HW7

Keerthana Golla

October 2023

1 Problem -1

Since the Tmst of Graph G is already given , we taking that mst in below code - it is undirected weighted graph

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, w):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, w))
        self.graph[v].append((u, w))

    def dfs(self, start, end, visited, path, weights):
        visited[start] = True
        path.append(start)

        if start == end:
            print("Path from {} to {}:".format(path[0],path[-1]), " -> ".join(map(str, path)))
            print("Maximum edge weight in the path:", max(weights))
            if(max(weights)<new_edge_weight):
              print("Mst remains same")
            else:
              print("Mst changes")
            return True

        for neighbor, weight in self.graph[start]:
            if not visited[neighbor]:
                if self.dfs(neighbor, end, visited, path, weights + [weight]):
                    return True

        path.pop()
        return False

    def find_path_in_mst(self, start, end):
        visited = {node: False for node in self.graph}
        path = []
        weights = []
        self.dfs(start, end, visited, path, weights)
```

```
mst = Graph()
mst.add_edge(2, 3, 2)
mst.add_edge(2, 4, 1)
mst.add_edge(4, 1, 3)
mst.add_edge(4, 5, 6)

new_edge_start_node = 2
new_edge_end_node = 1
new_edge_weight=1
mst.find_path_in_mst(new_edge_start_node, new_edge_end_node)
```

Note that no.of vertices are not changing, we are not adding any additional vertex, we are just adding an edge to the graph that means we can see what happens if that edge is going to be added to Tmst that is given and see if its going to be the minimum spanning tree for new graph after the addition of edge.

The above algorithm checks if there exists a path between two nodes where the new edge is added in the minimum spanning tree (Tmst) of a graph G, and whether the maximum weight in that path is less than a given new edge weight. The goal is to determine if adding the new edge changes the minimum spanning tree or not

Correctness of Algorithm:

1. Properties of minimum spanning tree:
   - Adding an edge to a minimum spanning tree creates a cycle because it introduces a direct connection between vertices, violating the acyclic property of a tree.
2. Depth-First Search (DFS):
   - The algorithm utilizes Depth-First Search (DFS) to find a path between the given `start` and `end` nodes in the graph.
   - DFS ensures that all possible paths are explored.
3. Path Existence Check:
   - since minimum spanning tree(Tmst) of graph G is already know , we will try getting the path using DFS from start node to end node of the new edge that is being added
   - The algorithm correctly identifies whether a path exists between `start` and `end` in the minimum spanning tree (Tmst).
   - If such a path is found, it proceeds to check the maximum edge weight in the path.This is always true since minimum spanning tree covers all the vertices and has a path from one vertex to any other vertex.
4. Maximum Edge Weight Check:
   - The algorithm correctly identifies the maximum edge weight in the path between `start` and `end`.
   - It then compares this maximum weight with the weight of the new edge (`new_edge_weight`).
5. MST Stability Check:
   - If the maximum weight in the path is less than the new edge weight, it concludes that the minimum spanning tree (Tmst) remains unchanged.
   - Otherwise, it correctly identifies that the minimum spanning tree would change with the addition of the new edge.since there exists an edge with weight greater than the new edge and that can be removed in the new cycle that is being created by adding the new edge in the path from start node to end node.

Time Complexity:

- We know that the DFS process traverses each node and each vertex once, resulting in a time complexity of $O(|V+E|)$.But we also know that the maximum number of edges in Minimum spanning tree is $V-1$
- So, **The overall time complexity of the algorithm is** $O(|V+(V-1)|) = O(|V|)$ since the main operation is the DFS on **minimum spanning tree**.

2 Problem - 2

Assuming The graph $G = G(V, E)$ to be connected undirected graph.

```
def is_good_graph(graph):
    n = len(graph)  # Number of vertices
    V1, V2 = set(), set()  # Initialize sets V1 and V2
    colors = [-1] * n  # -1 indicates unassigned

    def dfs(vertex, color):
        nonlocal V1, V2, colors

        # Assign color to the current vertex
        colors[vertex] = color

        # Add vertex to the corresponding set
        if color == 0:
            V1.add(vertex)
        else:
            V2.add(vertex)

        # Explore neighbors
        for neighbor in graph[vertex]:
            if colors[neighbor] == -1:
                # Neighbor is not colored, assign the opposite color
                if not dfs(neighbor, 1 - color):
                    return False
            elif colors[neighbor] == color:
                # Neighbor has the same color, not a "good" graph
                return False

        return True

    # Start DFS from an arbitrary vertex
    for vertex in range(n):
        if colors[vertex] == -1:
            if not dfs(vertex, 0):
                return False

    # Check if all edges are crossing edges
    for vertex in range(n):
        for neighbor in graph[vertex]:
            if colors[vertex] == colors[neighbor]:
                return False

    # The graph is "good"
    return True

graph = {
    0: [1],
    1: [2],
    2: [3,0],
    3: [0]
}
```

```
result = is_good_graph(graph)
print(result)
```

Correctness An edge that connects two vertices that are neither ancestors nor descendants of each other in the DFS tree.A cross edge essentially "crosses" levels of the DFS tree. Unlike tree edges, which form the structure of the DFS tree, and back and forward edges, which indicate relationships within the tree, cross edges connect unrelated parts of the DFS tree.

Ensuring that all edges are cross edges, is a way to guarantee that vertices with different colors are connected. If all edges are cross edges, then adjacent vertices must have different colors because they are not part of each other's DFS subtree. This property is important for bipartite graph coloring, where you want to color the vertices with two colors in such a way that no two adjacent vertices have the same color.

1. Graph Coloring: The algorithm correctly colors the vertices of the graph using two colors (0 and 1) in a way that no two adjacent vertices have the same color. This is ensured by the depth-first search (DFS) traversal.

2. Sets V_1 and V_2:The algorithm creates two sets, V_1 and V_2, representing vertices with color 0 and color 1, respectively. The sets are constructed during the DFS traversal and are disjoint ($V_1 \cap V_2 = \emptyset$).

3. Crossing Edges: The algorithm checks if all edges in the graph are crossing edges. This is verified after coloring the vertices. For any edge (a, b) in the graph, either a is in V_1 and b is in V_2, or vice versa. This property is maintained by the algorithm.

4. Correct Termination:The algorithm terminates successfully if it colors all vertices without encountering adjacent vertices with the same color. In this case, it checks for crossing edges, and if the condition holds for all edges, the graph is declared "good."

5. Conclusion: The algorithm is correct as it satisfies the conditions for a "good" graph, and it runs in the specified time complexity. The use of DFS ensures efficient exploration of the graph, and the coloring scheme is designed to identify sets V_1 and V_2 such that all edges are crossing edges. The algorithm is suitable for graphs represented in the adjacency list format, making it practical for various applications.

Running Time

The running time of the algorithm is $O(|V| + |E|)$ because: The DFS traversal visits each vertex once, and for each vertex, it explores its neighbors. This results in $O(|V| + |E|)$ time complexity.