Schemes for Obtaining Minimum Spanning Trees

Aim:

To understand and implement various algorithms for finding minimum spanning trees.

Logic:

Minimum spanning tree should have cycles and it is a sub graph that connects all the vertices with minimum possible cost. There are two different algorithms to find minimum spanning tree:

Prim's algorithm:

We start from single vertex adding on adjacent vertices with minimum cost of edge from the already existing minimum spanning tree.

Kruskal's Algorithm:

We start by picking out the edge of minimum cost and adding the vertices to minimum spanning tree and we progressively choose edges of next minimum cost edge that has no cycles.

Algorithm:

Prim's Algorithm :

1. Start
2. Choose any random vertex in the graph as the starting vertex.
3. Initialize a min heap with a tuple in it for the source vertex. Tuple format: (cost, vertex).
4. Initialize a set to keep track of visited vertices.
5. Pop from the min heap and store the data as current_cost and current_vertex.
6. If current_vertex is already in the visited set, move to step 5.

7. Add current_vertex to the visited set.

8. In a loop for all neighbors of current_vertex:

    8.1) Initialize a variable cost to store the edge cost between current_vertex and its neighbor.

    8.2) If the neighbor is not in the visited set, move to step 8.2.1.

        8.2.1) Push the tuple (cost, neighbor) to the min heap.

9. If the min heap is not empty, move to step 5.

10. End.

Kruskal's Algorithm :

1. Start
2. Sort all edges in non-decreasing order of their weights.
3. Initialize an empty set for the MST and a disjoint-set (union-find) structure to manage connected components.
4. Initialize a counter for the number of edges added to the MST (initially 0).
5. In a loop for each edge in the sorted edge list:

    5.1) Extract the edge and store its weight, start vertex, and end vertex as `weight`, `start`, and `end`.

    5.2) Check if `start` and `end` are in the same component using the union-find structure.

    5.3) If `start` and `end` are in different components, move to step 5.3.1.

        5.3.1) Add the edge to the MST set.

        5.3.2) Union the components of `start` and `end` in the disjoint-set structure.

        5.3.3) Increment the counter of edges in the MST by 1.

6. If the counter equals the number of vertices minus 1 (indicating a complete MST), go to step 7. Otherwise, continue to the next edge.
7. End.

Code:

Prim's algorithm:

```
import heapq


class Graph:
    def __init__(self, adjacency_matrix):
        self.graph = {}
        self.initialize_graph(adjacency_matrix)

    def initialize_graph(self, adjacency_matrix):
        for vertex_1 in adjacency_matrix:
            self.graph[vertex_1] = []
            for vertex_2, cost in adjacency_matrix[vertex_1].items():
                if cost != float('inf'):  # Skip if there's no direct edge
                    self.graph[vertex_1].append((vertex_2, cost))

    # Print the graph
    def print_graph(self):
```

```
    for vertex, edges in self.graph.items():
        print(f"{vertex} --> {edges}")

# Prim's algorithm for Minimum Spanning Tree
def prim_mst(self, start_vertex):
    mst_cost = 0
    visited = set()
    min_heap = [(0, start_vertex)]  # (cost, vertex)
    mst_edges = []

    while min_heap:
        current_cost, current_vertex = heapq.heappop(min_heap)

        if current_vertex in visited:
            continue

        visited.add(current_vertex)
        mst_cost += current_cost

        if current_cost != 0:
            mst_edges.append((current_vertex, current_cost))
```

```
            for neighbor, weight in self.graph[current_vertex]:
                if neighbor not in visited:
                    heapq.heappush(min_heap, (weight, neighbor))

        return mst_cost, mst_edges

# Example usage with adjacency matrix
if __name__ == "__main__":
    adjacency_matrix = {
        'A': {'B': 4, 'C': 2, 'D': float('inf')},
        'B': {'A': 4, 'C': 1, 'D': 7},
        'C': {'A': 2, 'B': 1, 'D': 3},
        'D': {'B': 7, 'C': 3}
    }

    new_graph = Graph(adjacency_matrix)

    print("Graph representation:")
    new_graph.print_graph()

    start_vertex = 'A'
    mst_cost, mst_edges = new_graph.prim_mst(start_vertex)
```

```
print(f"\nTotal cost of MST by Prim's Algorithm starting from '{start_vertex}': {mst_cost}")

print("Edges in the MST:", mst_edges)
```

Result:

```
[Running] python -u "C:\Users\ADMIN\AppData\Local\Temp\tempCodeRunnerFile.python"
Graph representation:
A --> [('B', 4), ('C', 2)]
B --> [('A', 4), ('C', 1), ('D', 7)]
C --> [('A', 2), ('B', 1), ('D', 3)]
D --> [('B', 7), ('C', 3)]

Total cost of MST by Prim's Algorithm starting from 'A': 6
Edges in the MST: [('C', 2), ('B', 1), ('D', 3)]

[Done] exited with code=0 in 0.163 seconds
```

Inference:

Prim's Algorithm is a method to connect all vertices in a graph with the shortest possible total distance, starting from one vertex and adding the closest connected vertices one by one. It's especially useful when there are a lot of edges in the graph.

Code:

Kruskal's Algorithm :

```
class UnionFind:
    def __init__(self, vertices):
        self.parent = {v: v for v in vertices}
        self.rank = {v: 0 for v in vertices}

    def find(self, vertex):
```

```
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])  # Path compression
        return self.parent[vertex]

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)

        if root1 != root2:
            # Union by rank
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1
            return True
        return False

class Graph:
    def __init__(self, adjacency_matrix):
        self.edges = []
        self.vertices = list(adjacency_matrix.keys())
        self.initialize_graph(adjacency_matrix)
```

```
# Initialize edges from the adjacency matrix
def initialize_graph(self, adjacency_matrix):
    for vertex_1 in adjacency_matrix:
        for vertex_2, cost in adjacency_matrix[vertex_1].items():
            if cost != float('inf') and (vertex_2, vertex_1, cost) not in self.edges:
                self.edges.append((vertex_1, vertex_2, cost))

# Print the graph edges
def print_edges(self):
    for edge in self.edges:
        print(f"{edge[0]} --({edge[2]})--> {edge[1]}")

# Kruskal's algorithm for Minimum Spanning Tree
def kruskal_mst(self):
    # Sort edges by weight
    self.edges.sort(key=lambda edge: edge[2])

    union_find = UnionFind(self.vertices)
    mst_cost = 0
    mst_edges = []

    for u, v, weight in self.edges:
        if union_find.union(u, v):
            mst_cost += weight
            mst_edges.append((u, v, weight))
```

```
        return mst_cost, mst_edges

# Example usage with adjacency matrix
if __name__ == "__main__":
    adjacency_matrix = {
        'A': {'B': 4, 'C': 2, 'D': float('inf')},
        'B': {'A': 4, 'C': 1, 'D': 7},
        'C': {'A': 2, 'B': 1, 'D': 3},
        'D': {'B': 7, 'C': 3}
    }

    new_graph = Graph(adjacency_matrix)

    print("Graph edges:")
    new_graph.print_edges()

    mst_cost, mst_edges = new_graph.kruskal_mst()

    print(f"\nTotal cost of MST by Kruskal's Algorithm: {mst_cost}")
    print("Edges in the MST:", mst_edges)
```

Result:

```
[Running] python -u "C:\Users\ADMIN\AppData\Local\Temp\tempCodeRunnerFile.python"
Graph edges:
A --(4)--> B
A --(2)--> C
B --(1)--> C
B --(7)--> D
C --(3)--> D

Total cost of MST by Kruskal's Algorithm: 6
Edges in the MST: [('B', 'C', 1), ('A', 'C', 2), ('C', 'D', 3)]

[Done] exited with code=0 in 0.086 seconds
```

Inference:

Kruskal's Algorithm is a method to connect all vertices in a graph with the shortest possible total distance by adding the smallest connections one at a time. It keeps adding edges but avoids any closed loops, so all vertices stay connected without forming cycles.