Weekly Challenge: Identifying Connected Components Using DFS

March 7, 2025

1 Problem Statement

Given an undirected graph of locations and the distances between them, your task is to determine the number of connected components using Depth First Search (DFS). The tasks require:

1. Identifying the number of connected components in a given graph using DFS and visualizing them.
2. Creating a custom graph and identifying the number of connected components in the graph using DFS and visualizing them.

The definition of connected components is the same as that you have studied in the course. However, for this weekly challenge, there are some added conditions for the connected component. For an edge to be considered part of the graph, its weight must be less than the threshold T. Additionally, if a node is connected to more than one node and at least one of these connections has a weight less than the threshold, then, provided that the other conditions of the connected component are satisfied, it will be part of the connected component.

Removing an edge $e \in E$ from the graph $G = (V, E)$, where the distance of e is equal to or greater than a specified threshold, will alter the edge set E but will not affect the vertex set V. The set of vertices V remains unchanged, as the removal of an edge does not impact the vertices that are part of the graph.

2 Tasks for Students

1. **Task 1:** Given the following graph, use DFS to determine the number of connected components and visualize them. Use $T = 4$.
2. **Task 2:** Take T equal to your birthday month. Generate a graph of your choice, identify the number of connected components, and visualize. Note that for random generation, you will generate a graph of at least 15 nodes.

Location 1	Location 2	Distance
A	B	3
A	C	5
B	C	2
C	D	7
D	E	1
E	F	4
G	D	8

Table 1: Graph Data

3 Submission

You will submit:

1. a python file containing the code for finding the number of connected components and visualization.
2. a report containing the graph before and after running the algorithm for each task. Note that the visualization will be an image showing all the connected components.

4 Evaluation Criteria

- Correct implementation of the DFS-based connected components algorithm as specified in the class, incorporating the threshold condition. [40 points]
- Proper handling of isolated nodes. [20 points]
- Flexibility in processing different graph structures. [20 points]
- Quality of visualization.[20 points]

Weekly Challenge 10: Identifying Connected Components Using DFS

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

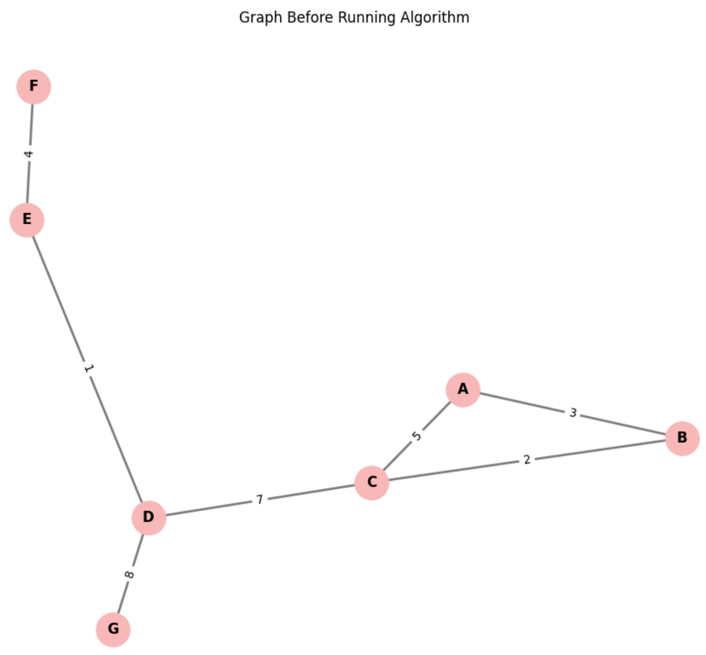
Task 1

Initial Graph

Location 1	Location 2	Distance
A	B	3
A	C	5
B	C	2
C	D	7
D	E	1
E	F	4
G	D	8

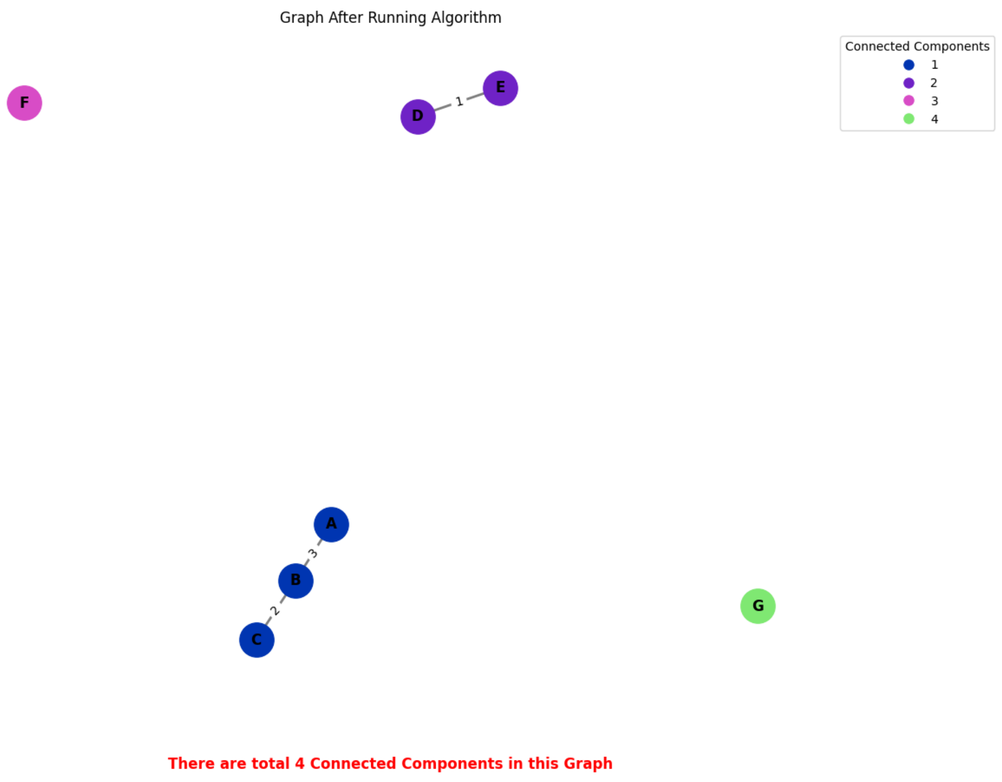
Table 1: Graph Data

The above given Graph Data is represented as follows in a Graph,



Connected Components

For the above graph the connected components I'm getting are {A, B, C}, {D, E}, {F}, {G}. Since our threshold $T = 4$ then we'll only keep edges where *distance* < 4 and discard the other edges where *distance* ≥ 4. So the resulting graph has $A - B - C$ as one connected component (with all edges < 4), $D - E$ another connected component (with edge < 4) while F and G are isolated (with no edge < 4).

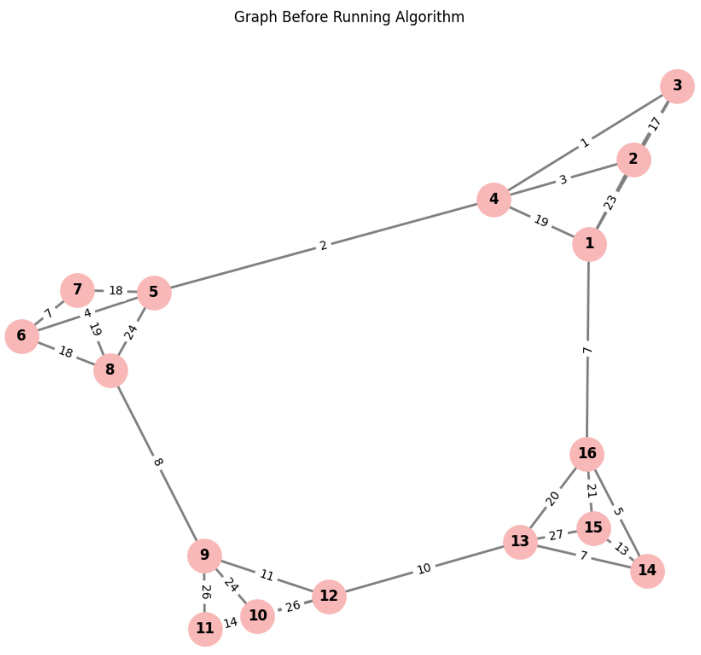


Task 2

The graph choice I've used for my algorithm is *ring_of_cliques* as follows,

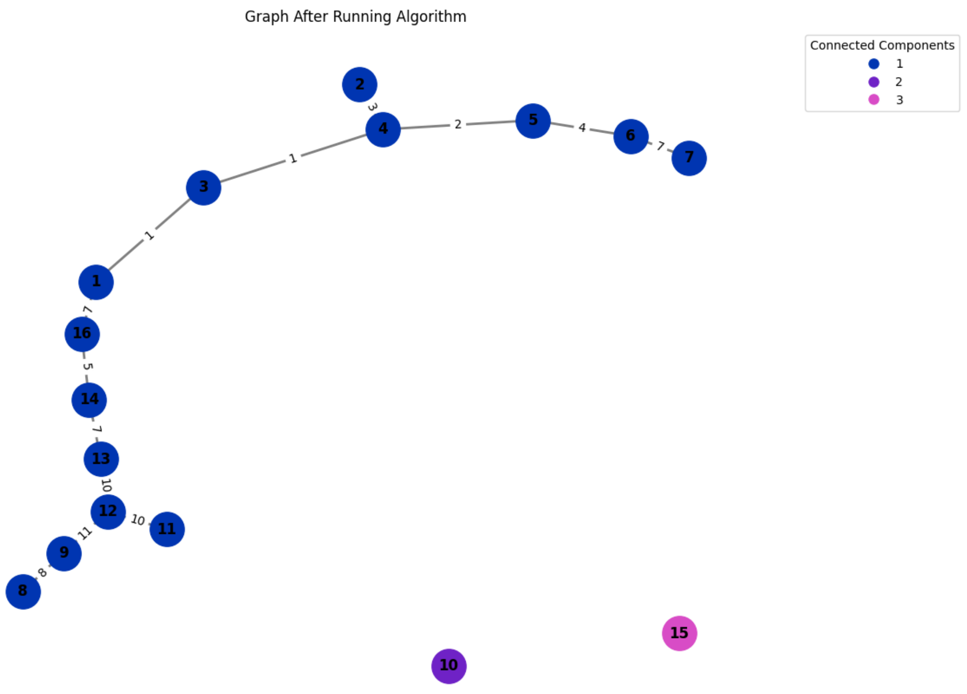
```
import networkx as nx
import random

def generate_ring_of_cliques(num_cliques, clique_size):
    G = nx.Graph()

    # Create cliques
    cliques = []
    for i in range(num_cliques):
        clique_nodes = [f"{i * clique_size + j + 1}" for j in range(clique_size)]
        cliques.append(clique_nodes)
        G.add_nodes_from(clique_nodes)
        for u in clique_nodes:
            for v in clique_nodes:
                if u != v:
                    weight = random.randint(1, 28)
                    G.add_edge(u, v, weight=weight)

    # Connect the cliques to form a ring
    for i in range(num_cliques):
        G.add_edge(cliques[i][-1], cliques[(i + 1) % num_cliques][0],
         weight=random.randint(1, 10))

    return G
```

Initial Graph



Connected Components

I've taken $T = 12$ according to my birthday month (December). So now to find connected components the nodes having *distance* ≥ 12 will be discarded while the nodes having *distance* < 12 will form a connected component in our graph.



There are total 3 Connected Components in this Graph

```
import networkx as nx
import matplotlib.pyplot as plt
import random

# TASK 1

# Graph Data
edges = [
    ('A', 'B', 3),
    ('A', 'C', 5),
    ('B', 'C', 2),
    ('C', 'D', 7),
    ('D', 'E', 1),
    ('E', 'F', 4),
    ('G', 'D', 8)
]

# Function to convert the given Graph Data to an actual Graph
def create_graph_with_all_edges(edges):
    G = nx.Graph()

    for u, v, weight in edges:
        G.add_edge(u, v, weight=weight)

    return G

# Function of DFS
def dfs(graph, node, visited, component, threshold):
    visited.add(node)
    component.add(node)

    for neighbor in graph[node]:
        edge_weight = graph[node][neighbor]['weight']

        if neighbor not in visited:

            # Condition 1:
            # if a node is connected
            # to more than one node and at least one of these connections has a
weight less
            # than the threshold, then, provided that the other conditions of the
connected
            # component are satisfied, it will be part of the connected component
            if edge_weight < threshold or any(graph[node][w]['weight'] < threshold
for w in graph[node]):
                dfs(graph, neighbor, visited, component, threshold)


# Function to find connected components using DFS
def find_connected_components(graph, threshold):
    visited = set()
    components = []

    for node in graph.nodes():
        if node not in visited:
            component = set()
            dfs(graph, node, visited, component, threshold)
            components.append(component)
    print("Identified Connected Components:", components)

    # Proper handling of isolated nodes
    isolated_nodes = set(graph.nodes()) - visited
    for isolated_node in isolated_nodes:
        components.append({isolated_node})

    return components

# Step 1: Creating Graph from given Graph Data by calling its function
G_task_1 = create_graph_with_all_edges(edges)

# Function to visualize graph 'before' running algorithm
def visualize_graph_before(graph, task="Task 1"):
    pos = nx.spring_layout(graph, seed=42, k=0.5)
    plt.figure(figsize=(12, 10))

    # Draw
    nx.draw_networkx_edges(graph, pos, width=2, edge_color='gray')
    nx.draw_networkx_nodes(graph, pos, node_color='#f9b8b8', node_size=800)
    nx.draw_networkx_labels(graph, pos, font_size=12, font_weight="bold")
    edge_labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels, font_size=10)

    plt.title(f"Graph Before Running Algorithm")
    plt.axis('off')
    plt.savefig(f"old_{task.lower().replace(' ', '_')}.png")
    plt.show()

visualize_graph_before(G_task_1, task="Task 1")

# Step 2: Setting threshold to 4 as given in assignment
T = 4

# Step 3: Finding connected components for Task 1
components_task_1 = find_connected_components(G_task_1, T)

# Condition 2:
# Removing an edge e ∈ E from the graph G = (V,E), where the distance of
# e is equal to or greater than a specified threshold, will alter the edge set E
but
# will not affect the vertex set V . The set of vertices V remains unchanged, as
# the removal of an edge does not impact the vertices that are part of the graph.
def remove_edges_above_threshold(graph, threshold):
    edges_to_remove = [(u, v) for u, v, data in graph.edges(data=True) if
data['weight'] >= threshold]
    graph.remove_edges_from(edges_to_remove)

# Step 4: Removing edges if any (whose weight is >= threshold)
remove_edges_above_threshold(G_task_1, T)

# Step 5: Finding connected components again after edge removal
components_after_removal = find_connected_components(G_task_1, T)

# Function to visualize graph 'after' running algorithm
def visualize_graph_after(graph, components, task="Task 1"):
    pos = nx.spring_layout(graph, seed=42, k=0.5)
    plt.figure(figsize=(12, 10))
    # Draw
    nx.draw_networkx_edges(graph, pos, width=2, edge_color='gray')
    colors = ['#0035b1', '#6f22c6', '#d84cc6', '#7fe872', '#fdffa9', '#f09609',
'#00aba9', '#ff0097']

    for i, component in enumerate(components):
        nx.draw_networkx_nodes(graph, pos, nodelist=list(component),
node_color=colors[i % len(colors)], node_size=800)

    nx.draw_networkx_labels(graph, pos, font_size=12, font_weight='bold')
    edge_labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels, font_size=10)
    patches = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor=colors[i
% len(colors)], markersize=10, label=f'{i+1}') for i in range(len(components))]
    plt.legend(handles=patches, loc='upper left', bbox_to_anchor=(1.05, 1),
title="Connected Components")

    num_components = len(components)
    plt.text(0.5, -0.1, f"There are total {num_components} Connected Components in
this Graph",
             horizontalalignment='center', verticalalignment='center',
             transform=plt.gca().transAxes, fontsize=12, color='red',
weight='bold')

    plt.title(f"Graph After Running Algorithm")
    plt.axis('off')
    plt.savefig(f"new_{task.lower().replace(' ', '_')}.png", bbox_inches='tight')
    plt.show()

visualize_graph_after(G_task_1, components_after_removal, task="Task 1")

# TASK 2

# Function create a Graph of our own choice (I used ring of cliques)
def generate_ring_of_cliques(num_cliques, clique_size):
    G = nx.Graph()

    # Create cliques
    cliques = []
    for i in range(num_cliques):
        clique_nodes = [f"{i * clique_size + j + 1}" for j in range(clique_size)]
        cliques.append(clique_nodes)
        G.add_nodes_from(clique_nodes)
        for u in clique_nodes:
            for v in clique_nodes:
                if u != v:
                    weight = random.randint(1, 28)  # Random weights between 1 and
28
                    G.add_edge(u, v, weight=weight)

    for i in range(num_cliques):
        G.add_edge(cliques[i][-1], cliques[(i+1) % num_cliques][0],
weight=random.randint(1, 10))

    return G

# Step 1: Creating Graph with at least 15 nodes
G_task_2 = generate_ring_of_cliques(4, 4)
```

```
# Calling function to visualize graph 'before' running algorithm
visualize_graph_before(G_task_2, task="Task 2")

# Step 2: Setting threshold to 12 as my birthday month is December
T_task_2 = 12

# Step 3: Finding connected components for Task 2
components_task_2 = find_connected_components(G_task_2, T_task_2)

# Step 4: Removing edges if any (whose weight is >= threshold)
remove_edges_above_threshold(G_task_2, T_task_2)

# Step 5: Finding connected components again after edge removal
components_after_removal_task_2 = find_connected_components(G_task_2, T_task_2)

# Calling function to visualize graph 'after' running algorithm
visualize_graph_after(G_task_2, components_after_removal_task_2, task="Task 2")
```