**UIT 2402 – ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS**

**EX 6: Kruskal’s Algorithm**

**Name:** B.VASUNDHARA

**Roll no:** 3122 21 5002 119

**Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

**ALGORITHM:**

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

**Program Code:**

# Class to represent a graph

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = []

    # Function to add an edge to graph

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    # A utility function to find set of an element i

    # (truly uses path compression technique)

    def find(self, parent, i):

        if parent[i] != i:

            # Reassignment of node's parent

            # to root node as

            # path compression requires

            parent[i] = self.find(parent, parent[i])

        return parent[i]

    # A function that does union of two sets of x and y

    # (uses union by rank)

    def union(self, parent, rank, x, y):

        # Attach smaller rank tree under root of

        # high rank tree (Union by Rank)

        if rank[x] < rank[y]:

            parent[x] = y

        elif rank[x] > rank[y]:

            parent[y] = x

        # If ranks are same, then make one as root

        # and increment its rank by one

        else:

            parent[y] = x

            rank[x] += 1

    # The main function to construct MST

    # using Kruskal's algorithm

    def KruskalMST(self):

        # This will store the resultant MST

        result = []

        # An index variable, used for sorted edges

        i = 0

        # An index variable, used for result[]

        e = 0

        # Sort all the edges in

        # non-decreasing order of their

        # weight

        self.graph = sorted(self.graph,

                            key=lambda item: item[2])

        parent = []

        rank = []

        # Create V subsets with single elements

        for node in range(self.V):

            parent.append(node)

            rank.append(0)

        # Number of edges to be taken is less than to V-1

        while e < self.V - 1:

            # Pick the smallest edge and increment

            # the index for next iteration

            u, v, w = self.graph[i]

            i = i + 1

            x = self.find(parent, u)

            y = self.find(parent, v)

            # If including this edge doesn't

            # cause cycle, then include it in result

            # and increment the index of result

            # for next edge

            if x != y:

                e = e + 1

                result.append([u, v, w])

                self.union(parent, rank, x, y)

            # Else discard the edge

        minimumCost = 0

        print("Edges in the constructed MST")

        for u, v, weight in result:

            minimumCost += weight

            print("%d -- %d == %d" % (u, v, weight))

        print("Minimum Spanning Tree", minimumCost)

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    g = Graph(4)

    g.addEdge(0, 1, 10)

    g.addEdge(0, 2, 6)

    g.addEdge(0, 3, 5)

    g.addEdge(1, 3, 15)

    g.addEdge(2, 3, 4)

    # Function call

    g.KruskalMST()

**OUTPUT:**

A screenshot of a computer

Description automatically generated