23CCE201 Data Structures

Name : E Eeshwar

Roll No : CB.EN.U4CCE23012

AIM :

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

LOGIC :

- To implement various schemes to obtain minimum spanning trees for weighted undirected graphs, there are two main methods: Prim's Method and Kruskal's Method.
- In Prim's Method, we aim to find the subset of edges, which connects all vertices within a graph with the minimum total edge weight, without the formation of a cycle between vertices. Three arrays are used: a "key" array for tracking the minimum weight edge that connects the two vertices, a "parent" array for storing parent vertices for each vertex in the spanning tree and "MST set" array which tracks the included vertices to avoid processing them again.
- In Kruskal's Method, we aim to form a spanning tree without the formation of any cycles. In this method, we from an adjacency matrix with each cell containing the weight between two selected vertices. This matrix is then converted as an edge list, and then sorted by weight. Then two sets are merged if any of the two vertices are not already connected.

ALGORITHM :

- **Prim's Method Algorithm :**
    1. Get input from the user for the number of vertices, information about the vertices, weights and its neighbours of the graph.

2. Initialize three arrays – “key”, “parent” and “mst_set”. Initialize elements of ‘key’ array as infinity except the first element, ‘parent’ array elements as -1 and ‘mst_set’ array elements as False.
3. In the main loop, choose a vertex with the smallest key, which is not available in the mst_set array, and add it to the mst_set array.
4. For each vertex adjacent to the selected vertex, update key value and set parent as selected vertex, if there is an edge between the two vertices, if the other vertex is not available in the mst_set array and check if the edge weight is less than the current key value.
5. Continue the process until all vertices are included in the graph.
6. Display the resultant edges which make up the minimum spanning tree.

- **Kruskal’s Method :**
  1. Get input for the number of vertices and their adjacency matrix from the user.
  2. If there exists an edge between two vertices, add them to the edge list.
  3. Sort the edge list according to their respective weights.
  4. Create ‘parent’ and ‘rank’ arrays to manage connected components of the graph.
  5. Check whether a vertex is available on any other set of the edge list. If yes, then add this edge to the minimum spanning tree and merge the two sets.
  6. If the tree does not create a cycle, then add the edge to the ‘result’ list.
  7. Display the edges which create the minimum spanning tree.

CODE :

- **Prim’s Method :**

```
import sys
def print_mst(parent, graph):
  print("\nEdge \tWeight")
  for i in range(1, len(graph)):
```

```
        print(f"{parent[i]} - {i} \t {graph[i][parent[i]]}")

    def prim_mst(graph, vertices):
        key = [sys.maxsize] * vertices
        parent = [-1] * vertices
        key[0] = 0
        mst_set = [False] * vertices
        for _ in range(vertices):
            u = min((key[i], i) for i in range(vertices) if not mst_set[i])[1]
            mst_set[u] = True
            for v in range(vertices):
                if graph[u][v] and not mst_set[v] and key[v] > graph[u][v]:
                    key[v] = graph[u][v]
                    parent[v] = u
        print_mst(parent, graph)

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

- **Kruskal's Method :**

```
    def add_edge(graph, u, v, w):
        graph.append((u, v, w))

    def find(parent, i):
        if parent[i] != i:
            parent[i] = find(parent, parent[i])
        return parent[i]

    def union(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(vertices, graph):
    result = []
    graph.sort(key=lambda x: x[2])  # Sort edges by weight
    parent = list(range(vertices))  # Initialize parent array
    rank = [0] * vertices          # Initialize rank array
    for u, v, w in graph:
        x, y = find(parent, u), find(parent, v)
        if x != y:
            result.append((u, v, w))
            union(parent, rank, x, y)
    sum = 0
    print("\nEdge \tWeight")
    for u, v, weight in result:
        print(f"{u} - {v}    {weight}")
        sum = sum + weight
    print("The total edge weight : ",sum)

def input_graph():
    vertices = int(input("\nEnter the number of vertices: "))
    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:
                add_edge(graph, i, j, row[j])
    kruskal_mst(vertices, graph)

input_graph()
```

RESULTS :

- **Prim's Method :**

```
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
The total edge weight :  11
```

- **Kruskal's Method :**

```
Enter the number of vertices: 4
Enter the adjacency matrix (row by row):
Row 1: 0 2 6 8
Row 2: 2 0 0 3
Row 3: 6 0 0 7
Row 4: 8 3 7 0

Edge    Weight
0 - 1    2
1 - 3    3
0 - 2    6
The total edge weight :  11
```

INFERENCES :

- With the use of Prim's Method and Kruskal's Method, we can find a minimum spanning tree which connects all the vertices available in the graph using the minimum total edge weight. These methods also ensures that there are no cycles formed in the spanning tree. Both these methods use a greedy approach which acquires all immediate neighbouring vertices that are not already involved in the spanning tree and provides an outcome with the least edge weight possible.