2024 NTU DIP HW3

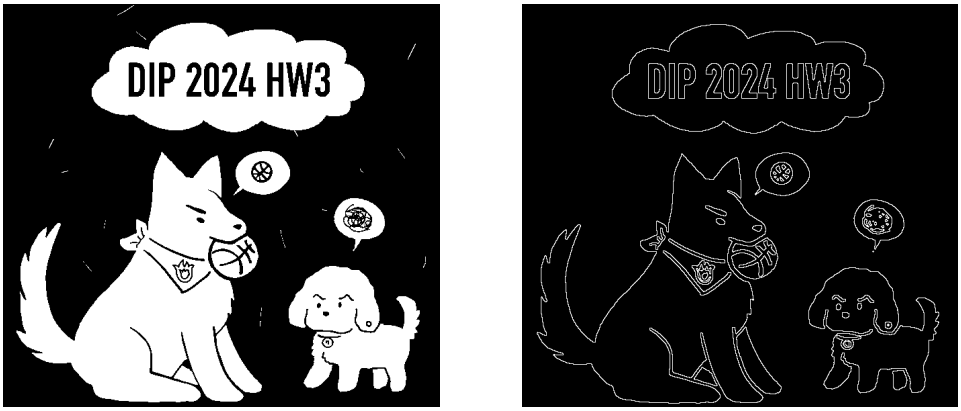
陳德維

April 15, 2024

Problem 1 : MORPHOLOGICAL PROCESSING

(a) Design a morphological processing to extract the objects' boundaries in **sample1.png** and output the result as **result1.png**.

Result



sample1 result1

Figure 1: Boundary Extraction w/ Opening Noise Removal

Approach

首先，我使用了講義上的 Boundary Extraction 的式子：

$$\beta(F(j,k)) = F(j,k) - (F(j,k) \ominus H(j,k))$$

其中我使用的 Structuring Element 為：

$$H = \begin{bmatrix} 1, & 1, & 1 \\ 1, & 1, & 1 \\ 1, & 1, & 1 \end{bmatrix}, \quad origin : (1,1)$$

結果相當的好，如下：

Figure 2: result1 (w/o Noise Removal)

但是 noise 的出現讓畫面變得不太美觀，因此我先做了一次 opening，其式子如下：

$$G(j,k) = [F(j,k) \ominus \tilde{H}(j,k)] \oplus H(j,k)$$

其中我使用的 Structuring Element 為：

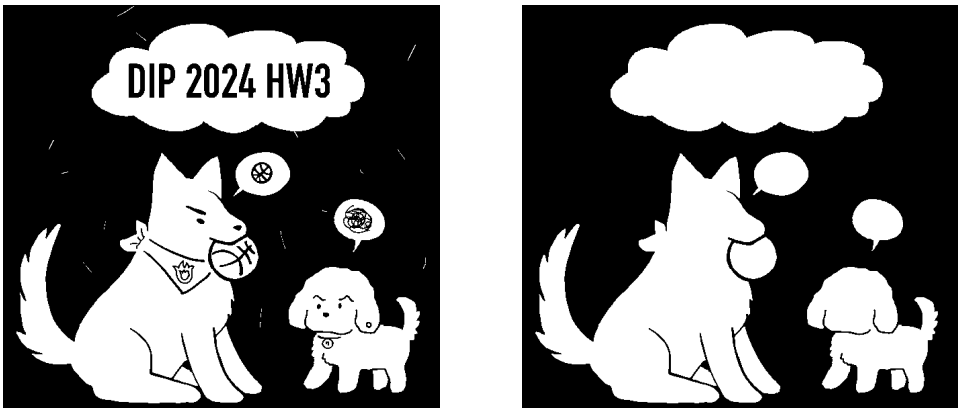
$$H = \begin{bmatrix} 0, & 1, & 0 \\ 1, & 1, & 1 \\ 0, & 1, & 0 \end{bmatrix}, \quad origin : (1,1)$$

Noise Removal 後再做 Boundary Extraction 的結果如下，可以看到背景變得乾淨了，但是某些細節因為 opening 而喪失，例如左邊狗狗圍巾上的火、右邊狗狗的對話框的生氣圖案：

Figure 3: result1 (w/ Noise Removal)

(b) Perform hole filling on **sample1.png** and output the result as **result2.png**.

Result



sample1 result2

Figure 4: Hole Filling w/ Subtractive Operator Noise Removal

Approach

首先，我使用了講義上的 Hole Filling 的式子：

$$G_i(j,k) = (G_{i-1}(j,k) \oplus H(j,k)) \cap F^c(j,k)\ ,\ i = 1,2,3\ldots$$

$$G(j,k) = G_i(j,k) \cup F(j,k)$$

其中我使用的 Structuring Element 為：

$$H = \begin{bmatrix} 0, & 1, & 0 \\ 1, & 1, & 1 \\ 0, & 1, & 0 \end{bmatrix},\quad origin : (1,1)$$

並定義了許多內部座標的做初始定位點 (圖中紅點)，如下：

Figure 5: sample1 feature points

最終的 G_i 如下：

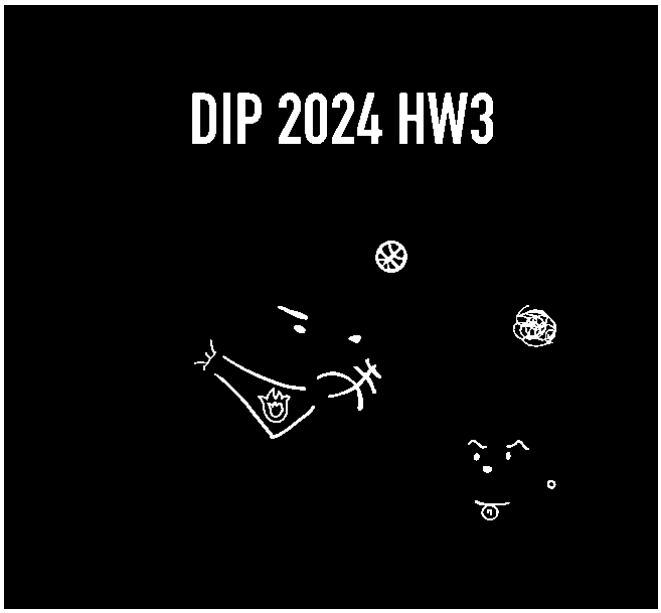


Figure 6: Final G_i in Hole Filling w/ $i = 120$

Hole Filling 的過程如下：

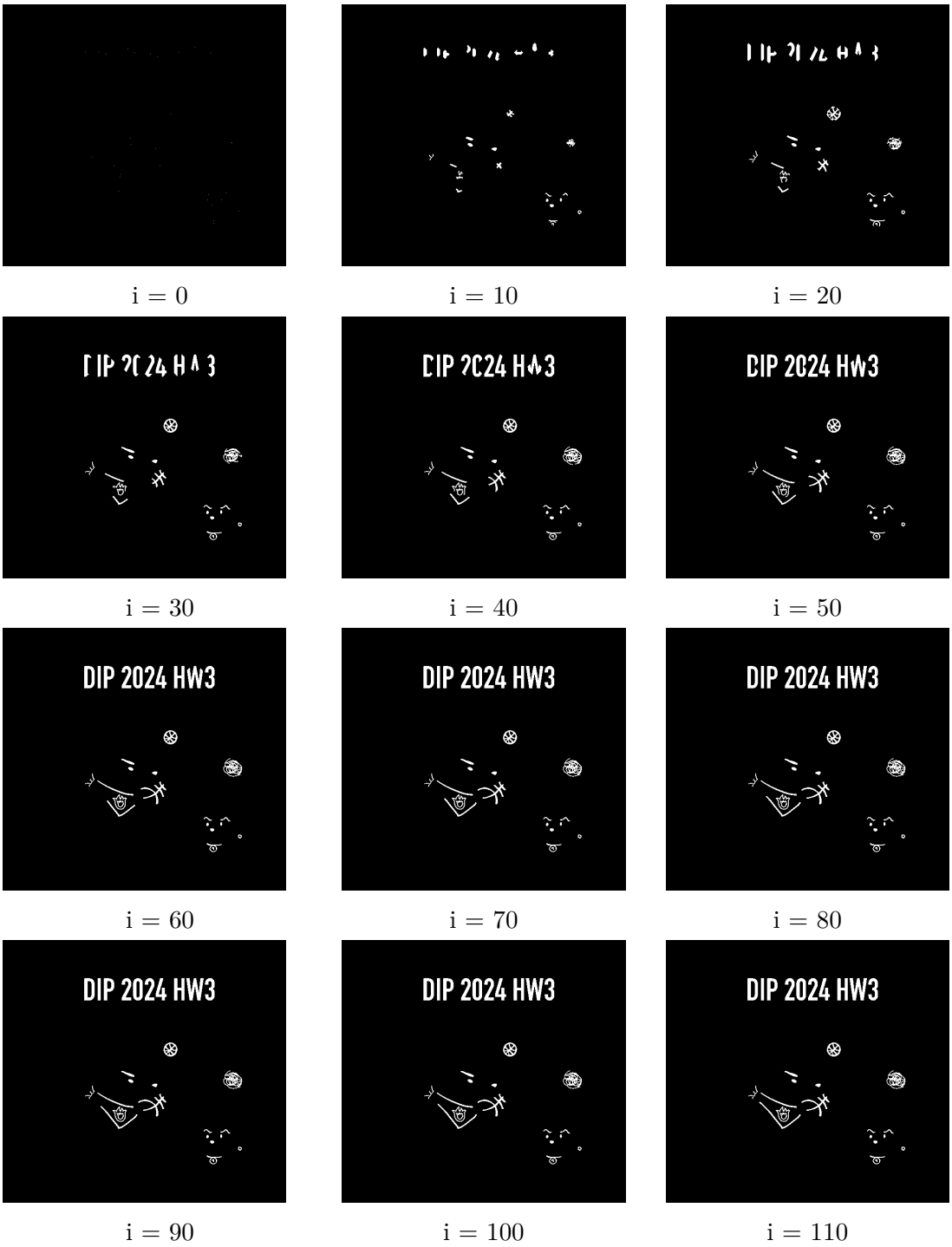


Figure 7: G_i in Hole Filling

最後，透過以下式子就能得到最終結果：

$$G(j,k) = G_i(j,k) \cup F(j,k)$$

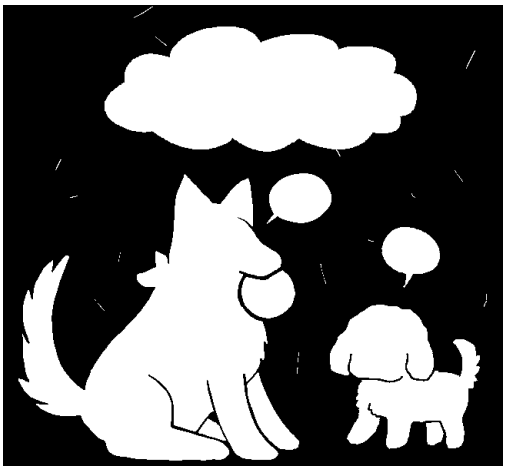


Figure 8: result2 (w/o Noise Removal)

同樣地，noise 的出現讓畫面變得不太美觀，因此我嘗試改做了一次 Subtractive Operator，其中我選用的 mask 為：

$$M = \begin{bmatrix} 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 0, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \end{bmatrix}, \quad origin : (2, 2)$$

並設立一個 threshold = 8，如果當前 pixel 是 255，則依照下面的規則，否則為 0：

$$F'_{i,j} = \begin{cases} 255, & \text{if } \sum_{mask} F_{i,j} * M_{i,j} \geq threshold \\ 0, & \text{otherwise} \end{cases}$$

Hole Filling 後再做 Noise Removal 的結果如下：

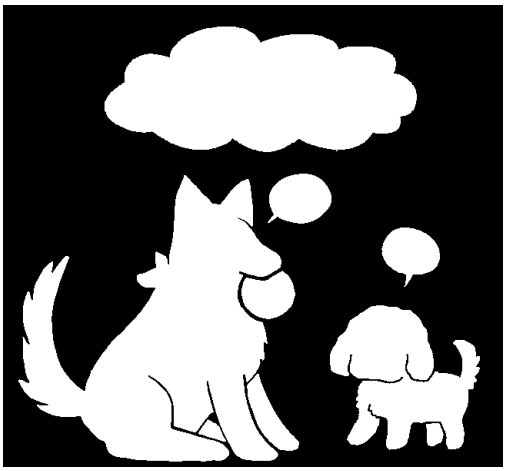
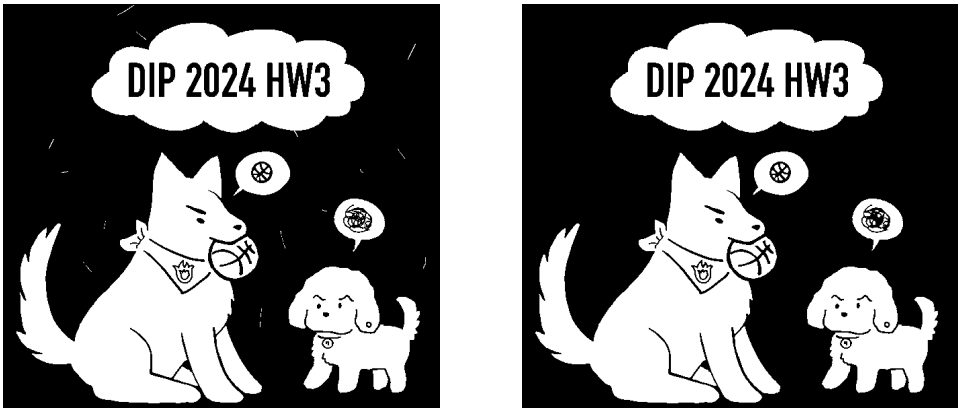


Figure 9: result1 (w/ Noise Removal)

(c) Design an algorithm to remove background noise of **sample1.png** and output the result as **result3.png**. Describe the steps in detail and specify the corresponding parameters. Please also perform noise removal on **sample1.png** as you did in **HW1**. Compare these two resultant images and provide discussions for these two approaches.

Result



sample1 result3

Figure 10: sample1 (Noise Removal)

Approach

我嘗試了數種方式，包含：

(1) Subtractive Operator
與前面所實做的 Subtractive Operator **相同，其中我選用兩種不同大小的** mask **為：**

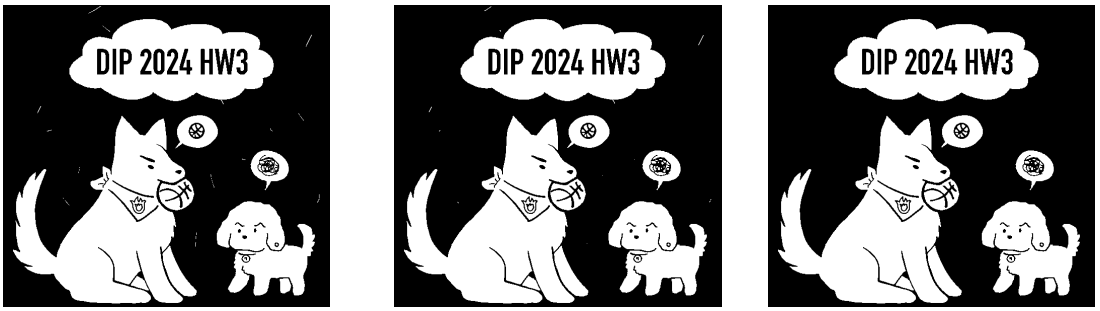
$$M_{3x3} = \begin{bmatrix} 1, & 1, & 1 \\ 1, & 0, & 1 \\ 1, & 1, & 1 \end{bmatrix}, \quad origin : (1,1), \quad threshold = 3$$

$$M_{5x5} = \begin{bmatrix} 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 0, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \\ 1, & 1, & 1, & 1, & 1 \end{bmatrix}, \quad origin : (2,2), \quad threshold = 8$$

如果當前 pixel **是** 1 (*i.e.* 255)**，則依照下面的規則，否則為** 0**：**

$$F'_{i,j} = \begin{cases} 255, & \text{if } \sum_{mask} F_{i,j} * M_{i,j} \geq threshold \\ 0, & \text{otherwise} \end{cases}$$

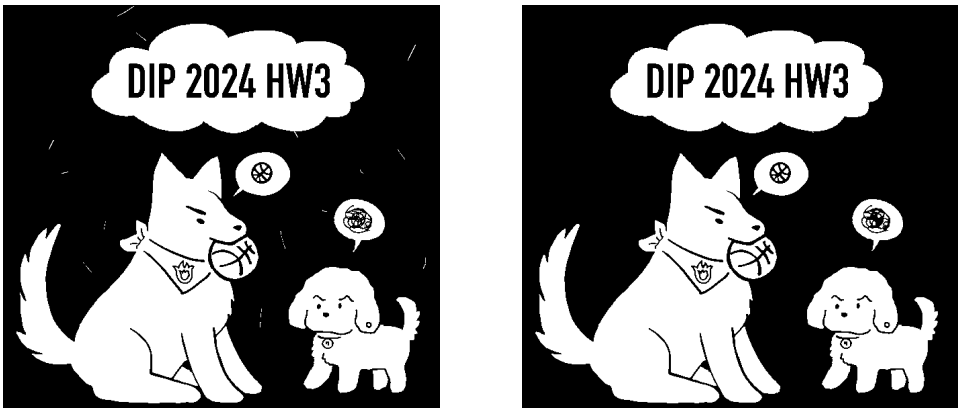
其中 M_{3x3} 的 threshold 選擇為 4，並且我連續做了 12 次。可以看到由於 mask 相較於圖片過小，因此只做一次的效果沒有很好：



sample1 M_{3x3} * 1 M_{3x3} * 12

Figure 11: sample1 (Subtractive Operator M_{3x3})

其中 M_{5x5} 的 threshold 選擇為 8，而且只做了 1 次：



sample1 M_{5x5} * 1

Figure 12: sample1 (Subtractive Operator M_{5x5})

可以看到背景變得乾淨了，但是某些細節喪失了，例如右邊狗狗的對話框的生氣圖案以及籃球的紋路與背景融合了。最後選用效果最好的 M_{5x5} 來當作 result。

(2) Opening

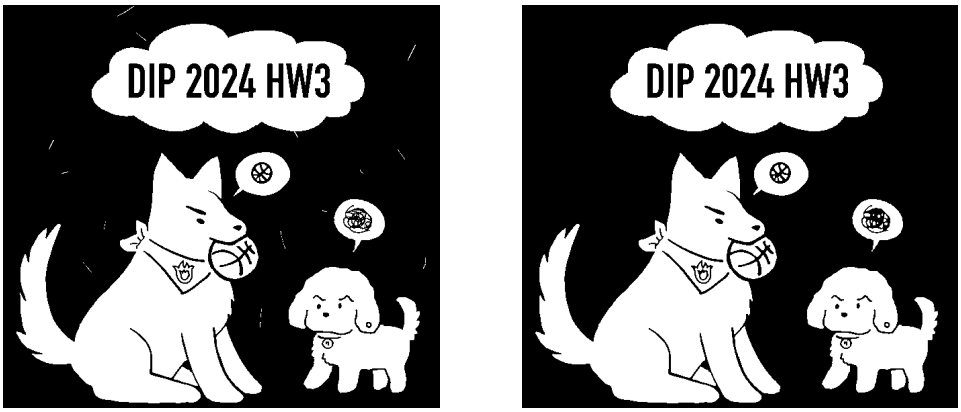
與前面實做的 opening 相同，其式子如下：

$$G(j,k) = [F(j,k) \ominus \tilde{H}(j,k)] \oplus H(j,k)$$

其中我使用另一個 Structuring Element 為：

$$H = \begin{bmatrix} 1, & 1, & 0 \\ 1, & 1, & 0 \\ 1, & 0, & 0 \end{bmatrix}, \quad origin : (1,1)$$

結果如下：



sample1 opening

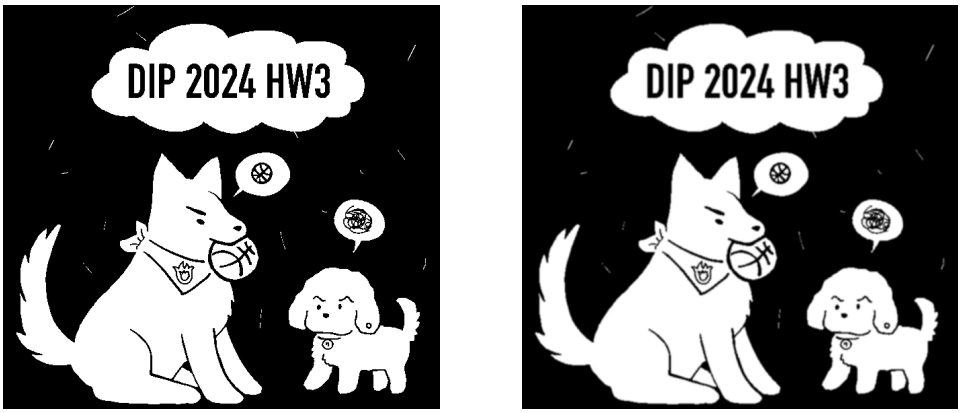
Figure 13: sample1 (Opening)

同樣地可以看到背景變得乾淨了，但是某些細節喪失了，例如右邊狗狗的對話框的生氣圖案以及籃球的紋路與背景融合了。

(3) Spacial Filtering
此為 HW1 的做法，其中我選用的 kernel 為：

$$K_{5x5} = \frac{1}{256}\begin{bmatrix} 1, & 4, & 6, & 4, & 1 \\ 4, & 16, & 24, & 16, & 4 \\ 6, & 24, & 36, & 24, & 6 \\ 4, & 16, & 24, & 16, & 4 \\ 1, & 4, & 6, & 4, & 1 \end{bmatrix}$$

結果如下：

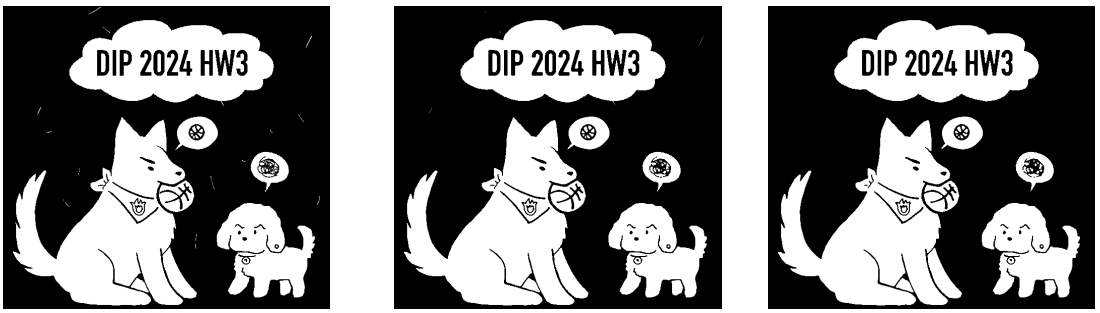


sample1 opening

Figure 14: sample1 (Spacial Filtering)

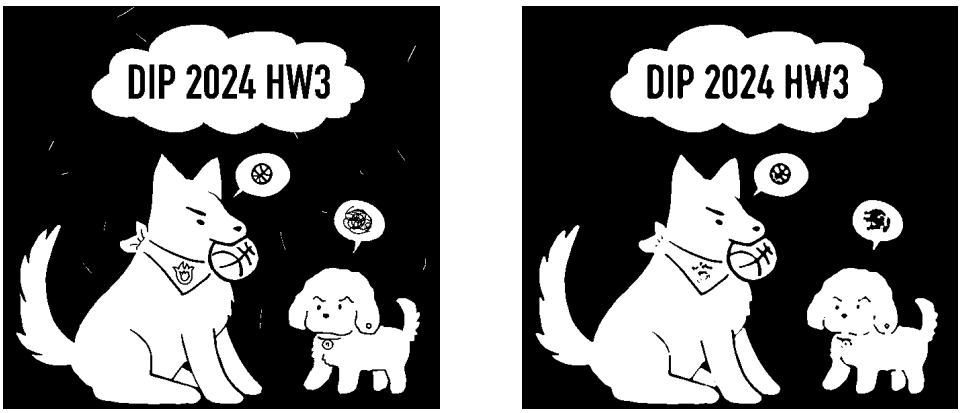
可以看到僅僅是糢糊了背景，對去除這裡的 Noise 沒有太大的幫助，因此可以推斷使用 Median Filtering 來處理會比較適合，因為這裡的 noise 的特性比較接近 impulse noise。

(4) Median Filtering
此為 HW1 的做法，其中我選用的 kernel 為 K_{3x3} 和 K_{5x5}：



sample1 K_{3x3} * 1 K_{3x3} * 2

Figure 15: sample1 (Median Filtering K_{3x3})



sample1 K_{5x5} * 1

Figure 16: sample1 (Median Filtering K_{5x5})

比較 Subtractive Operator, Opening, Median Filtering，我們可以看到他們對於清除背景 noise 都有不錯的效果，但對於一些線條較為密集的地方，都會被誤判為是 noise，導致細節失真，尤其是 Median Filtering：

sample1

Subtractive Operator M_{5x5} (result3)

Opening

Median Filtering

Figure 17: Comparison between different approach

(d) Design an algorithm to count the number of objects in **sample1.png**. Describe the steps in detail and specify the corresponding parameters.

Result

sample1

sample1 (Colored)

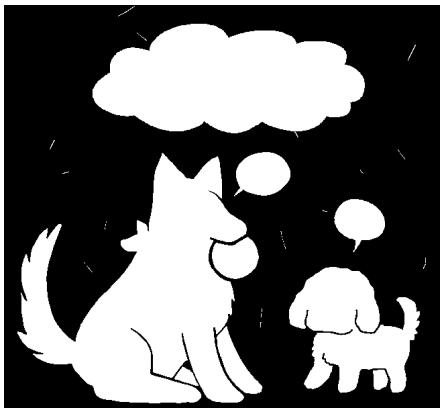
Figure 18: sample1 (Connected component labeling)

Approach

經過觀察可以發現有很多塊 object 被其紋路 (黑色線條) 所切開，因此如果直接對原圖去進行 connected component labeling 會出現同一個問體但是被分成很多份的狀況。因此，我先對原圖操作了 Hole filling，確保原圖同一物體盡可能的都被分到一起，再去進行 connected component labeling，最後再把物體的紋路 (黑色線條) 貼回去，過程如下：

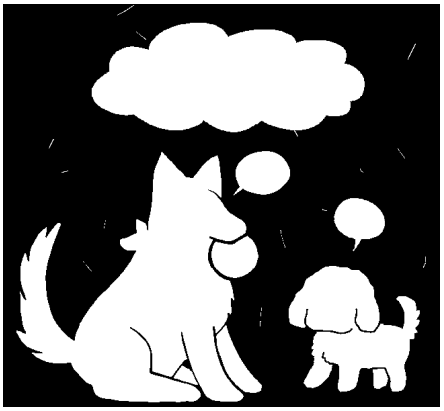
首先我們先對原圖進行 Hole Filling，也就是前面做的 result2：

sample1

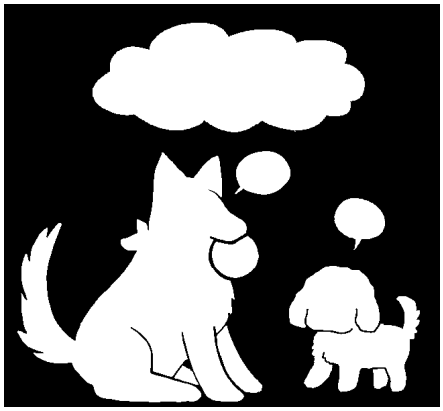


result2 (Hole Filling)

再來對 result2 進行 denoise，這邊選用前題討論的 opening：



result2 (Noise Removal)

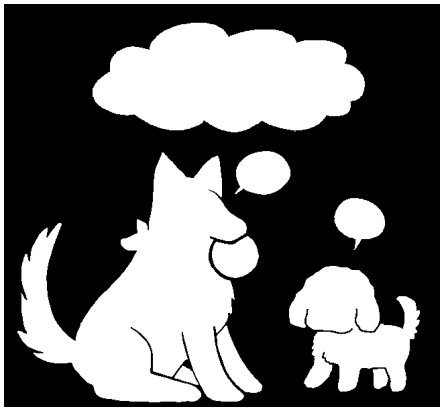


result2 (Opening

再來，遍歷這張圖，只要遇到 pixel value 是 255，並且先前還未訪問到的 pixel，就更新目前 component 的數量並當作此物體 id，並使用 bfs 將其周遭都 label 上這個新的 id，並對同樣 id 的 pixel 上相同的色：

```
466     h, w = img.shape
467     res = np.zeros_like(img)
468     for i in range(h):
469         for j in range(w):
470             if img[i, j] == 255 and res[i, j] == 0:
471                 cnt += 1
472                 bfs(img=img, res=res, coords=(i, j), cnt=cnt)
473     return res, cnt
```

Figure 21: Connected component labeling (BFS)



result2 (Opening)

result2 (Colored)

最後，再把物體的紋路 (黑色線條) 貼回去，我們去檢查 sample1 哪些地方是黑的，直接修改 result2 (Coloring) 呈黑色即可達成：

result2 (Colored)

sample1 (Colored)

Discussion

可以看到大部分都分類的很不錯了，只有兩隻狗的腳的部分，由於原本就與狗身體分開，因此能夠預期他沒辦法被正確的分類。

Problem 2 : TEXTURE ANALYSIS

(a) Please apply Law's method to **sample2.png** to extract the feature vectors. Describe your approach for extracting the feature vectors. You may include the feature images (normalize to 255) in your report for explanation, but it is optional.

Parameter

(a) **Mask**
我使用了 9 組 3x3 的 mask，由以下三組 vector 兩兩相乘而來，

$$L_3 = \begin{bmatrix} 1, & 2, & 1 \end{bmatrix}, \quad E_3 = \begin{bmatrix} -1, & 0, & 1 \end{bmatrix}, \quad S_3 = \begin{bmatrix} 1, & -2, & 1 \end{bmatrix}$$

(b) **Micro-structure impulse response array**
將 sample2 用以上 9 個 Mask 進行 convolution 可以得到 9 個 Micro-structure Array，如下 (經過 normalized)：

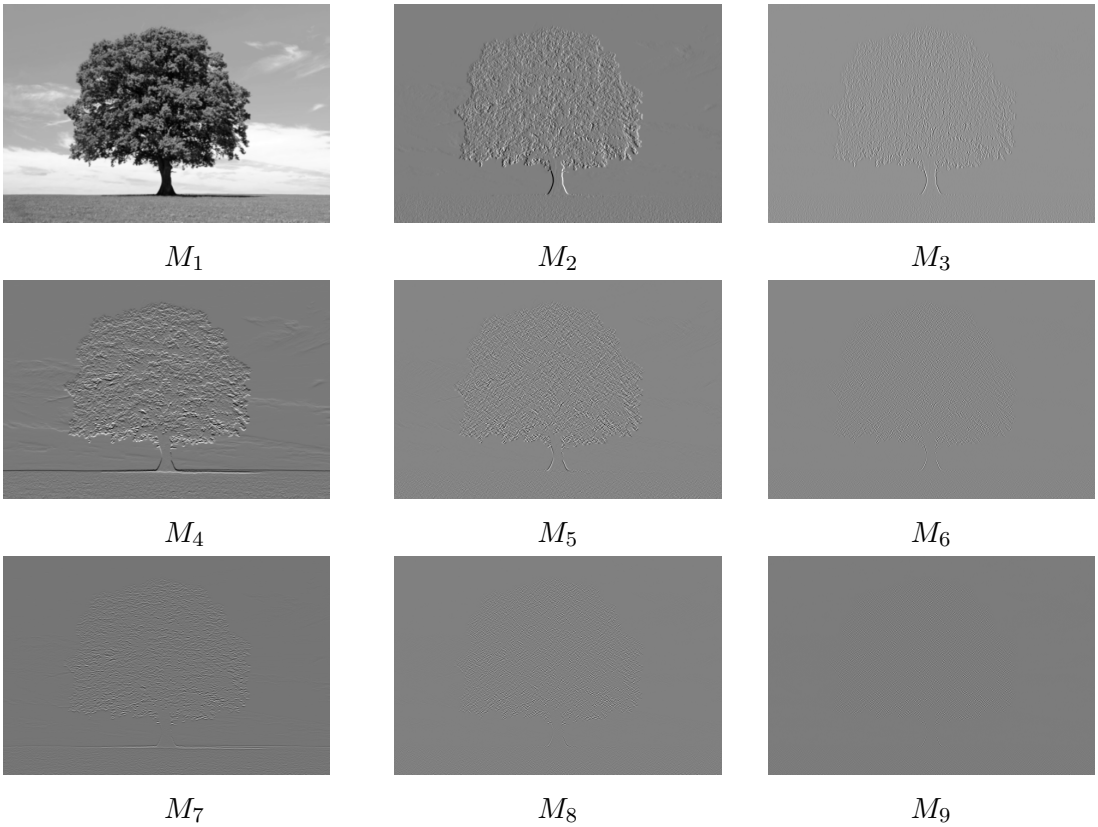


Figure 24: sample2 (Micro-structure Array)

(c) **Energy Computation**
我使用了 3 種 metric 來計算 energy：L1-norm, L2-norm, mean，其結果分別如下 (經過 normalized, Energy Mask Size = 23)：

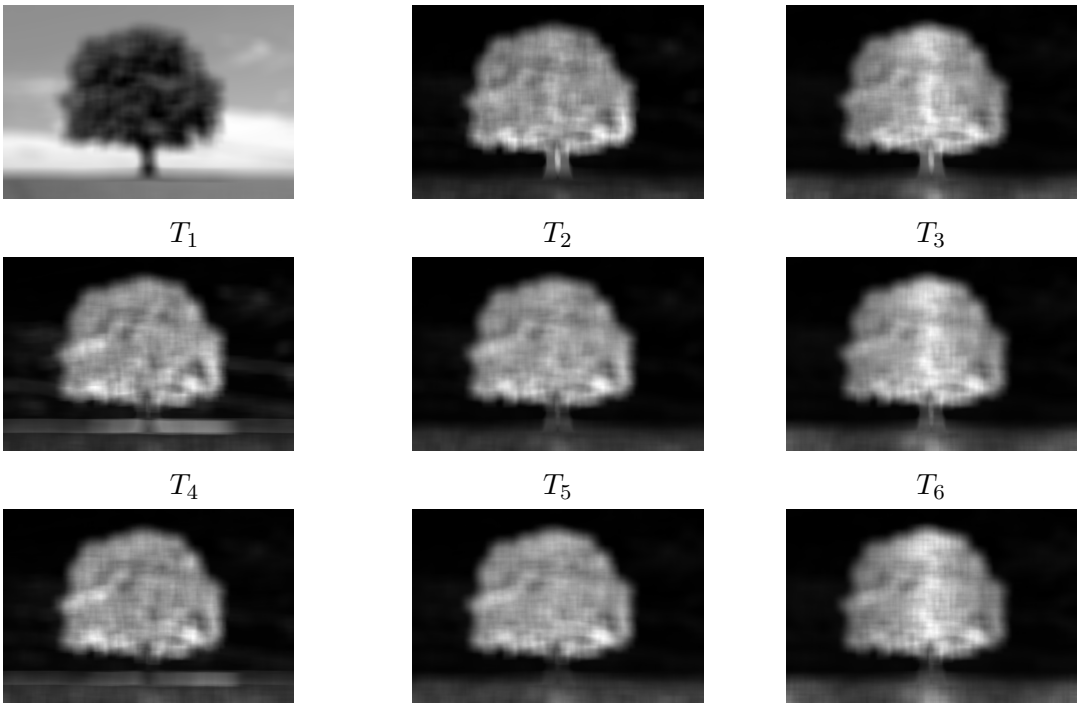


Figure 25: sample2 energy (L1-norm)

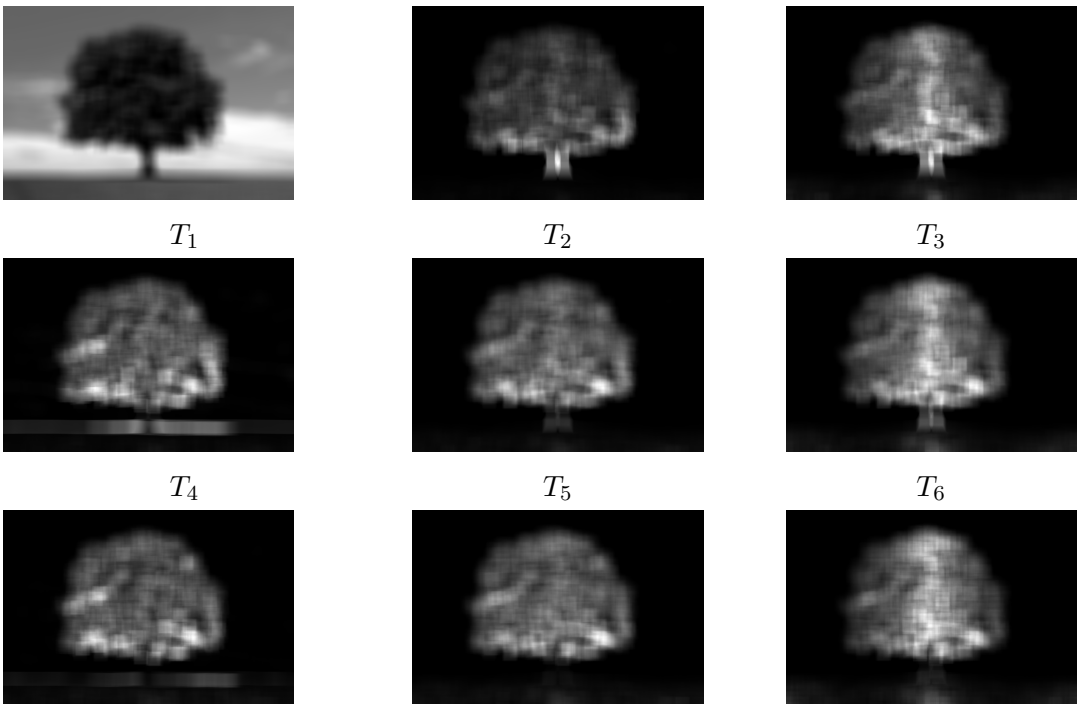


Figure 26: sample2 energy (L2-norm)

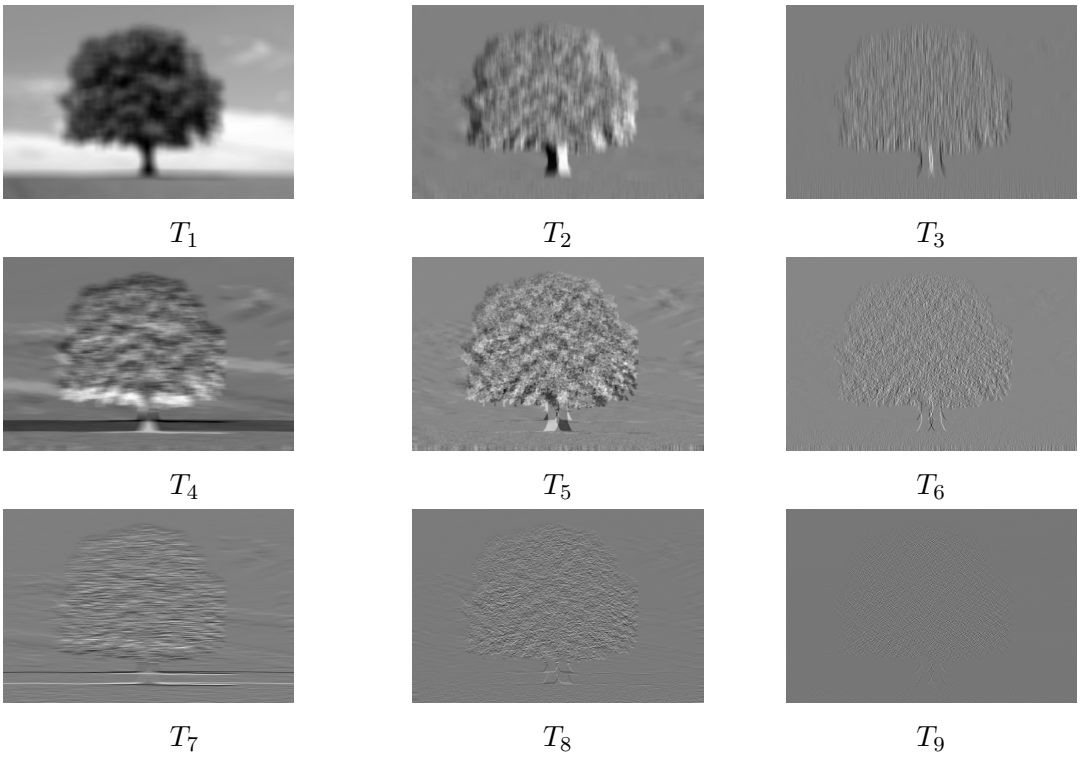
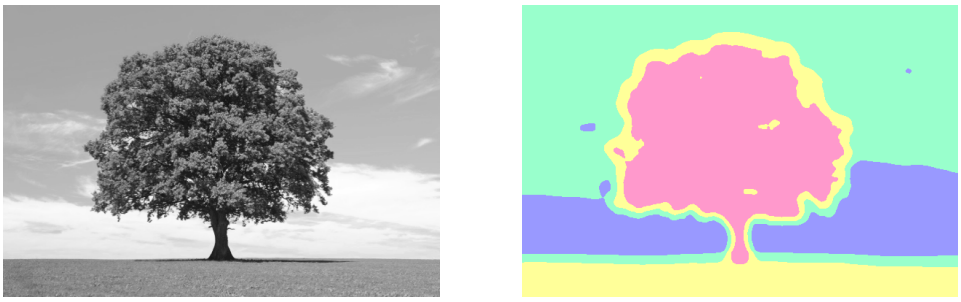


Figure 27: sample2 energy (Mean)

(b) Apply the k-means algorithm to classify each pixel using the feature vectors obtained in (a). Assign the same color to pixels with the same texture and save the result as **result4.png**. Describe in detail how you use the features in the k-means algorithm and all the chosen parameters.

Result



sample2 result4 (mask=23, mean, k=4)

Figure 28: Law's Method + k-means

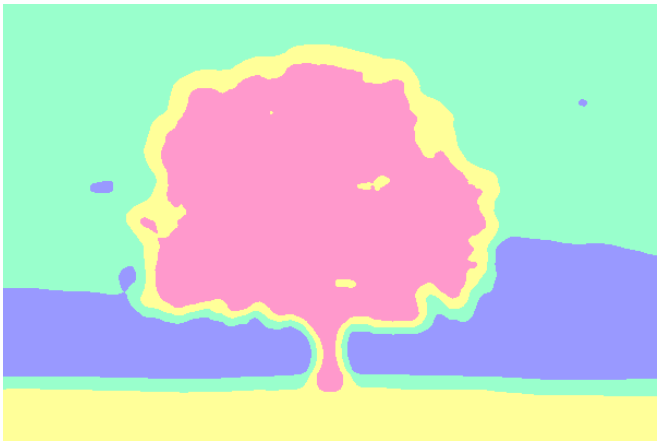
Approach

我總共調整了 3 種參數：

- Energy Mask size：[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
- Energy strategy：[L1-norm, L2-Norm, mean]
- K-means k：[3, 4, 5]

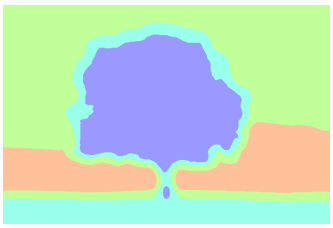
並由肉眼來評估各種組合的好壞。另外，我選用了全部的 feature(9 種) 來做 k-means, 因為視覺化後的 feature 是經過 normalized 的，我不確定是否可以取代掉，因此就決定全用。而 k 選用到 5 是認為或許樹葉與樹幹可以分開，但是結果好像不如預期，但是卻有令人沒料的結果。而 Micro-structure impulse response array 的 Mask 我只有用 3x3 的，以下列出我覺得不錯的結果：

- **Best**



mask=23, mean, k=4

- $k = 4$



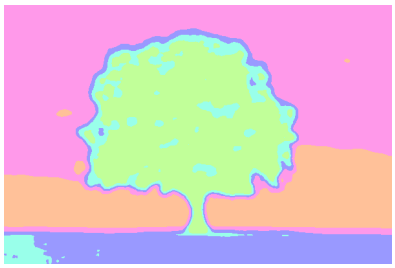
mask=29, L2-norm

mask=3, L2-norm

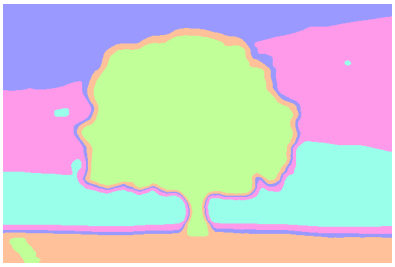
mask=7, L1-norm

- $k = 5$ 可以看到 $k = 5$ 時，樹、地面、天空分的效果也很不錯，只是大多問題是天空被分成三類，推測是雲的厚度變化導致的：

mask=19, mean



mask=15, L1-norm



mask=25, L1-norm

mask=29, L2-norm

- **藝術品等級**

mask=3, L1-norm, k=4

(c) Based on **result4.png**, add different texture patterns to each region and save it as **result5.png**. Describe the implementation details and discuss on the result.

Result

result4 result5

Figure 33: Texture Transfer

Approach

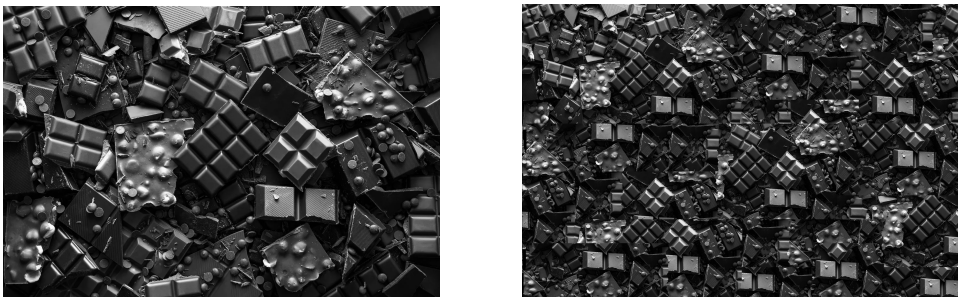
先準備 4 種材質，其 image size 大於等於 result4，接著選定 4 個材質在 result4 上任一一點座標當作起始點，將這幾個座標的 set id 與 4 個材質的 image 做 mapping，最後遍歷整張 result4，按照剛剛找到的 mapping 關係直接將該 pixel 相對應的材質複製到 result5 上即可。

```python
def texture_transfer(img: np.ndarray, cluster: np.ndarray, kmeans_k: int):
    assert kmeans_k >= 1, "k should be positive"

    wood = read_image("wood.jpg", 0, options=cv2.IMREAD_COLOR)
    raindrop = read_image("raindrop.jpg", 0, options=cv2.IMREAD_COLOR)
    rock = read_image("rock.jpg", 0, options=cv2.IMREAD_COLOR)
    wool = read_image("wool.jpg", 0, options=cv2.IMREAD_COLOR)
    rugs = read_image("rugs.jpg", 0, options=cv2.IMREAD_COLOR)

    h, w = img.shape
    res = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)

    textures = [raindrop, wood, wool, rock]
    labels = [cluster[50, 70], cluster[160, 260], cluster[320, 110], cluster[380, 100]]
    for i in range(h):
        for j in range(w):
            k = np.where(labels == cluster[i, j])[0][0]
            res[i, j] = textures[k][i, j]
    return res
```

Figure 34: Texture Transfer 實做

(d) TA is an avid chocolate enthusiast, and **sample3.png** features a variety of chocolates. Your task is to use **image quilting** on **sample3.png** to at least double the amount of chocolates, and save the result as **result6.png**. You will be rewarded with bonus points if the increased amount of chocolates brings great joy to TA. Don't forget to discuss your approach and result.

Result



sample3 result6

Figure 35: Image Quilting

Approach

參考*"Image Quilting for Texture Synthesis and Transfer"* https://dl.acm.org/doi/pdf/10.1145/383259.383296這篇論文，以及 https://github.com/axu2/image-quilting/tree/master的實作，實作步驟如下：

(1) 從 sample3 採樣出許多 patch image 來當作 quilting 的單位元素，稍後步驟從左上到右下依序填入適當的 patch image

(2) 為確保跟原圖類似，我將第一個 patch image 設為與原圖相同的 block，這樣縫合下去的就會與原圖有類似的趨勢

(3) 對於下一個要縫合的 patch image，檢查他的位置上方以及左方的 patch image(如果有的話)，決定一個 overlap 面積，從第一步驟採樣出來的單位元素選取一個 MSE 最小的 patch 出來

(4) 決定好步驟 3 的 patch image，我們就要找到其縫合上去 boundary 的最小 MSE 是要怎麼縫。我選擇實做 Dijkstra Shortest Path 來尋找 MSE boundary，再依照這個 boundary 貼上去即可

(5) 重複步驟 3, 4 直到整張照片完成

Parameter

- Patch Length: 200

- Overlap ratio: 20%
- Sampling Step: 40
  如果 sample 所有的 pixel，那會跑得過慢，因此我選擇每隔 40 pixel sample 一次

Other result

- Patch Length: 50, Overlap ratio: 20%

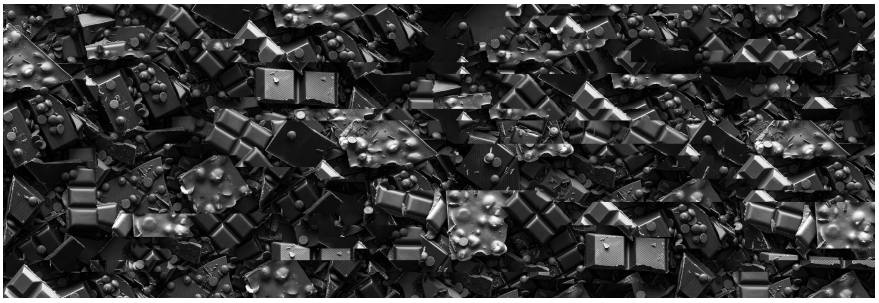


Figure 36

- Patch Length: 75, Overlap ratio: 20%

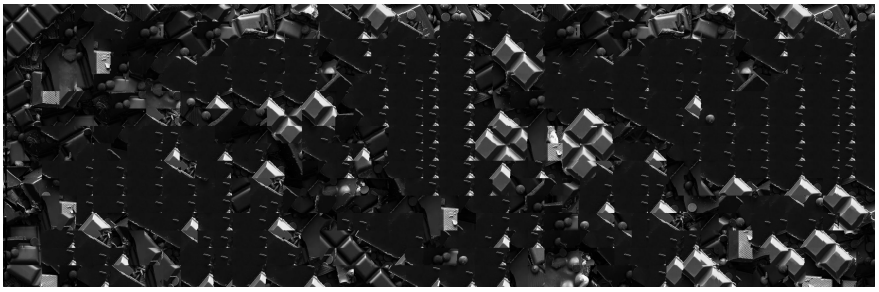


Figure 37

- Patch Length: 100, Overlap ratio: 20%

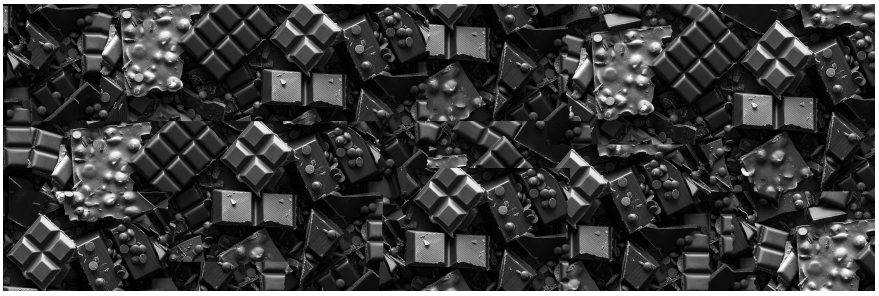


Figure 38

- Patch Length: 125, Overlap ratio: 20%

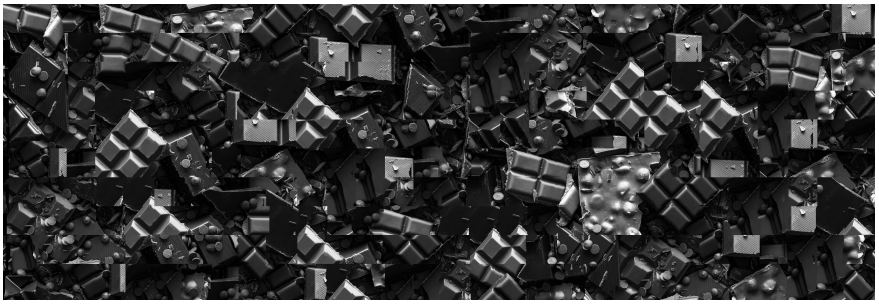


Figure 39

- Patch Length: 150, Overlap ratio: 20%

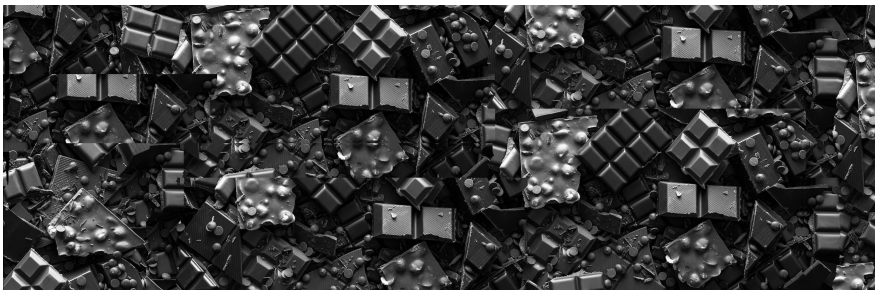


Figure 40