Computer Vision 8820

Alex Darwiche

UGA ID: 811878643

**HW2 – Skeletonization**

1. **High level pseudo code for skeletonization:**

def find\_distances(img):

create distance\_mask = img

while distance updates continue on img:

loop through rows in img:

loop through cols in img:

min\_val = Find the min of all 4-neighbors of current pixel

distance \_mask[pixel] = min\_val + 1

continue this until no more updates are possible

def find\_skeletons():

skeleton\_mask = distance\_mask

loop through rows in distance\_mask:

loop through cols in distance\_mask:

if pixel is locally max distance:

skeleton\_mask[pixel] = 1

# We need to remove this line for rebuilding, as we need to

# maintain the distance information

else:

skeleton\_mask[pixel] = 0

def rebuild\_binary():

new\_binary = skeleton\_mask

max = find max value in skeleton\_mask  
 for iteration in range(max):

loop through rows in img:

loop through cols in img:

if pixel > 0:

all neighbors = max(pixel\_value – 1 or previous\_value)

# Once rebuilt image, make all foreground = 1 and background = 0

loop through rows in img:

loop through cols in img:

if pixel > 0:

new\_binary[pixel] = 1

1. **Description of pseudocode.**
   1. The first function, find\_distances, loops through the image so long as it can still make updates. It will continue to find the min distance to a boundary for each pixel in the image, for each pixel within the current distance contour. So it would take 6 iterations for a pixel to be marked with a distance of 6.
   2. The second function, find\_skeletons, loops through the distances created in the first function. Now, this function finds all points that are locally maximum (distance from boundary) throughout the image. These points are given a value of 1 or they maintain their distance value. Anything that is not locally maximum, is set to 0. The remaining points are the “skeletons” of the image.
   3. The final function, rebuild\_binary, takes the skeletons (those with distance information preserved) and rebuilds the binary by proliferating out from the skeletons, based on their distance. The algorithm will always essentially find the max between the neighbors of the current pixel and the current pixel’s distance minus 1. This proliferates out until the original binary image is reconstructed.
2. **Original Binary Image (B)**

A graph showing a person holding a hammer and wrench

AI-generated content may be incorrect.

1. **Distance Masked Image**

A graph with different shapes

AI-generated content may be incorrect.

1. **Skeletons of Binary Image**

A graph with lines and numbers

AI-generated content may be incorrect.

1. **Reconstructed Binary Image**   
    A graph showing a person holding a hammer and wrench

   AI-generated content may be incorrect.
2. **Comments on results obtained**
   1. In this assignment I was able to find the distances, skeletons, and recreate an identical binary image to the first one provided from HW01. This makes sense as this style of compression is loss-less.
3. **Code Below this point**

import numpy as np

import matplotlib.pyplot as plt

# File path

file\_path = "img/comb.img"

# Image dimensions

width, height = 512, 512

header\_size = 512

def show\_image(img,cmap\_str='gray\_r'):

    norm = plt.Normalize(vmin=0, vmax=100)  # Normalize so that only positive values are highlighted

    plt.imshow(img, cmap=cmap\_str,norm=norm)

    plt.show()

def show\_image\_binary(img,cmap\_str='gray\_r'):

    norm = plt.Normalize(vmin=0, vmax=1)  # Normalize so that only positive values are highlighted

    plt.imshow(img, cmap=cmap\_str,norm=norm)

    plt.show()

# Read the file

with open(file\_path, "rb") as f:

    f.seek(header\_size)

    image\_data = np.frombuffer(f.read(), dtype=np.uint8).copy()

# Reshape into 2D array

image\_array = image\_data.reshape((height, width))

# Display the base comb image

# show\_image(image\_array)

# Question 1: Find the Binary Image

def find\_binary\_img(img):

    B\_t = img.copy()

    for ind1, row in enumerate(B\_t):

        for ind2, col in enumerate(row):

            B\_t[ind1][ind2] = 0 if col > 128 else 1

    return B\_t

b\_t = find\_binary\_img(image\_array)

# Display the binary image B\_T

# show\_image(b\_t)

# Question 2: Find Connected Components Iteratively

def iter\_connected\_comps(img, filter):

    equivalence\_table = dict()

    component\_data = dict()

    list\_equiv = []

    label = 0

    B\_image = np.zeros\_like(img, dtype=np.int16)

    # First loop through the image and label the components as their encountered, using a new label if

    # the top or left neighbor do not already have a label

    for ind1, row in enumerate(img):

        for ind2, col in enumerate(row):

            if col == 1:

                # if col1

                if ind2 == 0:

                    left\_neighbor = 0

                else:

                    left\_neighbor = B\_image[ind1][ind2-1]

                # if row1

                if ind1 == 0:

                    top\_neighbor = 0

                else:

                    top\_neighbor = B\_image[ind1-1][ind2]

                if top\_neighbor > 0 or left\_neighbor > 0:

                    min\_label = min(x for x in [top\_neighbor, left\_neighbor] if x > 0)

                    B\_image[ind1][ind2] = min\_label

                else:

                    B\_image[ind1][ind2] = label

                    label += 1

                # Check if the neighbors are already in the equivalence table

                if top\_neighbor > 0 and left\_neighbor > 0 and left\_neighbor != top\_neighbor:

                    list\_equiv.append([left\_neighbor,top\_neighbor])

    # delete duplicates in equivalence list

    seen\_unique = list()

    seen = set()

    for i in list\_equiv:

        pair = frozenset(i)

        if pair not in seen:

            seen\_unique.append(i)

            seen.add(pair)

    list\_equiv = seen\_unique

    # Loop through the equivalence list (labels that are equal) and build

    # an equivalence table. This should consolidate so that all components

    # within a singular object are equal to one another.

    for i in list\_equiv:

        in\_items0 = len([key for key, value in equivalence\_table.items() if i[0] in value])>0

        in\_items1 = len([key for key, value in equivalence\_table.items() if i[1] in value])>0

        # Combine lists

        if i[0] in equivalence\_table and i[1] in equivalence\_table:

            if i[1] != i[0]:

                equivalence\_table[i[0]] = list(set(equivalence\_table[i[0]]) | set(equivalence\_table[i[1]])).append(i[1])

                del equivalence\_table[i[1]]

        elif in\_items0 and in\_items1:

            key1 = [key for key, value in equivalence\_table.items() if i[0] in value][0]

            key2 = [key for key, value in equivalence\_table.items() if i[1] in value][0]

            if key1 != key2:

                equivalence\_table[key1] = list(set(equivalence\_table[key1]) | set(equivalence\_table[key2]))

                del equivalence\_table[key2]

        # Add if one list exists

        elif i[0] in equivalence\_table and i[1] not in equivalence\_table[i[0]]:

            if in\_items1:

                val = [key for key, value in equivalence\_table.items() if i[1] in value][0]

                if val != i[0]:

                    equivalence\_table[i[0]] = list(set(equivalence\_table[i[0]]) | set(equivalence\_table[val]))

                    del equivalence\_table[val]

            else:

                equivalence\_table[i[0]].append(i[1])

        # Add if other list exists

        elif i[1] in equivalence\_table and i[0] not in equivalence\_table[i[1]]:

            if in\_items0:

                val = [key for key, value in equivalence\_table.items() if i[0] in value][0]

                equivalence\_table[i[1]] = list(set(equivalence\_table[i[1]]) | set(equivalence\_table[val]))

                if val != i[1]:

                    del equivalence\_table[val]

            else:

                equivalence\_table[i[1]].append(i[0])

        elif in\_items0:

            key1 = [key for key, value in equivalence\_table.items() if i[0] in value][0]

            equivalence\_table[key1].append(i[1])

        elif in\_items1:

            key1 = [key for key, value in equivalence\_table.items() if i[1] in value][0]

            equivalence\_table[key1].append(i[0])

        # if none exist

        else:

            equivalence\_table[i[0]] = [i[1]]

    for ind1, row in enumerate(B\_image):

        for ind2, col in enumerate(row):

            if col > 0:

                if col not in equivalence\_table:

                    new\_label = [key for key, value in equivalence\_table.items() if col in value]

                    if len(new\_label) > 0:

                        B\_image[ind1][ind2] = new\_label[0]

    for ind1, row in enumerate(B\_image):

        for ind2, col in enumerate(row):

            if col > 0:

                if col not in component\_data:

                    component\_data[col] = {'size': 1, 'coords': [[ind2,ind1]]}

                else:

                    component\_data[col]['size'] += 1

                    component\_data[col]['coords'].append([ind2,ind1])

    filtered\_comps = [x for x in component\_data.keys() if component\_data[x]['size'] > filter]

    keys = list(component\_data.keys())

    for i in keys:

        if i not in filtered\_comps:

            for ind1, row in enumerate(img):

                for ind2, col in enumerate(row):

                    if B\_image[ind1][ind2] == i:

                        B\_image[ind1][ind2] = 0

            del component\_data[i]

    return B\_image, equivalence\_table, component\_data

def print\_comps(comp\_data):

    component\_num = 0

    for component in comp\_data:

        component\_num += 1

        area = comp\_data[component]['size']

        centroid = comp\_data[component]['centroid']

        bounding\_box = comp\_data[component]['bounding\_box']

        axis\_of\_elongation = comp\_data[component]['axis\_of\_elongation']

        eccentricity = comp\_data[component]['eccentricity']

        perimeter = comp\_data[component]['perimeter']

        compactness = comp\_data[component]['compactness']

        print(f"Component #{component\_num}:")

        print(f"{'='\*30}")

        print(f"Area: {area}")

        print(f"Centroid: {centroid}")

        print(f"Bounding Box: {bounding\_box}")

        print(f"Axis of Elongation: {axis\_of\_elongation}°")

        print(f"Eccentricity: {eccentricity:.2f}")

        print(f"Perimeter: {perimeter}")

        print(f"Compactness: {compactness:.2f}")

b\_image, eql\_table, cd = iter\_connected\_comps(b\_t,1000)

import copy

def find\_distances(img):

    '''

    This function loops through the image so long as it can still make updates.

    It will continue to find the min distance to a boundary for each pixel in

    the image, for each pixel within the current distance contour. So it would

    take 6 iterations for a pixel to be marked with a distance of 6.

    '''

    skeleton\_mask = copy.deepcopy((img))

    max\_val = 0

    while max(np.unique(skeleton\_mask))+1 > max\_val:

        for ind1, row in enumerate(img):

            for ind2, col in enumerate(row):

                if col > 0:

                    if ind2 == 0:

                        left\_neighbor = 0

                    else:

                        left\_neighbor = skeleton\_mask[ind1][ind2-1]

                    # if row1

                    if ind1 == 0:

                        top\_neighbor = 0

                    else:

                        top\_neighbor = skeleton\_mask[ind1-1][ind2]

                    # if row1

                    if ind2 == 511:

                        right\_neighbor = 0

                    else:

                        right\_neighbor = skeleton\_mask[ind1][ind2+1]

                    # if row1

                    if ind1 == 511:

                        bot\_neighbor = 0

                    else:

                        bot\_neighbor = skeleton\_mask[ind1+1][ind2]

                    if min(bot\_neighbor,top\_neighbor,left\_neighbor,right\_neighbor) == max\_val:

                        skeleton\_mask[ind1][ind2] = min(bot\_neighbor,top\_neighbor,left\_neighbor,right\_neighbor) + 1

        max\_val+=1

    return skeleton\_mask

def find\_skeletons(mask):

    '''

    This functionloops through the distances created in the first function. Now, this

    function finds all points that are locally maximum (distance from boundary)

    throughout the image. These points are given a value of 1 or they maintain their

    distance value. Anything that is not locally maximum, is set to 0. The remaining

    points are the “skeletons” of the image.

    '''

    mask2 = copy.deepcopy(mask)

    for ind1, row in enumerate(mask):

        for ind2, col in enumerate(row):

            if col > 0:

                if ind2 == 0:

                    left\_neighbor = 0

                else:

                    left\_neighbor = mask[ind1][ind2-1]

                # if row1

                if ind1 == 0:

                    top\_neighbor = 0

                else:

                    top\_neighbor = mask[ind1-1][ind2]

                # if row1

                if ind2 == 511:

                    right\_neighbor = 0

                else:

                    right\_neighbor = mask[ind1][ind2+1]

                # if row1

                if ind1 == 511:

                    bot\_neighbor = 0

                else:

                    bot\_neighbor = mask[ind1+1][ind2]

                neighbors = [top\_neighbor,bot\_neighbor,left\_neighbor,right\_neighbor]

                ge\_neighbors = [neighbor for neighbor in neighbors if neighbor>col]

                if len(ge\_neighbors) > 0:

                    mask2[ind1][ind2] = 0

    return mask2

def rebuild\_binary(skeleton\_mask):

    '''

    This function takes the skeletons (those with distance information preserved) and

    rebuilds the binary by proliferating out from the skeletons, based on their

    distance. The algorithm will always essentially find the max between the neighbors

    of the current pixel and the current pixel’s distance minus 1. This proliferates out

    until the original binary image is reconstructed.

    '''

    new\_mask = copy.deepcopy(skeleton\_mask)

    for iterations in range(np.max(new\_mask)):

        for ind1, row in enumerate(new\_mask):

            for ind2, col in enumerate(row):

                if col > 0:

                    if ind2 == 0:

                        continue

                    else:

                        new\_mask[ind1][ind2-1] = max(col-1, new\_mask[ind1][ind2-1])

                    # if row1

                    if ind1 == 0:

                        continue

                    else:

                        new\_mask[ind1-1][ind2] = max(col-1, new\_mask[ind1-1][ind2])

                    # if row1

                    if ind2 == 511:

                        continue

                    else:

                        new\_mask[ind1][ind2+1] = max(col-1, new\_mask[ind1][ind2+1])

                    # if row1

                    if ind1 == 511:

                        continue

                    else:

                        new\_mask[ind1+1][ind2] = max(col-1, new\_mask[ind1+1][ind2])

    for ind1, row in enumerate(new\_mask):

            for ind2, col in enumerate(row):

                if col > 0:

                    new\_mask[ind1][ind2] = 1

    return new\_mask

distance\_mask = find\_distances(b\_t)

skeletons = find\_skeletons(distance\_mask)

new\_binary = rebuild\_binary(skeletons)

# Visualize all images

show\_image\_binary(b\_t)

show\_image(distance\_mask)

show\_image(skeletons)

show\_image\_binary(new\_binary)