23CCE201 - Data Structures

Name: Anirudh Anand

Roll No.: CB.EN.U4CCE23002

Undirected Graphs – Minimum Spanning Trees

I.Aim:

To implement the various schemes to obtain Minimum Spanning Trees Weighted Undirected Graphs.

II. Logic:

The two traversal schemes implemented in this report are the Prim's and Kruskal's algorithm

- **Prim's Algorithm:**
  The Prim's Algorithm is a greedy algorithm that is used to find a Minimum Spanning Tree from a weighted undirected graph. It starts from an arbitrary node and the tree is expanded one vertex at a time by adding the next closest vertex that connects to the current MST.
- **Kruskal's Algorithm:**
  Kruskal's is also a greedy algorithm but it differs from Prim's as it considers all the edges at once regardless of whether they connect to the MST or not. It sorts all the edges by their weights then considers each of them and adds it to the MST if it doesn't form a cycle.

III. Algorithm:

- **Prim's Algorithm**
  1. Select any one vertex from the graph in the Minimum Spanning Tree(MST).
  2. Initialise two sets one for vertices in the MST and one for vertices not yet added
  3. We keep track of the minimum weight edge for each vertex in the MST that connects to another vertex not in the MST
  4. Add this edge and the connected node to the MST
  5. Repeat steps 3 and 4 until all nodes are included in the MST
  6. End

- **Kruskal's Algorithm:**
  1. Sort all edges in the graph in ascending order based on their weights
  2. Initialise an empty MST that will contain the edges
  3. Add the minimum weight edge to the MST as long as it doesn't form a cycle with other edges already in the MST
  4. Repeat until the MST contains V-1 edges where V is the number of vertices.

IV. Code:

```
from heapq import heappop, heappush
from collections import defaultdict


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)  # For Prim's
        self.edges = []  # For Kruskal's


    def add_edge(self, u, v, weight):
        # For Prim's Algorithm
        self.graph[u].append((weight, v))
        self.graph[v].append((weight, u))
        # For Kruskal's Algorithm
        self.edges.append((weight, u, v))


    # Prim's Algorithm
    def prim_mst(self):
        mst_cost = 0
        visited = set()
        min_heap = [(0, 0)]  # Start from node 0 with a cost of 0
        mst_edges = []


        while min_heap and len(visited) < self.V:
```

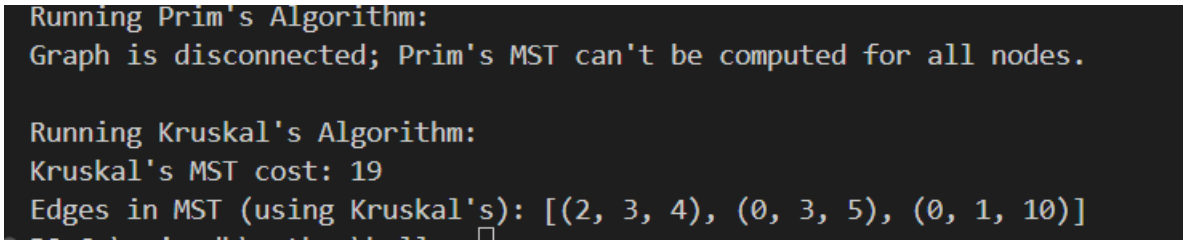
```
        weight, u = heappop(min_heap)
        if u in visited:
            continue
        visited.add(u)
        mst_cost += weight
        mst_edges.append(u)

        for w, v in self.graph[u]:
            if v not in visited:
                heappush(min_heap, (w, v))

    # Check if all nodes are visited (graph is connected)
    if len(visited) != self.V:
        print("Graph is disconnected; Prim's MST can't be computed for all nodes.")
    else:
        print("Prim's MST cost:", mst_cost)
        print("Edges in MST (using Prim's):", mst_edges)

# Helper functions for Kruskal's Algorithm
def find(self, parent, i):
    if parent[i] == i:
        return i
    return self.find(parent, parent[i])

def union(self, parent, rank, x, y):
    root_x = self.find(parent, x)
    root_y = self.find(parent, y)
    if rank[root_x] < rank[root_y]:
        parent[root_x] = root_y
    elif rank[root_x] > rank[root_y]:
```

```
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    # Kruskal's Algorithm
    def kruskal_mst(self):
        self.edges.sort()  # Sort edges by weight
        parent = []
        rank = []
        mst_cost = 0
        mst_edges = []

        # Initialize disjoint sets for each vertex
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        for weight, u, v in self.edges:
            root_u = self.find(parent, u)
            root_v = self.find(parent, v)

            # Only add the edge if it doesn't form a cycle
            if root_u != root_v:
                mst_cost += weight
                mst_edges.append((u, v, weight))
                self.union(parent, rank, root_u, root_v)

        print("Kruskal's MST cost:", mst_cost)
        print("Edges in MST (using Kruskal's):", mst_edges)
```

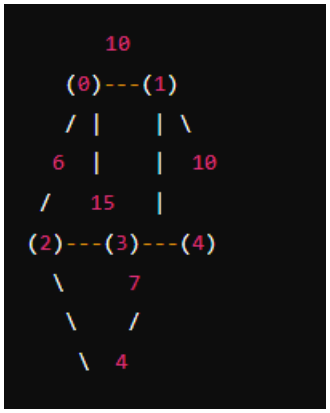
```
g = Graph(5)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 6)
g.add_edge(0, 3, 5)
g.add_edge(1, 3, 15)
g.add_edge(2, 3, 4)

print("Running Prim's Algorithm:")
g.prim_mst()

print("\nRunning Kruskal's Algorithm:")
g.kruskal_mst()
```

V. Results:

Disconnected Graph:



Connected Graph:



```
Running Prim's Algorithm:
Prim's MST cost: 26
Vertices in MST (using Prim's): [0, 3, 2, 4, 1]

Running Kruskal's Algorithm:
Kruskal's MST cost: 26
Edges in MST (using Kruskal's): [(2, 3, 4), (0, 3, 5), (3, 4, 7), (0, 1, 10)]
PS C:\anirudh\python\hello>
```

VI. Inference:

Thus, we have performed the Prim's and Kruskal's schemes on a given weighted undirected graph and have obtained a Minimum Spanning Tree starting from vertex 0. Prim's algorithm then expands from this starting vertex, adding the closest unvisited vertex at each step. Since there's no specific starting vertex required by Kruskal's algorithm (as it processes edges globally), Kruskal's doesn't have a defined starting point—rather, it sorts and examines edges based on weight. We obtain the same unique MST from both the schemes.