Computer Vision 8820

Alex Darwiche

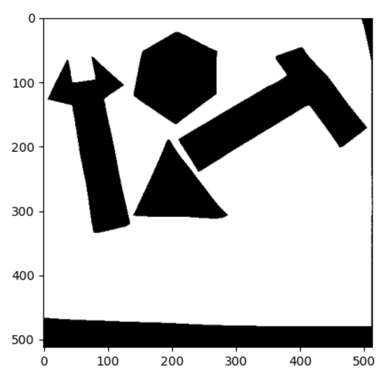
UGA ID: 811878643

HW2 – Skeletonization

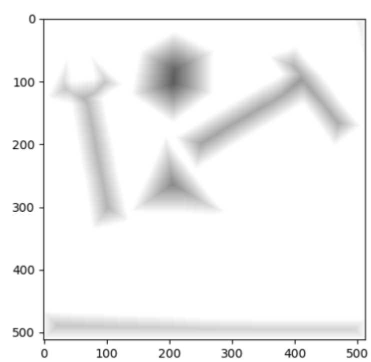
1. **High level pseudo code for skeletonization:**

```
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
```

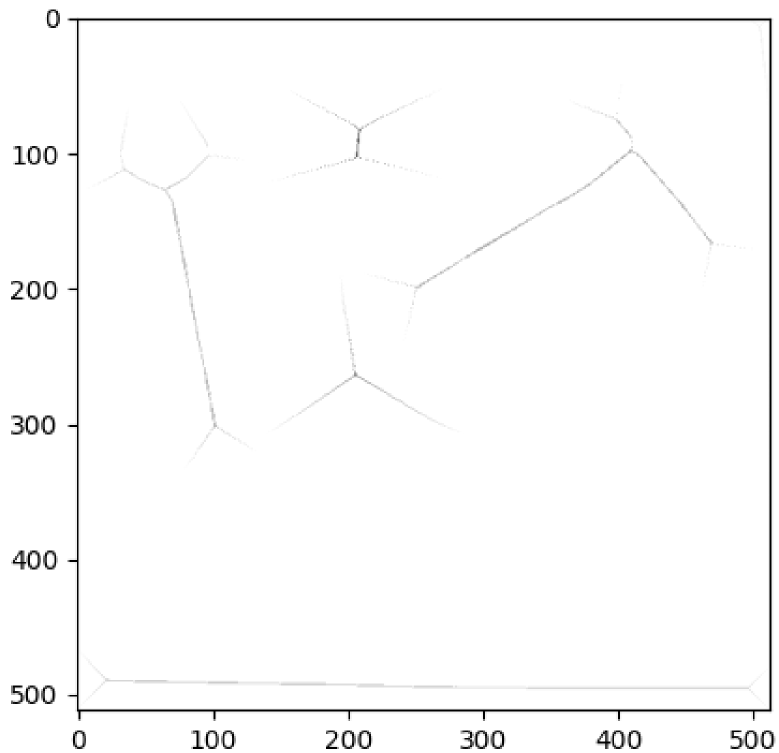
2. **Description of pseudocode.**
    a. The first function, find_distances, loops through the image so long as it can still make updates. It will continue to find the min distance to a boundary for each pixel in the image, for each pixel within the current distance contour. So it would take 6 iterations for a pixel to be marked with a distance of 6.
    b. 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.
    c. 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.
3. **Original Binary Image (B)**



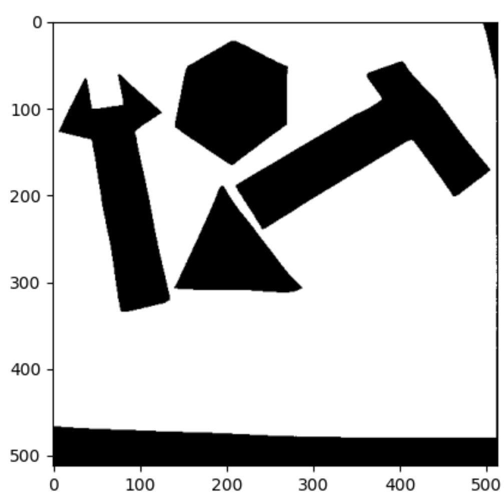
4. **Distance Masked Image**



5. **Skeletons of Binary Image**



6. **Reconstructed Binary Image**



7. **Comments on results obtained**
    a. 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.

8. 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
```

```python
    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)
```