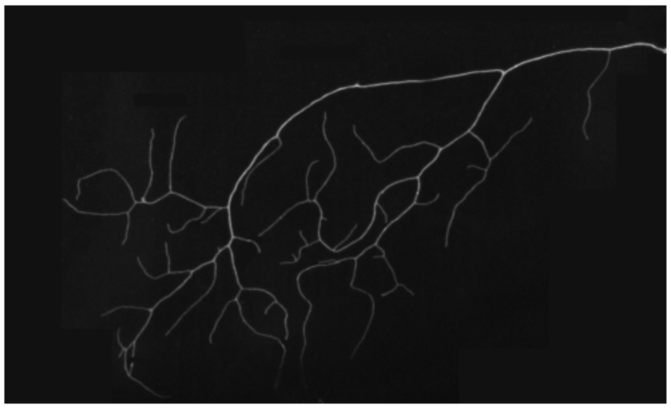
```
import tifffile
import numpy as np
import matplotlib.pyplot as plt
import math
import cv2


file_path = 'plasma_membrane_dendrite_cropped.tif'


image = tifffile.imread(file_path)


# Function to display the image
def display_image(image, p=False):

    # Print the image shape
    if p:
        print(f"Image shape: {image.shape}")
        print(image)

    # Ensure the image tensor is in the range [0, 1]
    # image_normalized = image / 255.0

    # Display the image
    plt.imshow(image, cmap='gray')
    plt.axis('off')  # Turn off axis numbers
    plt.show()


display_image(image, True)
```

```
Image shape: (849, 1411)
[[139 139 139 ... 143 133 144]
 [139 139 139 ... 147 144 141]
 [139 139 139 ... 137 144 141]
 ...
 [139 139 139 ... 139 139 139]
 [139 139 139 ... 139 139 139]
 [139 139 139 ... 139 139 139]]
```



```
print(image.dtype)    # <----- IMPORTANT! 16-bit image
```

```
uint16
```

Assumptions and Notes

- Because this images shows neuronal dendrites, I assume this a tree structure with no cycles. Therefore, I can use a depth first search approach.
- We also assume we can start with the 'Cell Body', which is the root of the tree structure.

```
# DFS Algorithm Code

class TreeNode:
    def __init__(self, id):
        self.id = id
        self.children = []

    # for creating synthetic data
    def add_child(self, child_node):
        self.children.append(child_node)

def countLeaves(node):
    # Base case: if the node is None, return 0
    if not node:
        return 0

    # Base case: if the node is a leaf (no children), return 1
    if not node.children:
        return 1

    # Recursive case: count leaves in all subtrees of the current node
    leaf_count = 0
    for child in node.children:
        leaf_count += countLeaves(child)

    # Print the current node and its leaf count
    print(f"Node {node.id} has {leaf_count} leaves")

    return leaf_count



# Generate synthetic data
root = TreeNode("root")
child1 = TreeNode("child1")
child2 = TreeNode("child2")
leaf1 = TreeNode("leaf1")
leaf2 = TreeNode("leaf2")
leaf3 = TreeNode("leaf3")

root.add_child(child1)
root.add_child(child2)
child1.add_child(leaf1)
child1.add_child(leaf2)
child2.add_child(leaf3)

total_leaves = countLeaves(root)
```

```
Node child1 has 2 leaves
Node child2 has 1 leaves
Node root has 3 leaves
```

Now the question becomes how should we get the 'relationship' data among those joint points in the image.

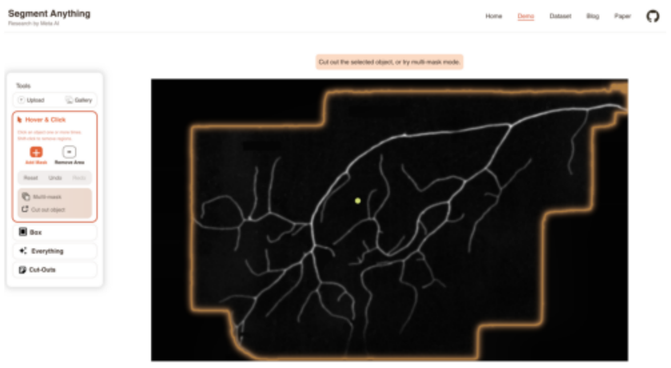
Step 1 - Segment the image

a few approaches mind

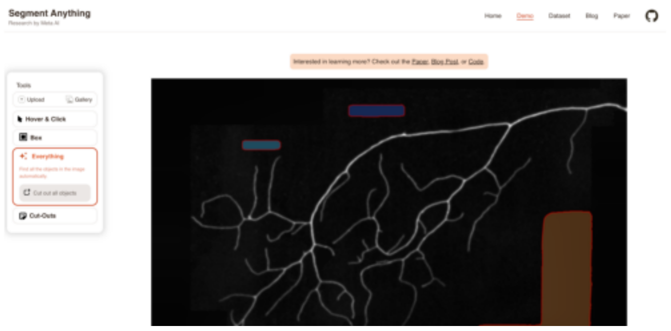
- Deep learning based segmentation (pret-trained vision model or cnn) - pixel level accuracy is important.
- Manual approach - enhance the image, get edges, find joint points and leaf nodes and create a graph - pixel level accuracy is ALSO IMPORTANT

```
# Try a pre-trained vision model - SAM
```

```
# Segment Anything Model seems not to work - it can't detect the "edges"
# However, some finetuning using our private data could result in better performance.

i1 = cv2.imread('i1.png')
i2 = cv2.imread('i2.png')

display_image(i1)
display_image(i2)
```



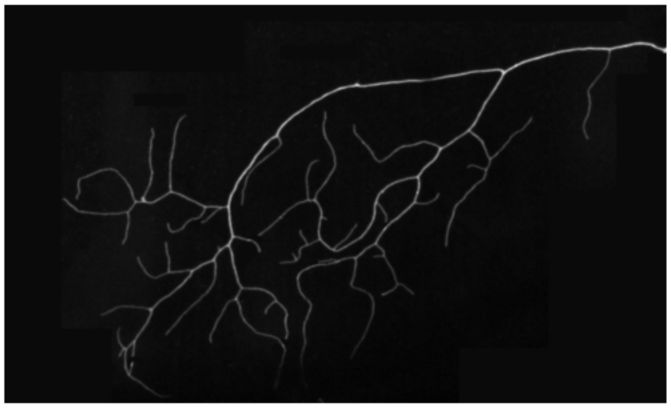
Cut out the selected object, or try multi-mask mode.



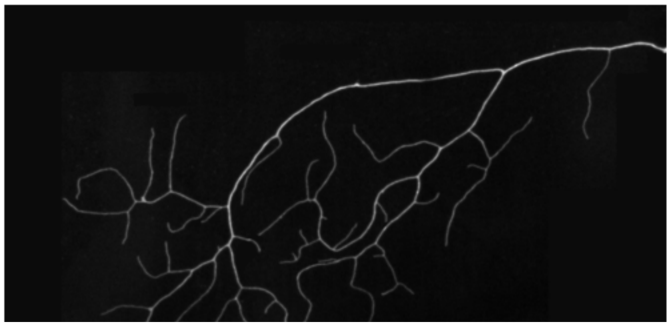
Interested in learning more? Check out the Paper, Blog Post, or Code.

```
# IMPLEMENTATION OF MANUAL APPROACH

# for noise reduction
smoothed_image = cv2.GaussianBlur(image, (5, 5), 0)
display_image(smoothed_image, True)

# # Enhance contrast using CLAHE, histogram equalization technique
# # I set clipLimit to 2.0 to limit the amplification effects
# clahe = cv2.createCLAHE(clipLimit=2.0)
# contrasted_image = clahe.apply(smoothed_image)

# display_image(contrasted_image, True)

# Normalize the image.
# This step converts a 16-bit image to a 8-bit image    <----- This could lead to loss of information
normalized_image = cv2.normalize(smoothed_image, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8UC1)
display_image(normalized_image, True)

THRESHOLD_VALUE = 30 # <----- This is based on experiential expereiments results

_, segmented_image = cv2.threshold(normalized_image, THRESHOLD_VALUE, 255, cv2.THRESH_BINARY)
display_image(segmented_image, True)
```

```
Image shape: (849, 1411)
[[139 139 139 ... 143 142 141]
 [139 139 139 ... 142 142 142]
 [139 139 139 ... 141 142 143]
 ...
 [139 139 139 ... 139 139 139]
 [139 139 139 ... 139 139 139]
 [139 139 139 ... 139 139 139]]
```



```
Image shape: (849, 1411)
[[11 11 11 ... 14 13 12]
 [11 11 11 ... 13 13 13]
 [11 11 11 ... 12 13 14]
 ...
 [11 11 11 ... 11 11 11]
 [11 11 11 ... 11 11 11]
 [11 11 11 ... 11 11 11]]
```



Step 2 - Get the skeleton

```
binary_segmented_image = segmented_image > 0
```

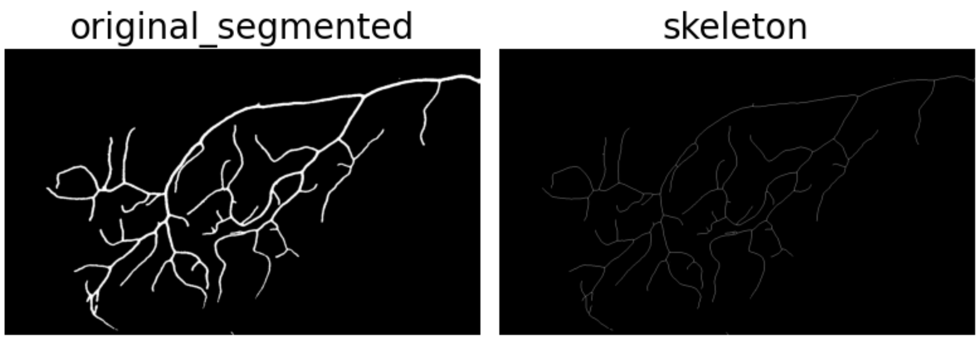
```
 [0 0 0 ... 0 0 0]
```

```
binary_segmented_image.max()
```

```
True
```

```
 [0 0 0 ... 0 0 0]]
```

```
# Get the skeleton using skimage package
# code source: https://scikit-image.org/docs/stable/auto_examples/edges/plot_skeleton.html

# we turned it into a 1 pixel wide skeleton

from skimage.morphology import skeletonize
from skimage import data
import matplotlib.pyplot as plt


# perform skeletonization
skeleton = skeletonize(binary_segmented_image)

# display results
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 4),
                         sharex=True, sharey=True)

ax = axes.ravel()

ax[0].imshow(binary_segmented_image, cmap=plt.cm.gray)
ax[0].axis('off')
ax[0].set_title('original_segmented', fontsize=20)

ax[1].imshow(skeleton, cmap=plt.cm.gray)
ax[1].axis('off')
ax[1].set_title('skeleton', fontsize=20)

fig.tight_layout()
plt.show()
```



```
display_image(skeleton, True)
```

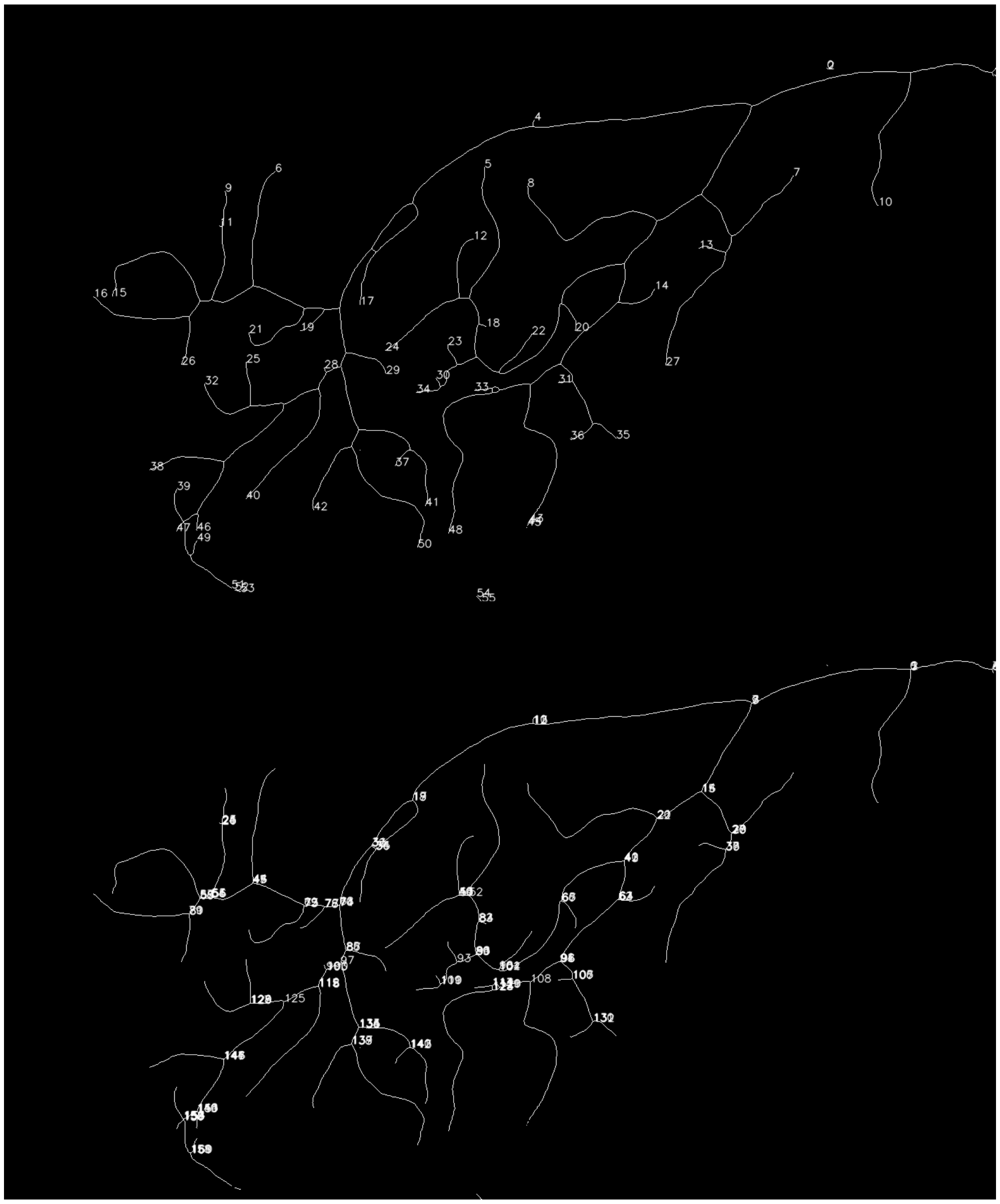
```
Image shape: (849, 1411)
[[False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]]
```

Note that: the joints are pixels surrounded by three or four neighbors, while the edges pixels have only two. Free endpoints have only one, which are leaf nodes in our case.

```
def find_neighbors(loc, skeleton): # loc is a tuple here
  neighbors = []
  # check all sournding pixels and skip itself
  for i in range(-1, 2):
    for j in range(-1, 2):
      if i == 0 and j == 0:
        continue
      r, c = loc[0] + i, loc[1] + j
      # make sure they are inside the skeleton
      if 0 <= r < skeleton.shape[0] and 0 <= c < skeleton.shape[1]:
        if skeleton[r, c]:
          neighbors.append((r, c))
  return neighbors



def is_junction(loc, skeleton):
  return len(find_neighbors(loc, skeleton)) > 2

def is_leaf(loc, skeleton):
  return len(find_neighbors(loc, skeleton)) == 1


def traverse_nodes(skeleton):
  junctions = []
  leaf_nodes = []
  for r in range(skeleton.shape[0]):
    for c in range(skeleton.shape[1]):
      if skeleton[r, c]:
        if is_junction((r, c), skeleton):
          junctions.append((r, c))
        elif is_leaf((r, c), skeleton):
          leaf_nodes.append((r, c))
  return junctions, leaf_nodes


junctions, leaf_nodes = traverse_nodes(skeleton)
print(len(junctions))
```

```
    162
```

```
from google.colab.patches import cv2_imshow

def mark_nodes_on_skeleton(skeleton, nodes): # could be either leaf nodes or junctions

    # Ensure skeleton is in the right format (255 for lines, 0 for background)
    marked_img = (skeleton * 255).astype('uint8')

    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 0.5
    font_color = 255  # White color for binary image
    font_thickness = 1

    for i, point in enumerate(nodes):
        cv2.putText(marked_img, str(i), (point[1], point[0]), font, font_scale, font_color, font_thickness, cv2.LINE_AA)

    # Display the image
    cv2_imshow(marked_img)

    return marked_img


# Showing marked leaf nodes here is only for inspection purposes

marked_leaf_nodes_image = mark_nodes_on_skeleton(skeleton, leaf_nodes)
marked_junctions_image = mark_nodes_on_skeleton(skeleton, junctions)
```



There are still some errors in marking leaf nodes, and it doesn't achieve 100% accuracy, at least based on inspection using human eyes. The junction nodes detection seem to have a worse performance, and there are so many duplicates as showed in the graph. However, it's a positive sign that we can create a graph (a simple tree here) given the results.

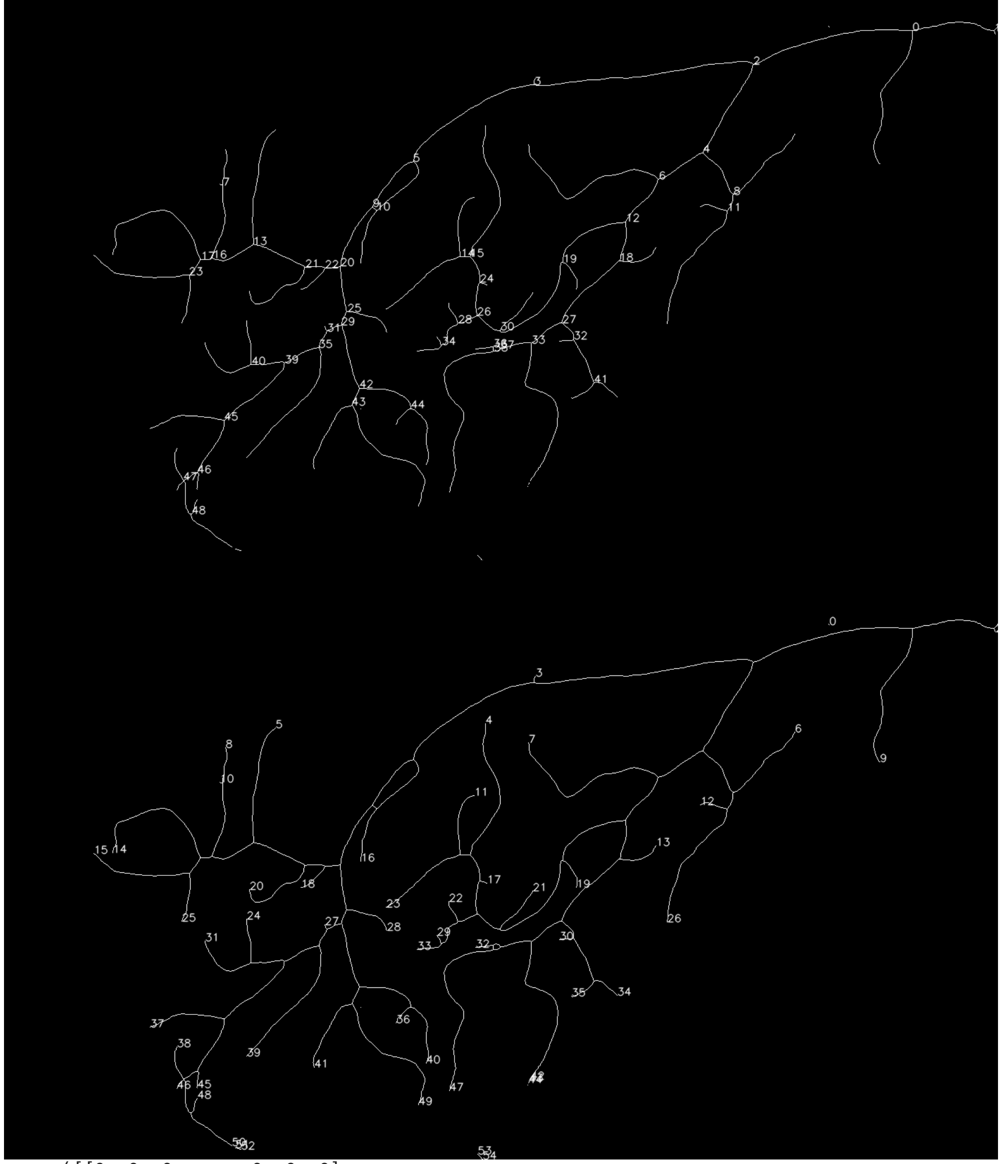
Also, note that the code provided at the beginning of the notebook assumes the input is a simple tree structure. However, given the results after image preprocessing, there could be cycles in the structure, so we need to handle this by treating it as a graph for robustness.

To solve the issue that many junctions packed together, which mostly likely are not due to the original biological structure, we can leverage an unsupervised machine learning algorithm to cluster neayby junctions into one.

```
from sklearn.cluster import DBSCAN
import random
random.seed(1)
# https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

# The maximum distance between two samples for them to be considered as in the same neighborhood.
# Considering that each pixel represents 162 nm in length
EPS = 2 # RADIUS

# Also could be used for leaf nodes
# we could adjust how we would like the clustered points to be created - random or centre
def cluster_nodes( nodes, eps=EPS, min_samples=1, random_choice=False):

    if not nodes:
        return []

    clustering = DBSCAN(eps=eps, min_samples=min_samples).fit(nodes)

    if random_choice:
      clustered_nodes = []
      for cluster_id in set(clustering.labels_):
          # Extract points that belong to the current cluster
          points = np.array(nodes)[clustering.labels_ == cluster_id]
          # Choose a random point from the cluster
          random_point = tuple(map(int, random.choice(points))) # shows location
          clustered_nodes.append(random_point)
      return clustered_nodes


    cluster_centers = []
    for cluster_id in set(clustering.labels_):
          points = np.array(nodes)[clustering.labels_ == cluster_id]
          centroid = np.mean(points, axis=0)
          cluster_centers.append(tuple(map(int, centroid)))

    return cluster_centers


clustered_junctions = cluster_nodes(junctions, eps=EPS, min_samples=1, random_choice=True)
clustered_leaf_nodes = cluster_nodes(leaf_nodes, eps=EPS, min_samples=1, random_choice=True)

print("Num of junctions: ", len(clustered_junctions))
print("Num of leaf nodes: ", len(clustered_leaf_nodes))
```

```
    Num of junctions:  49
    Num of leaf nodes:  55
```

```
mark_nodes_on_skeleton(skeleton, clustered_junctions)
mark_nodes_on_skeleton(skeleton, clustered_leaf_nodes)
```



```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

```
# Given all nodes on hand, we now track the edges and make a graph
# Note this appraoch leverages a simple DFS algorithm, which could be very computational expensive considering we have to trace the path pixe

def trace_path_dfs(skeleton, start, visited):

    visited.add(start)
    neighbors = find_neighbors(start, skeleton)
    neighbors = [n for n in neighbors if n not in visited]

    for neighbor in neighbors:
        if neighbor in all_nodes:
            return neighbor
        elif neighbor not in visited:
            result = trace_path_dfs(skeleton, neighbor, visited)
            if result:
                # If a node in 'all_nodes' is found in the recursion, return it
                return result

    # If no nodes in 'all_nodes' are found in any path, return None
    return None
```

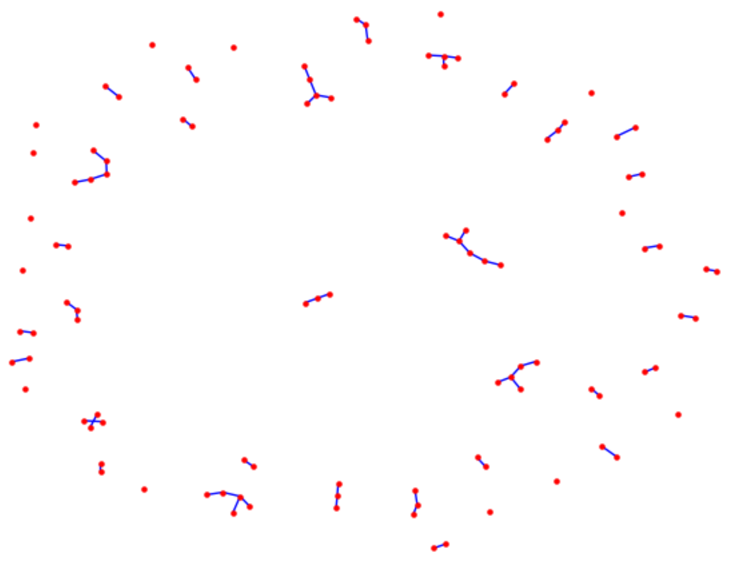
```
import networkx as nx

G = nx.Graph()

all_nodes = set(clustered_junctions + clustered_leaf_nodes)
print("Num of all nodes:", len(all_nodes))

# Add all nodes to the graph
for node in all_nodes:
    G.add_node(node)

visited = set()
for node in all_nodes:
    if node not in visited:
        end_node = trace_path_dfs(skeleton, node, visited)
        # There was a discrepency between the number of pre-added nodes and the final number of nodes in the graph
        # It's possibly caused by the clustering process, in which we created nodes that didn't originally exist.
        # consistency check edges
        if end_node and end_node in all_nodes and end_node != node:
            G.add_edge(node, end_node)
        # Add the current node to visited set
        visited.add(node)
```

```
Num of all nodes: 104
```

```
print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())
```

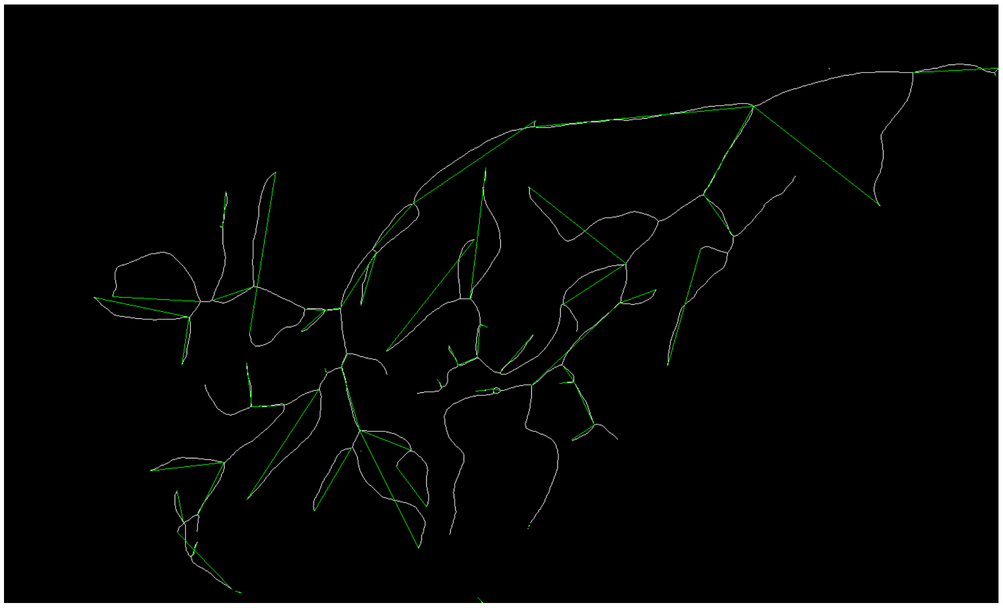
```
Number of nodes: 104
Number of edges: 57
```

```
nx.draw(G, node_color='red', edge_color='blue', node_size=5, with_labels=False)
```



We could notice that there are so many disconnections between componenents. To inspect why this happens, we draw all edges on original skeleton.

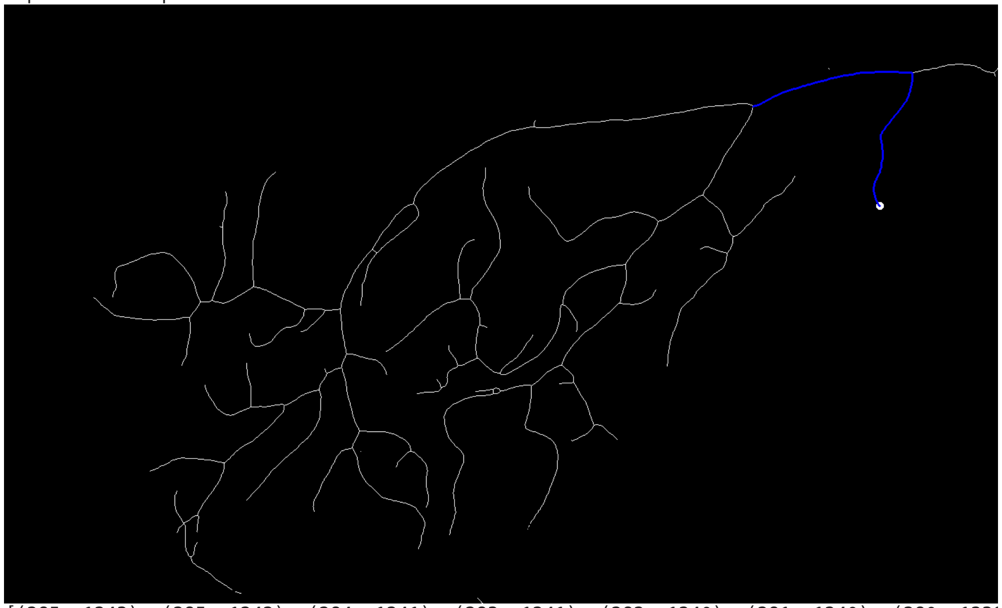
```
# Draw all edges on the original skeleton

def draw_edges_on_skeleton(skeleton, graph):
    # Ensure skeleton is in the right format (255 for lines, 0 for background)
    marked_img = (skeleton * 255).astype('uint8')

    # Convert the image to BGR format since we're going to draw green lines
    marked_img = cv2.cvtColor(marked_img, cv2.COLOR_GRAY2BGR)

    # Draw each edge in the graph
    for edge in graph.edges():
        point1 = edge[0]
        point2 = edge[1]

        # Swap coordinates due to cv2
        point1 = (point1[1], point1[0])
        point2 = (point2[1], point2[0])

        cv2.line(marked_img, point1, point2, (0, 255, 0), 1)

    # Display the image
    cv2_imshow(marked_img)



draw_edges_on_skeleton(skeleton, G)
```



```
# MAKE A MODIFIED TRACE PATH FUNCTION GIVEN THE INSPECTION RESULTS

def trace_path_dfs_modified(skeleton, start, visited, all_nodes, path):

    visited.add(start)
    path.append(start)

    connected_nodes = []
    neighbors = find_neighbors(start, skeleton)
    neighbors = [n for n in neighbors if n not in visited]

    if not neighbors:
        return []

    # Only edges between the node and nodes to whom they have direct connections can be added to the graph <--- THIS WILL BECOME A CONSTANT (
    for neighbor in neighbors:
        if neighbor in all_nodes:
            path.append(neighbor)
            if is_valid_path(skeleton, path, paths):
                paths.append(path.copy())
                connected_nodes.append(neighbor)
                break # BREAK THE LOOP ONCE A DESTINATION IS FOUND
        else:
            # find_neighbors will catch the node in all_nodes again
            # need visited for all_nodes
            connected_nodes.extend(trace_path_dfs_modified(skeleton, neighbor, visited, all_nodes, path))

        visited.add(neighbor)

    path.pop()

    return set(connected_nodes)



# ALSO CHECK IF IT SURPASSES OTHER NODES IN ALL_NODES.      <---- To counter the challenge that two nodes usually connect with each other if
def is_valid_path(skeleton, path, paths, overlap_threshold=0.1):
    # Check if path is on the skeleton and not overlapping with nodes in all_nodes
    for i in range(1, len(path)-1):
        if not skeleton[path[i]]:
            return False
        if path[i] in all_nodes:
            return False

    # Check for overlap with existing paths
    for existing_path in paths:
        if get_overlap_percentage(path, existing_path) > overlap_threshold:
            return False

    return True

def get_overlap_percentage(path1, path2):
    set_path1 = set(path1)
    set_path2 = set(path2)
    overlap = set_path1.intersection(set_path2)
    if not set_path1 or not set_path2:
        return 0
    return len(overlap) / min(len(set_path1), len(set_path2))
```

```
# (285, 1243) FOR DEBUGGING PURPOSE

TEST_NODE = (285, 1243)

paths = [[(285, 1243), (285, 1242), (284, 1241), (283, 1241), (282, 1240), (281, 1240), (280, 1239), (279, 1239), (278, 1238), (277, 1238), (


def draw_paths_on_skeleton(skeleton, paths, node):
    print("Input total", len(paths), "paths.")
    # Ensure skeleton is in the right format (255 for lines, 0 for background)
    marked_img = (skeleton * 255).astype('uint8')

    # Convert the image to BGR format since we're going to draw green lines
    marked_img = cv2.cvtColor(marked_img, cv2.COLOR_GRAY2BGR)

    x, y = node
    cv2.circle(marked_img, (y, x), 3, (255, 255, 255) , 3)

    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 0.5
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255)]
    radius = 1
    thickness = -1  # Negative thickness makes cv2.circle fill the circle

    for i, path in enumerate([paths[0]]):

      for x, y in path:
        cv2.circle(marked_img, (y, x), radius, colors[i%3], thickness)


    # Display the image
    cv2_imshow(marked_img)
    print(path)


draw_paths_on_skeleton(skeleton, paths, TEST_NODE)
```

```
Input total 1 paths.
```



```
[(285, 1243), (285, 1242), (284, 1241), (283, 1241), (282, 1240), (281, 1240), (280, 123
```

Looks like detected paths were not as accurate as we expect. However, since we cannot guarantee pixel level accuracy, which is set in the skeletonization step and relies a lot on the skeletonization algorithm we use.

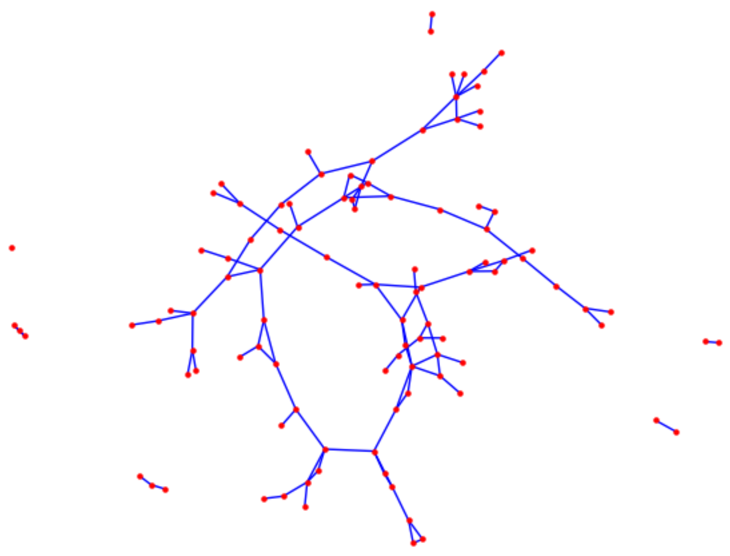
```
from collections import defaultdict

# REFRESH VISITED FOR EACH NODE

G_new = nx.Graph()

all_nodes = set(clustered_junctions + clustered_leaf_nodes)
# manual_adj_matrix = defaultdict(list)
print("Num of all nodes:", len(all_nodes))

# Add all nodes to the graph
for node in all_nodes:
    G_new.add_node(node)

# visited = set()
# i = 1
for node in all_nodes:
    visited = set() # refresh the visited for each node, each could possibily traverse the entire image
    path = []
    paths = []
    end_nodes = trace_path_dfs_modified(skeleton, node, visited, all_nodes, path)
    # print(i, len(paths), node, paths)
    # i += 1

    for end_node in end_nodes:
      # verify if the path is valid (the path is the length of activated pixels)
        if end_node != node:
            G_new.add_edge(node, end_node)
            # manual_adj_matrix[node].append(end_node)
```

```
    Num of all nodes: 104
```

```
print("Number of nodes:", G_new.number_of_nodes())
print("Number of edges:", G_new.number_of_edges())
```

```
    Number of nodes: 104
    Number of edges: 111
```

```
nx.draw(G_new, node_color='red', edge_color='blue', node_size=5, with_labels=False)
```



```
draw_edges_on_skeleton(skeleton, G_new)
```