UNIVERSITY INSTITUTE of
COMPUTING
Asia's Fastest Growing University

CU
CHANDIGARH
UNIVERSITY

NAAC
GRADE A+
ACCREDITED UNIVERSITY

Worksheet- 3

Student Name: Harsh UID: 24MCA20045

Branch: MCA Section/Group: 1(A)

Semester: 2 Date of Performance:24/02/2025

Subject Name: Design and Analysis of Algorithms Lab Subject Code: 24CAP-612

1. (a) **Aim/Overview of the practical**:

To implement Kruskal's Algorithm to find the Minimum Cost Spanning Tree (MST) of a given undirected weighted graph using the Greedy approach.

2. **Task to be done:**

1. Take input for the number of vertices and edges in the graph.
2. Store and sort the edges in ascending order based on weight.
3. Use Union-Find (Disjoint Set Union - DSU) to avoid cycles while selecting edges.
4. Select edges one by one until we form an MST with n-1 edges (where n is the number of vertices).
5. Output the MST edges and the total minimum cost.

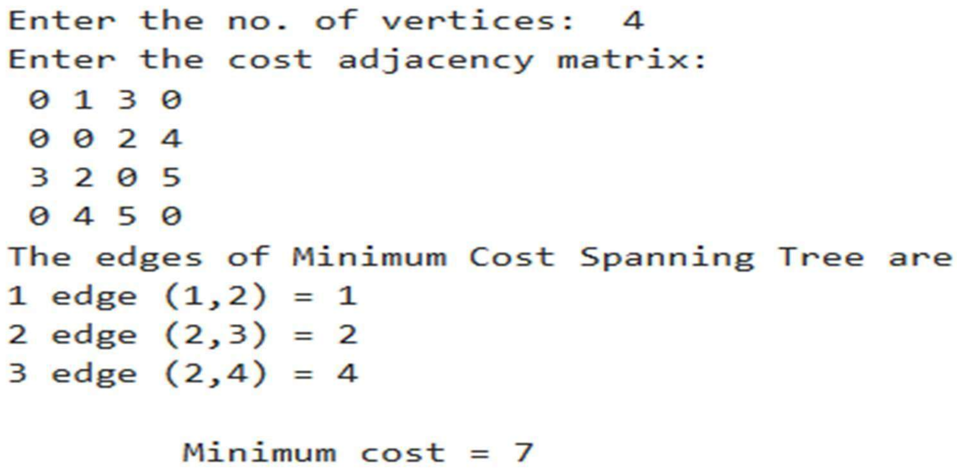
3. **Algorithm:**

1. Input the graph (vertices, cost matrix).
2. Sort edges by weight.
3. Use find() and union() to avoid cycles.
4. Select edges until MST has (n-1) edges.
5. Print MST edges and minimum cost.

4. Code for experiment/practical:

```
class KruskalMST:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
        self.parent = [0] * (vertices + 1)

    def add_edge(self, u, v, weight):
        self.graph.append((weight, u, v))

    def find(self, i):
        if self.parent[i] == 0:
            return i
        return self.find(self.parent[i])

    def union(self, i, j):
        if i != j:
            self.parent[j] = i
            return True
        return False

    def kruskal(self):
        self.graph.sort()  # Sort edges by weight
        mincost = 0
        ne = 0
        print("The edges of Minimum Cost Spanning Tree are")

        for weight, u, v in self.graph:
            if ne >= self.V - 1:
                break
            u_set = self.find(u)
            v_set = self.find(v)

            if self.union(u_set, v_set):
                print(f"{ne + 1} edge ({u},{v}) = {weight}")
                mincost += weight
                ne += 1

        print(f"\n\tMinimum cost = {mincost}")

# Taking input from user
n = int(input("Enter the number of vertices: "))
mst = KruskalMST(n)

print("Enter the cost adjacency matrix:")
cost = []
for i in range(1, n + 1):
    row = list(map(int, input().split()))
    for j in range(1, n + 1):
        if row[j - 1] != 0:  # Ignoring self-loops
            mst.add_edge(i, j, row[j - 1])
mst.kruskal()
```

6.**Output:**



```
Enter the no. of vertices:  4
Enter the cost adjacency matrix:
 0 1 3 0
 0 0 2 4
 3 2 0 5
 0 4 5 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,2) = 1
2 edge (2,3) = 2
3 edge (2,4) = 4

        Minimum cost = 7
```

Output of time complexity – O(ElogE)

(b) **Aim /Overview of the practical:**

To implement Topological Sorting using Depth First Search (DFS) for a Directed Acyclic Graph (DAG).

Task to be done –

1. Take input for vertices and directed edges.
2. Construct the adjacency list for the graph.
3. Perform DFS-based Topological Sorting.
4. Use a stack to store the topological order.
5. Print the final topological sort order.

Algorithm-

1. Initialize graph as an adjacency list and a visited list.
2. Take input for the number of vertices and edges.
3. Construct the graph by adding directed edges.
4. Perform DFS on each unvisited node:

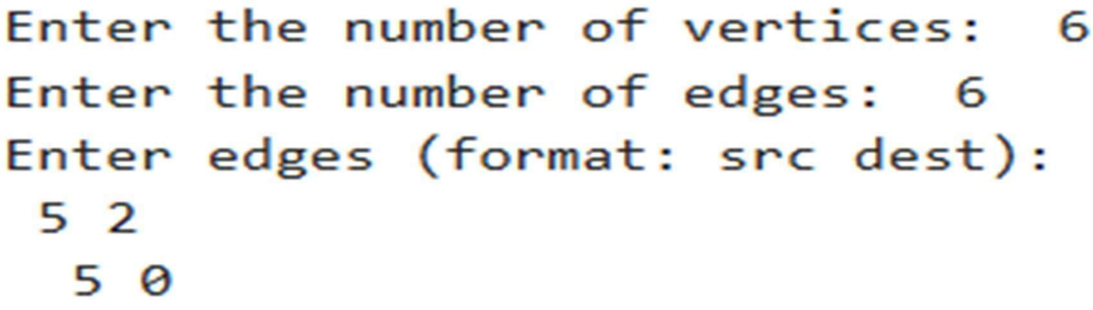
- Mark the node as visited.
- Recursively visit all its unvisited neighbors.
- Push the node onto the stack after visiting all neighbors.

5. Print the topological order (reverse of stack).

7. **Code for experiment/practical**:

```
    from collections import defaultdict

def topological_sort_util(v, adj, visited, rec_stack, stack):
    visited[v] = True
    rec_stack[v] = True

    for i in adj[v]:
        if not visited[i]:
            topological_sort_util(i, adj, visited, rec_stack, stack)
        elif rec_stack[i]:
        raise RuntimeError("Cycle detected!Topological
sorting is not possible.")

    rec_stack[v] = False
    stack.append(v)

def topological_sort(adj, V):
    stack = []
    visited = [False] * V
    rec_stack = [False] * V

    for i in range(V):
        if not visited[i]:
            topological_sort_util (i, adj, visited, rec_stack, stack)

    return stack[::-1]  # Reverse to get the correct order

def main():
    V = int(input("Enter the number of vertices: "))
    E = int(input("Enter the number of edges: "))

    adj = defaultdict(list)

    print("Enter edges (format: src dest):")
    for _ in range(E):
        src, dest = map(int, input().split())
        adj[src].append(dest)

    try:
        ans = topological_sort(adj, V)
        print("Topological Sort Order:", ans)
    except RuntimeError as e:
        print(e)

if __name__ == "__main__":
    main()
```

8. **Output**:



```
Enter the number of vertices:  6
Enter the number of edges:  6
Enter edges (format: src dest):
 5 2
  5 0
```

Time Complexity: O(V+E)

9. **Learning Outcomes-**

1. Learn to store graphs using lists and matrices.
2. Find the Minimum Spanning Tree using Kruskal's algorithm.
3. Perform Topological Sorting using DFS.
4. Detect cycles using the union-find method.
5. Understand the efficiency of graph algorithms.

Evaluation Grid:

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.	Conduct		12
2.	Worksheet		8
3.	Viva		10