23CCE201 Data Structures

Name : Tarun Sri Vathsan K

Roll No : CB.EN.U4CCE23058

AIM :

- To implement various schemes to obtain minimum spanning trees for weighted undirected graphs.

LOGIC :

Prim's Algorithm:

Prim's algorithm constructs the Minimum Spanning Tree (MST) by focusing on vertices. It starts with an arbitrary vertex, adding it to the MST and initializing a process of progressively growing the MST one edge at a time. At each step, the algorithm selects the minimum-weight edge that connects a vertex already in the MST to a vertex outside of it. This approach ensures that each new edge added to the MST is the smallest possible edge that maintains the tree structure. This selection process repeats until all vertices are part of the MST.

Kruskal's Algorithm:

Kruskal's algorithm builds the Minimum Spanning Tree (MST) by focusing on edges. The process starts by sorting all edges of the graph in ascending order based on their weights. Then, it iterates through this sorted list, selecting each edge in turn. For each edge, the algorithm checks if adding it to the MST would create a cycle with the edges already in the MST. A data structure called Union-Find or Disjoint Set Union (DSU) is used, which keeps track of which vertices are connected. If an edge doesn't form a cycle, it's added to the MST; otherwise, it's skipped. This process repeats until the MST contains exactly $V-1$ (where V is the number of vertices in the graph), at which point the MST is complete.

ALGORITHM :

Prim’s Algorithm:

1. Initialize the graph with the number of vertices and an adjacency matrix where each entry represents the weight of the edge between two vertices.
2. Set up an array, key, to hold the minimum edge weights required to add each vertex to the MST. Initialize all entries in key to infinity except for the first vertex, which is set to zero.
3. Create a parent array to store the MST structure, with each entry initialized to -1. This array will keep track of the parent node for each vertex in the MST.
4. Initialize a boolean array, mstSet, with values set to False for each vertex. This array will indicate whether a vertex has been included in the MST.
5. For each vertex in the graph, perform the following:
6. Select the vertex u that has the minimum key value among vertices not yet included in the MST.
7. Set mstSet[u] to True to mark u as included in the MST.
8. For each adjacent vertex v of u, check if there is an edge between u and v (i.e., if the weight is non-zero), v is not yet in the MST, and the weight of the edge (graph[u][v]) is less than the current key value of v.
9. If these conditions are met, update key[v] with the new weight and set parent[v] to u.
10. Once all vertices have been added to the MST, print the edges by iterating through the parent array. For each vertex i starting from 1, print the edge from parent[i] to i with its corresponding weight from the adjacency matrix.

Kruskal’s Algorithm :

1. Initialize the graph with a specified number of vertices and an empty list to hold the edges.
2. Prompt the user to input an adjacency matrix, representing the graph’s edge weights.

3. Define a function to add edges, which takes two vertices and the edge weight as inputs, appending each edge as a tuple in the graph's edge list.
4. Implement a find function to determine the set (or component) of a vertex. This function checks if the vertex is its own parent; if not, it recursively traces up the tree to find the root, applying path compression to speed up future lookups.
5. Define a union function to connect two sets by rank. If one set's rank is lower, it becomes a child of the higher-ranked set. If ranks are equal, one set becomes the child of the other, and the rank of the new parent is incremented.
6. To build the MST, first sort the list of edges by weight in ascending order.
7. Initialize arrays for parent and rank. Each vertex is initially its own parent with a rank of 0.
8. For each edge, find the roots of its endpoints using the find function. If the roots are different, add the edge to the MST result list and connect the sets by calling the union function.
9. After processing all edges, the MST result list contains all edges of the MST. Print each edge and its weight in the format "Edge Weight."

CODE :

1. **Prim's Algorithm**

```
import sys

class Graph():

  def __init__(self, vertices):

    self.V = vertices

    self.graph = [[0] * vertices for _ in range(vertices)]

  def printMST(self, parent):

    print("\nEdge \tWeight")
```

```
        for i in range(1, self.V):
            print(f"{parent[i]} - {i} \t {self.graph[i][parent[i]]}")
    def primMST(self):
        key, parent = [sys.maxsize] * self.V, [-1] * self.V
        key[0], mstSet = 0, [False] * self.V
        for _ in range(self.V):
            u = min((key[i], i) for i in range(self.V) if not mstSet[i])[1]
            mstSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] and not mstSet[v] and key[v] > self.graph[u][v]:
                    key[v], parent[v] = self.graph[u][v], u
            self.printMST(parent)

def input_graph():
    vertices = int(input("\nEnter the number of vertices: "))
    g = Graph(vertices)
    print("Enter the adjacency matrix (row by row):")
    for i in range(vertices):
        g.graph[i] = list(map(int, input(f"Row {i + 1}: ").split()))
    g.primMST()
input_graph()
```

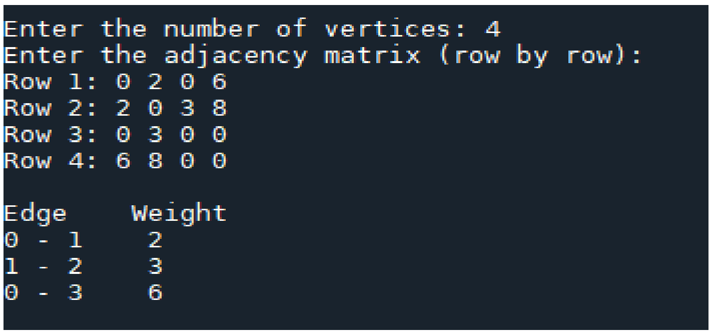
2.Kruskal’s Algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
```

```
def add_edge(self, u, v, w):
    self.graph.append((u, v, w))
def find(self, parent, i):
    if parent[i] != i:
        parent[i] = self.find(parent, parent[i])
    return parent[i]
def union(self, parent, rank, x, y):
    if rank[x] < rank[y]:
        parent[x] = y
    elif rank[x] > rank[y]:
        parent[y] = x
    else:
        parent[y] = x
        rank[x] += 1
def kruskal_mst(self):
    result = []
    self.graph.sort(key=lambda x: x[2])
    parent = list(range(self.V))
    rank = [0] * self.V
    for u, v, w in self.graph:
        x, y = self.find(parent, u), self.find(parent, v)
        if x != y:
            result.append((u, v, w))
            self.union(parent, rank, x, y)
    print("\nEdge \tWeight")
    for u, v, weight in result:
        print(f"{u} - {v}    {weight}")
def input_graph():
    vertices = int(input("\nEnter the number of vertices: "))
    g = Graph(vertices)
    g.graph = []
    print("Enter the adjacency matrix (row by row):")
    for i in range(vertices):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        for j in range(i + 1, vertices):
            if row[j] != 0:
                g.add_edge(i, j, row[j])
    g.kruskal_mst()
input_graph()
```

RESULTS :

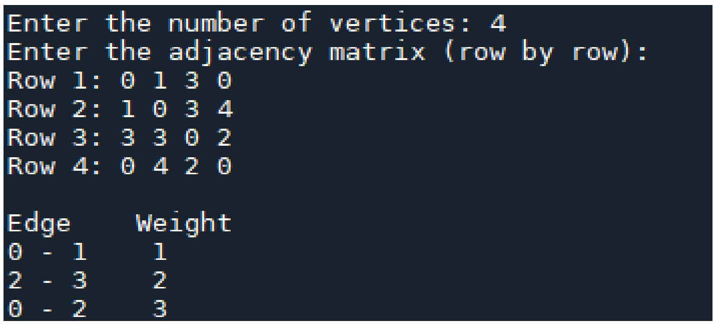
Prim’s Algorithm



```
Enter the number of vertices: 4
Enter the adjacency matrix (row by row):
Row 1: 0 2 0 6
Row 2: 2 0 3 8
Row 3: 0 3 0 0
Row 4: 6 8 0 0

Edge    Weight
0 - 1     2
1 - 2     3
0 - 3     6
```

Kruskal’s Algorithm



```
Enter the number of vertices: 4
Enter the adjacency matrix (row by row):
Row 1: 0 1 3 0
Row 2: 1 0 3 4
Row 3: 3 3 0 2
Row 4: 0 4 2 0

Edge    Weight
0 - 1     1
2 - 3     2
0 - 2     3
```

INFERENCES :

The implementations of Prim's and Kruskal's algorithms for constructing a Minimum Spanning Tree (MST) both aim to connect all vertices in a weighted, undirected graph with the minimum total edge weight. Prim’s algorithm starts with an initial vertex and gradually expands the MST by adding the minimum-weight edge that connects a new vertex, maintaining a priority on direct edge costs from the growing tree. In contrast, Kruskal’s algorithm sorts all edges by weight and incrementally builds the MST by selecting the smallest edge that connects disjoint sets of vertices, using union-find operations to prevent cycles.