Dept. of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur

ALGORITHMS
(EC31205)

Coding Task: 2

Title: Scanpath Similarity Analysis

Date of Submission: November 10th, 2023

Name: Irsh Vijay
Roll Number: 21EC30025

Instructors:
Prof. Debashis Sen

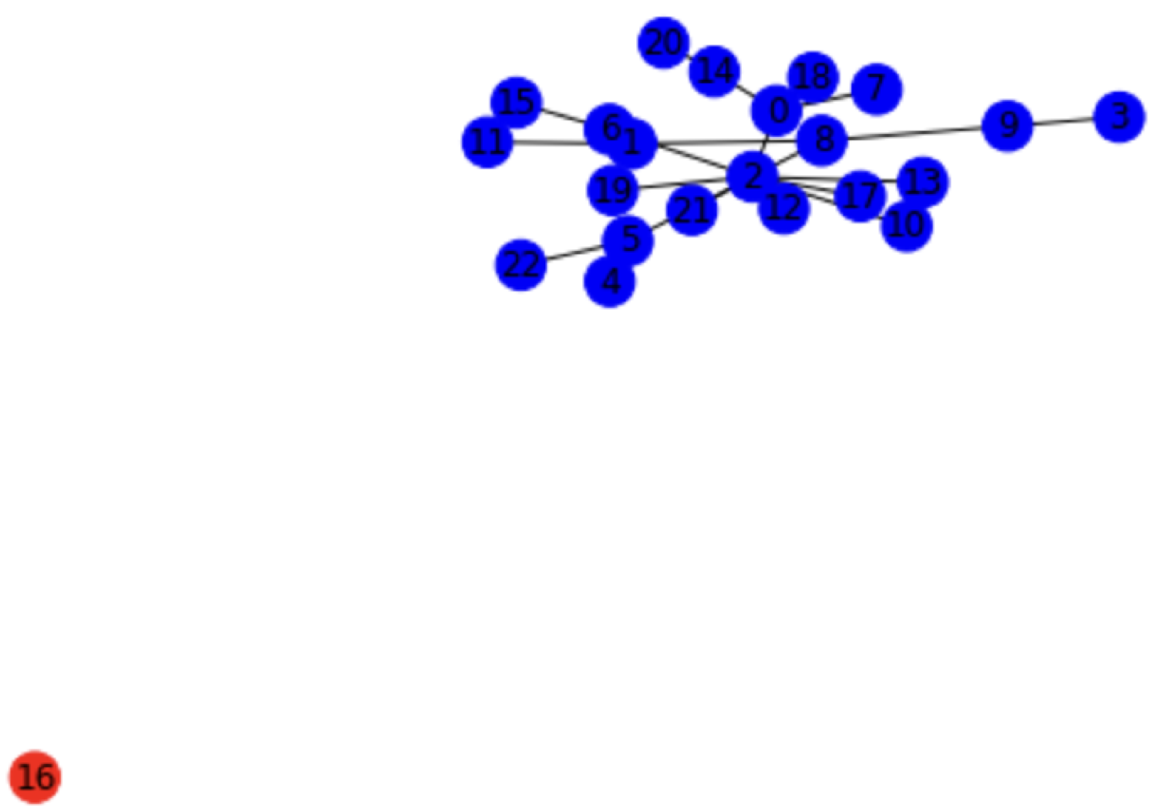
I. Scanpath Similarity Analysis:

Problem Statement:

A scanpath refers to the sequence of eye movements made by an individual's eyes as they explore and visually process a visual stimulus, such as an image, scene, or text.

- In part A, we load the dataset, compute the edit distance between the scanpaths of every pair of individuals, and store the results in a 2D list (Edit distance matrix). This matrix stores a similarity metric between pairs of scanpaths in the dataset.
- In part B, we use Prim's MST algorithm to construct an MST from the distance matrix obtained in part A, remove the largest edge in this MST (Thus dividing MST into two connected components), and label nodes 'red'/'blue' depending on their connected component membership using BFS/DFS. This gives us two sets of people whose scanpaths had the most edit distance.

Output with grid_size = 8:



The Code:

Reencode Scanpath function maps the scanpath coordinate to a specific grid cell of size grid_size * grid_size.

```
def reencode_scanpath(scanpath, grid_size):
  reencoded_array = []
  grid_size_x = w//grid_size
  grid_size_y = h//grid_size
  for point_x, point_y in scanpath:
    reencoded_array.append(int(point_x/w*grid_size_x)+int(point_y/h*grid_size_y)*(grid_size_x))
  return reencoded_array
```

We use dynamic programming to calculate edit distance between all pairs of reencoded scanpaths and store it in the distance_matrix.

```
def get_all_pair_edit_distances(scanpath_dataset, grid_size):
  distance_matrix = []
  for rows in range(total):
    distance_matrix.append([0]*total)
  #Your code here to fill distance_matrix

  for i in range(total):
    for j in range(i, total):
      scanpath_i = reencode_scanpath(scanpath_dataset[i], grid_size)
      scanpath_j = reencode_scanpath(scanpath_dataset[j], grid_size)

      def get_edit_distance(scanpath1, scanpath2):
        len1 = len(scanpath1)
        len2 = len(scanpath2)

        dp = [[0] * (len2 + 1) for _ in range(len1 + 1)]

        for i in range(len1 + 1):
          dp[i][0] = i
        for j in range(len2 + 1):
          dp[0][j] = j

        for i in range(1, len1 + 1):
          for j in range(1, len2 + 1):
            cost = 0 if scanpath1[i - 1] == scanpath2[j - 1] else 1
            dp[i][j] = min(
                dp[i - 1][j] + 1,
                dp[i][j - 1] + 1,
                dp[i - 1][j - 1] + cost
            )

        return dp[len1][len2]

      distance = get_edit_distance(scanpath_i, scanpath_j)

      distance_matrix[i][j] = distance
      distance_matrix[j][i] = distance

  # Your code ends here
  return distance_matrix
```

Prim's MST Algorithm is used to find the Minimum Spanning Tree and the edge with the largest weight is removed from the edge list.

```
def prim_mst(distance_matrix):
  n = len(distance_matrix)
  visited = [False] * n
  mst = []

  visited[0] = True

  while len(mst) < n - 1:
    min_edge = None
    min_weight = float('inf')

    for u in range(n):
      if visited[u]:
        for v in range(n):
          if not visited[v] and distance_matrix[u][v] < min_weight:
            min_edge = (u, v, distance_matrix[u][v])
            min_weight = distance_matrix[u][v]

    u, v, weight = min_edge
    mst.append((u, v, weight))
    visited[v] = True

  return mst
```

```
mst_edge_list_temp = [edge_info for edge_info in mst_edge_list]
mst_edge_list_temp.sort(key=lambda edge: edge[2])
print("Length of MST Edge List: ", len(mst_edge_list_temp))
largest_edge = mst_edge_list_temp.pop()
print("Largest Edge: ", largest_edge[2])

_ = mst_edge_list.pop(mst_edge_list.index(largest_edge))
```

Finally, BFS is used to label the nodes and color the components:

```
def label_components(mst_edge_list, n):
  graph = [[] for _ in range(n)]
  for u, v, _ in mst_edge_list:
    graph[u].append(v)
    graph[v].append(u)

  node_labels = [None] * n

  def bfs(node, label):
    queue = [node]
    while queue:
      current_node = queue.pop(0)
      node_labels[current_node] = label
      for neighbor in graph[current_node]:
        if node_labels[neighbor] is None:
          queue.append(neighbor)

  for i in range(n):
    if node_labels[i] is None:
      bfs(i, 'blue' if i == 0 else 'red')

  return node_labels

n = len(edit_distance_matrix)
node_labels = label_components(mst_edge_list, n)
```