern2



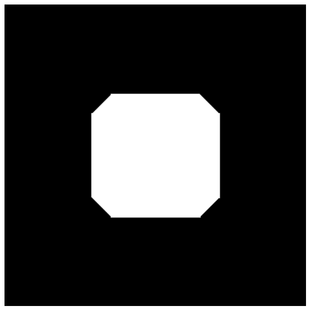
(c) Shrinking: pattern3



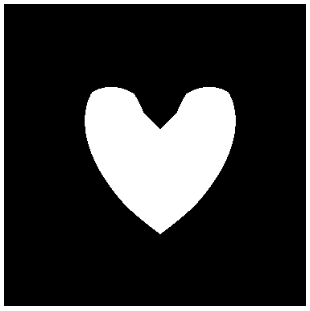
(d) Shrinking: pattern4



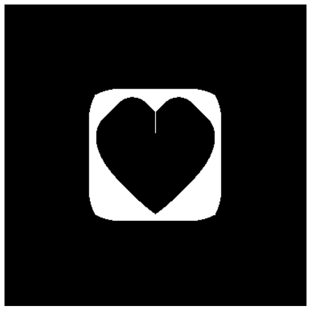
(e) Thinning: pattern1



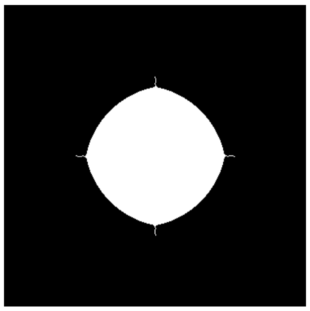
(f) Thinning: pattern2

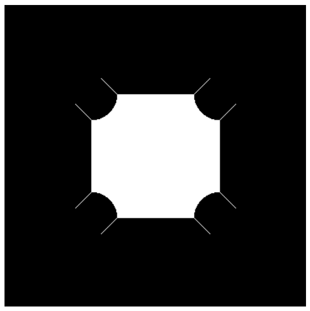


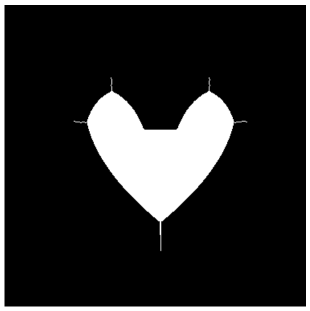
(g) Thinning: pattern3



(h) Thinning: pattern4


(i) Skeletonizing: pattern1


(j) Skeletonizing: pattern2


(k) Skeletonizing: pattern3

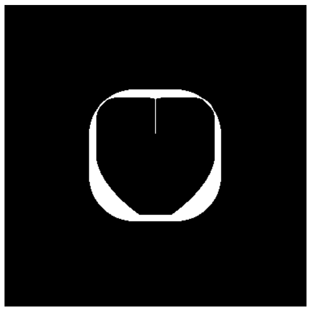
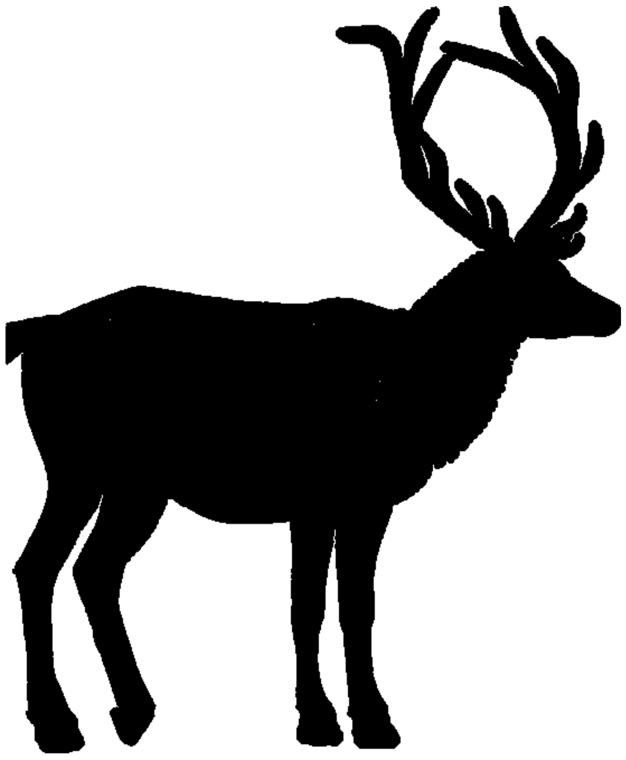

(l) Skeletonizing: pattern4

Figure 15: Shrinking, Thinning, Skeletonizing after 20 iterations on four patterns

B.2 Defect Dtection


(a) 3 iteration

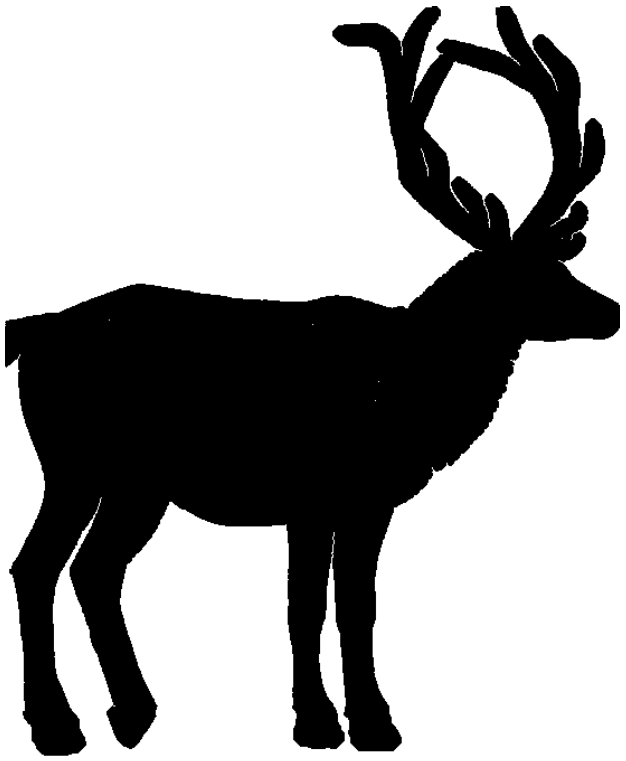

(b) 5 iteration

Figure 16: Different iteration on $deer.raw$
Morphological Processing shrinks on white part, so raw image is inverted

B.3 Object Analysis

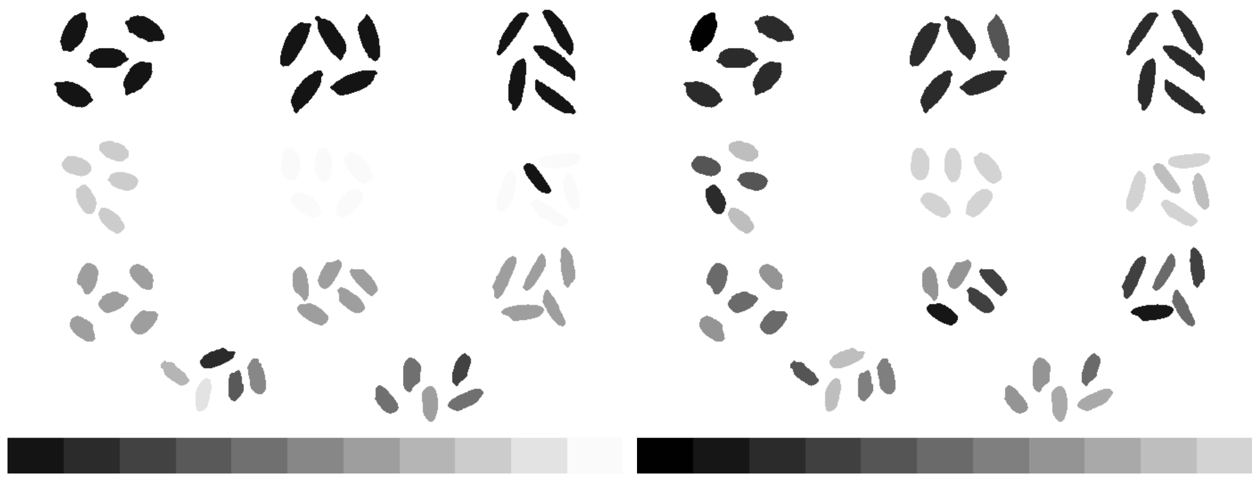

(a) Classify using color (b) Classify using color and shape

Figure 17: RiceSizeFancy Result