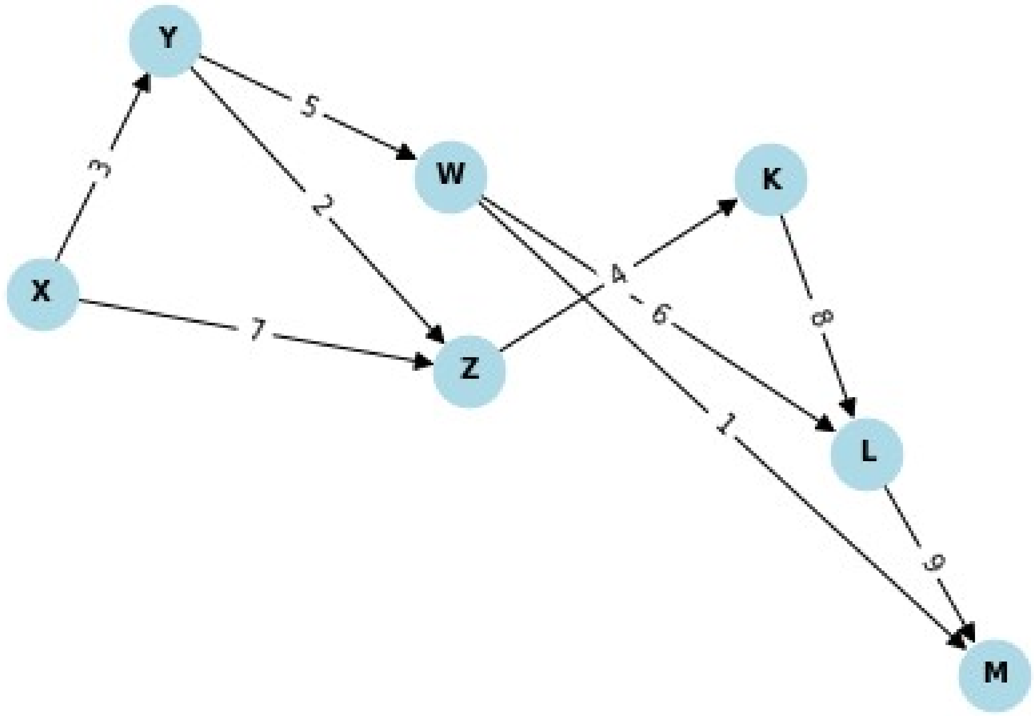
Algorithm implementation on Graphs

Rayhan Muhammed R
CB.EN.U4CCE22038

Introduction:

This introduction aims to explore the practical implementation of graph algorithms, their applications and the underlying principles that drive their effectiveness in solving diverse computations.

1. Dijkstra's Algorithm:



Objective:

To find the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

Algorithm:

```
def dijkstra(graph, source):
    visited_nodes = set()
```

```
    distances = {node: float('inf') for node in graph.nodes}
    distances[source] = 0
    priority_queue = [(0, source)]  # (distance, node)

    while priority_queue:
        current_distance, current_node = min(priority_queue)
        priority_queue.remove((current_distance, current_node))
        visited_nodes.add(current_node)

        for neighbor in graph.edges[current_node]:
            if neighbor not in visited_nodes:
                new_distance = distances[current_node] + graph.distances[(current_node, neighbor)]
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    priority_queue.append((new_distance, neighbor))

    return distances
```

Code:

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}

    def addNode(self, value):
        self.nodes.add(value)

    def addEdge(self, fromNode, toNode, distance):
        self.edges[fromNode].append(toNode)
        self.distances[(fromNode, toNode)] = distance

    def visualize(self):
        for node in self.nodes:
            neighbors = self.edges[node]
            for neighbor in neighbors:
                distance = self.distances[(node, neighbor)]
                print(f"{node} --{distance}--> {neighbor}")

def dijkstra(graph, initial):
    visited = {initial: 0}
    path = defaultdict(list)
    nodes = set(graph.nodes)
    while nodes:
```

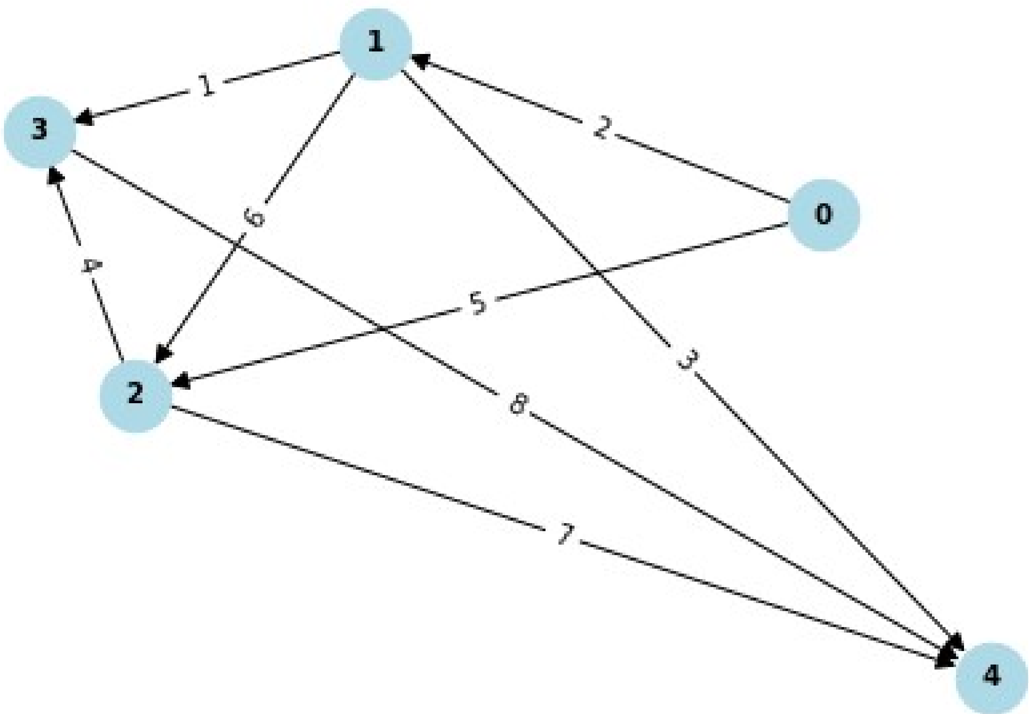
```
        minNode = None
        for node in nodes:
            if node in visited:
                if minNode is None:
                    minNode = node
                elif visited[node] < visited[minNode]:
                    minNode = node
        if minNode is None:
            break
        nodes.remove(minNode)
        currentWeight = visited[minNode]
        for edge in graph.edges[minNode]:
            weight = currentWeight + graph.distances[(minNode, edge)]
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge].append(minNode)
    return visited, path

# Create a different graph
customGraph = Graph()
customGraph.addNode("X")
customGraph.addNode("Y")
customGraph.addNode("Z")
customGraph.addNode("W")
customGraph.addNode("K")
customGraph.addNode("L")
customGraph.addNode("M")
customGraph.addEdge("X", "Y", 3)
customGraph.addEdge("X", "Z", 7)
customGraph.addEdge("Y", "Z", 2)
customGraph.addEdge("Y", "W", 5)
customGraph.addEdge("Z", "K", 4)
customGraph.addEdge("W", "L", 6)
customGraph.addEdge("W", "M", 1)
customGraph.addEdge("K", "L", 8)
customGraph.addEdge("L", "M", 9)

# Visualize the different graph
print("Graph Visualization:")
customGraph.visualize()

# Run Dijkstra's algorithm
initial_node = "X"
shortest_distances, shortest_paths = dijkstra(customGraph, initial_node)

# Display the results
print("\nShortest Distances:")
for node, distance in shortest_distances.items():
```

```
    print(f"From {initial_node} to {node}: {distance}")

print("\nShortest Paths:")
for node, path in shortest_paths.items():
    print(f"Path to {node}: {' -> '.join(path + [node])}")
```

Output:

```
Graph Visualization:
Z --4--> K
W --6--> L
W --1--> M
X --3--> Y
X --7--> Z
Y --2--> Z
Y --5--> W
K --8--> L
L --9--> M

Shortest Distances:
From X to X: 0
From X to Y: 3
From X to Z: 5
From X to W: 8
From X to K: 9
From X to L: 14
From X to M: 9

Shortest Paths:
Path to Y: X -> Y
Path to Z: X -> Y -> Z
Path to W: Y -> W
Path to K: Z -> K
Path to L: W -> L
Path to M: W -> M
```

2. Floyd's Algorithm:



Objective:

To compute the shortest paths between all pairs of vertices in a weighted graph.

Algorithm:

```
def floyd_warshall(self):
        dist_matrix = [row[:] for row in self.graph_matrix]

        for k in range(self.num_vertices):
            for i in range(self.num_vertices):
                for j in range(self.num_vertices):
                    dist_matrix[i][j] = min(
                        dist_matrix[i][j],
                        dist_matrix[i][k] + dist_matrix[k][j]
                    )

        return dist_matrix
```

Code:

```
from collections import defaultdict
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
    def __init__(self):
        self.nodes = set()
        self.node_indices = {}  # Map nodes to integers for indexing
        self.edges = defaultdict(list)
        self.distances = {}

    def addNode(self, value):
        if value not in self.node_indices:
            index = len(self.node_indices)
            self.node_indices[value] = index
            self.nodes.add(value)

    def addEdge(self, fromNode, toNode, distance):
        self.edges[fromNode].append(toNode)
        self.distances[(fromNode, toNode)] = distance

    def visualize(self):
        G = nx.DiGraph()
        for node, index in self.node_indices.items():
            G.add_node(index, label=node)
```

```
        for fromNode, neighbors in self.edges.items():
            for toNode in neighbors:
                G.add_edge(self.node_indices[fromNode], self.node_indices[toNode],
weight=self.distances[(fromNode, toNode)])

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, font_weight='bold', arrowsize=15, node_size=700,
node_color='lightblue', font_size=10)
        edge_labels = {(i, j): w['weight'] for i, j, w in G.edges(data=True)}
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        plt.show()

    def floyd_warshall(self):
        num_nodes = len(self.nodes)
        distance_matrix = [[float('inf')] * num_nodes for _ in range(num_nodes)]

        for node, index in self.node_indices.items():
            distance_matrix[index][index] = 0
            for neighbor in self.edges[node]:
                neighbor_index = self.node_indices[neighbor]
                distance_matrix[index][neighbor_index] = self.distances[(node, neighbor)]

        for k in range(num_nodes):
            for i in range(num_nodes):
                for j in range(num_nodes):
                    distance_matrix[i][j] = min(
                        distance_matrix[i][j],
                        distance_matrix[i][k] + distance_matrix[k][j]
                    )

        return distance_matrix


# Create a different graph
customGraph = Graph()
customGraph.addNode("A")
customGraph.addNode("B")
customGraph.addNode("C")
customGraph.addNode("D")
customGraph.addNode("E")
customGraph.addEdge("A", "B", 2)
customGraph.addEdge("A", "C", 5)
customGraph.addEdge("B", "C", 6)
customGraph.addEdge("B", "D", 1)
customGraph.addEdge("B", "E", 3)
customGraph.addEdge("C", "D", 4)
customGraph.addEdge("C", "E", 7)
```

```
customGraph.addEdge("D", "E", 8)

# Visualize the different graph
print("Graph Visualization:")
customGraph.visualize()

# Run Floyd's algorithm
result_matrix = customGraph.floyd_warshall()

# Display the result matrix
print("\nFloyd's Algorithm Result:")
for row in result_matrix:
    print(row)
```

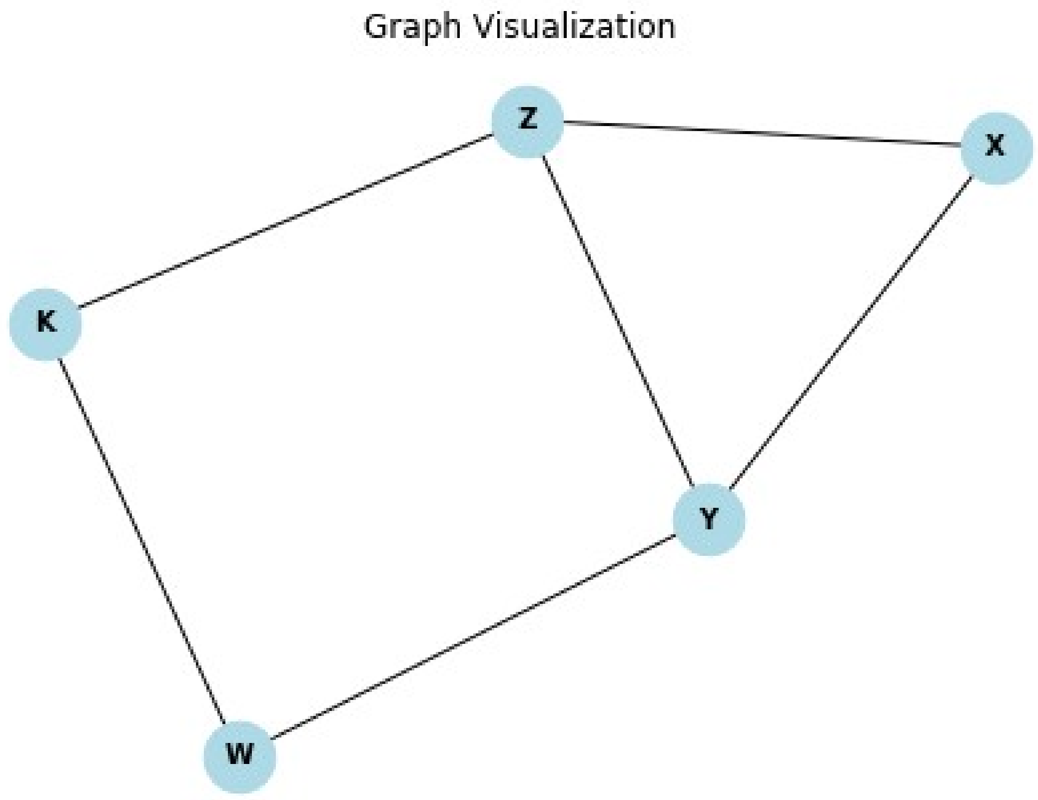
<u>Output:</u>

```
Graph Visualization:

Floyd's Algorithm Result:
[0, 2, 5, 3, 5]
[inf, 0, 6, 1, 3]
[inf, inf, 0, 4, 7]
[inf, inf, inf, 0, 8]
[inf, inf, inf, inf, 0]
```

<u>3. Breadth First Search:</u>

<u>Objective:</u>

Traverse a graph level by level, starting from a source vertex.



Algorithm:

```
function bfs(adjacency_list, start_vertex):
    visited = set()
    visited.add(start_vertex)
    queue = deque([start_vertex])
    while queue:
        current_vertex = queue.popleft()
        print(current_vertex)
        for adjacent_vertex in adjacency_list[current_vertex]:
            if adjacent_vertex not in visited:
            visited.add(adjacent_vertex)
            queue.append(adjacent_vertex)
```

Code:

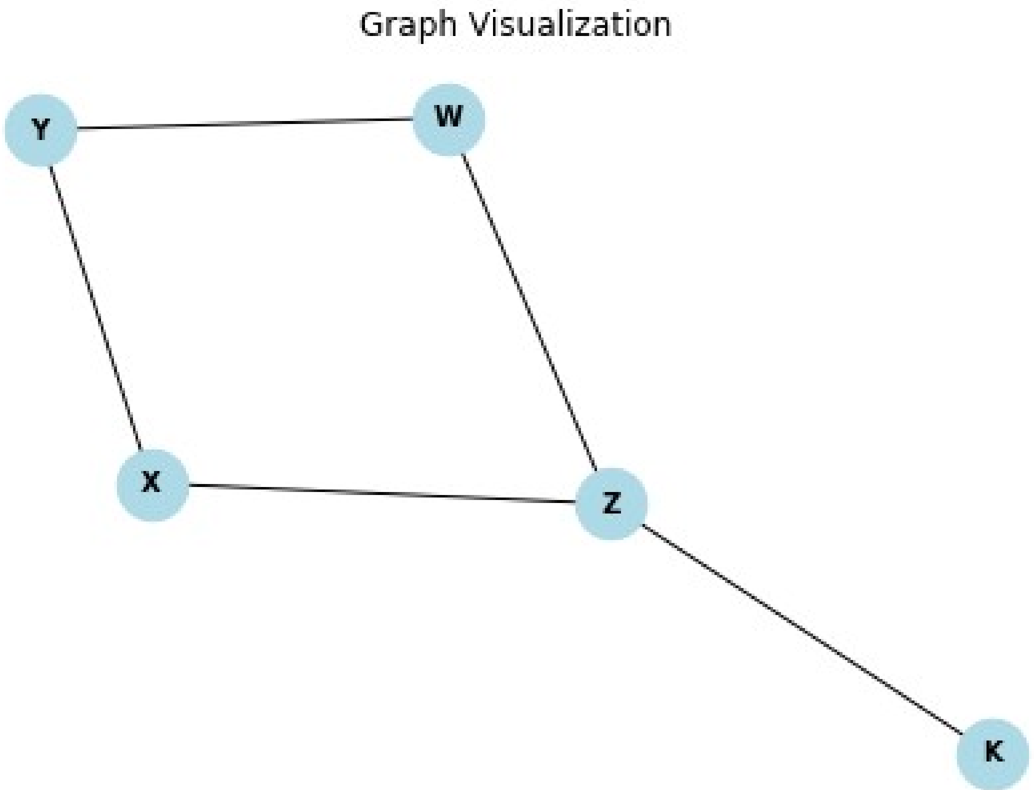
```
from collections import deque
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adjacency_list.keys():
            self.adjacency_list[vertex] = []
            return True
        return False

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.adjacency_list.keys() and vertex2 in self.adjacency_list.keys():
            self.adjacency_list[vertex1].append(vertex2)
            self.adjacency_list[vertex2].append(vertex1)
            return True
        return False

    def bfs(self, start_vertex):
        visited = set()
        visited.add(start_vertex)
        queue = deque([start_vertex])
        while queue:
            current_vertex = queue.popleft()
            for adjacent_vertex in self.adjacency_list[current_vertex]:
                if adjacent_vertex not in visited:
                    visited.add(adjacent_vertex)
```

```
            queue.append(adjacent_vertex)

# Create a different graph
new_graph = Graph()
new_graph.add_vertex("X")
new_graph.add_vertex("Y")
new_graph.add_vertex("Z")
new_graph.add_vertex("W")
new_graph.add_vertex("K")
new_graph.add_edge("X", "Y")
new_graph.add_edge("X", "Z")
new_graph.add_edge("Y", "Z")
new_graph.add_edge("Y", "W")
new_graph.add_edge("Z", "K")
new_graph.add_edge("W", "K")

# Visualize the different graph using NetworkX and Matplotlib
G = nx.Graph()
for vertex, neighbors in new_graph.adjacency_list.items():
    G.add_node(vertex)
    for neighbor in neighbors:
        G.add_edge(vertex, neighbor)

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, font_weight='bold', arrowsize=15, node_size=700,
node_color='lightblue', font_size=10)

plt.title("Graph Visualization")
plt.show()
```

Output:

```
Graph:
X : ['Y', 'Z']
Y : ['X', 'Z', 'W']
Z : ['X', 'Y', 'K']
W : ['Y', 'K']
K : ['Z', 'W']

BFS Traversal:
X
Y
Z
W
K
```

4. Depth First Search:



Objective:

Traverse a graph by exploring as far as possible along each branch before backtracking.

Algorithm:

```
def dfs(self, vertex):
        visited = set()
        traversal_path = []

        def dfs_recursive(current_vertex):
            nonlocal visited, traversal_path
            visited.add(current_vertex)
            traversal_path.append(current_vertex)

            for adjacent_vertex in
self.adjacency_list[current_vertex]:
                if adjacent_vertex not in visited:
                    dfs_recursive(adjacent_vertex)

        dfs_recursive(vertex)
        return traversal_path
```

Code:

```
from collections import deque

import networkx as nx

import matplotlib.pyplot as plt



class Graph:

    def __init__(self):

        self.adjacency_list = {}



    def add_vertex(self, vertex):

        if vertex not in self.adjacency_list.keys():
```

```
        self.adjacency_list[vertex] = []
        return True
    return False

def print_graph(self):
    for vertex in self.adjacency_list:
        print(vertex, ":", self.adjacency_list[vertex])

def add_edge(self, vertex1, vertex2):
    if vertex1 in self.adjacency_list.keys() and vertex2 in self.adjacency_list.keys():
        self.adjacency_list[vertex1].append(vertex2)
        self.adjacency_list[vertex2].append(vertex1)
        return True
    return False

def remove_edge(self, vertex1, vertex2):
    if vertex1 in self.adjacency_list.keys() and vertex2 in self.adjacency_list.keys():
        try:
            self.adjacency_list[vertex1].remove(vertex2)
            self.adjacency_list[vertex2].remove(vertex1)
        except ValueError:
            pass
        return True
    return False
```

```python
    def remove_vertex(self, vertex):
        if vertex in self.adjacency_list.keys():
            for other_vertex in self.adjacency_list[vertex]:
                self.adjacency_list[other_vertex].remove(vertex)
            del self.adjacency_list[vertex]
            return True
        return False

    def dfs(self, vertex):
        visited = set()
        stack = [vertex]
        traversal_path = []  # Store the traversal path
        while stack:
            current_vertex = stack.pop()
            if current_vertex not in visited:
                traversal_path.append(current_vertex)
                visited.add(current_vertex)

            for adjacent_vertex in self.adjacency_list[current_vertex]:
                if adjacent_vertex not in visited:
                    stack.append(adjacent_vertex)

        return traversal_path

# Create a different graph
```

```
new_graph = Graph()
new_graph.add_vertex("X")
new_graph.add_vertex("Y")
new_graph.add_vertex("Z")
new_graph.add_vertex("W")
new_graph.add_vertex("K")
new_graph.add_edge("X", "Y")
new_graph.add_edge("X", "Z")
new_graph.add_edge("Y", "W")
new_graph.add_edge("Z", "W")
new_graph.add_edge("Z", "K")

# Print the graph structure
print("Graph Structure:")
new_graph.print_graph()
dfs_path = new_graph.dfs("X")
print("\nDFS Process:")
for vertex in dfs_path:
    print(vertex)
G = nx.Graph()
for vertex, neighbors in new_graph.adjacency_list.items():
    G.add_node(vertex)
    for neighbor in neighbors:
        G.add_edge(vertex, neighbor)
pos = nx.spring_layout(G)
```

nx.draw(G, pos, with_labels=True, font_weight='bold', arrowsize=15, node_size=700, node_color='lightblue', font_size=10)

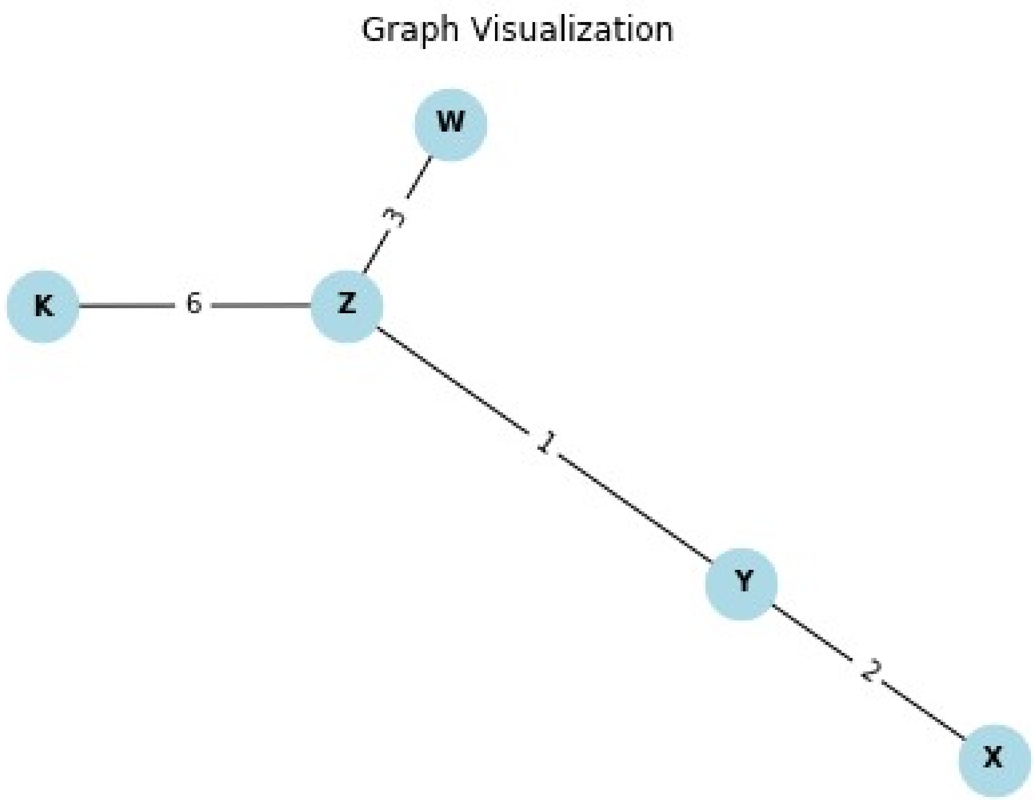
plt.title("Graph Visualization")

plt.show()

<u>Output:</u>

```
Graph Structure:
X : ['Y', 'Z']
Y : ['X', 'W']
Z : ['X', 'W', 'K']
W : ['Y', 'Z']
K : ['Z']

DFS Process:
X
Z
K
W
Y
```

<u>5. Prim’s Algorithm:</u>



Objective:

Find the minimum spanning tree (MST) of a connected, undirected graph.

Algorithm:

```
def prim(graph):
    mst = []
    visited = set()
    start_node = list(graph.keys())[0]
    visited.add(start_node)

    while len(mst) < len(graph) - 1:
        u, v, weight = find_min_weight_edge(graph, visited)
        mst.append((u, v, weight))
        visited.add(v)

    return mst
```

Code:

```
import sys

import networkx as nx

import matplotlib.pyplot as plt



class Graph:

    def __init__(self, vertexNum, edges, nodes):

        self.edges = edges

        self.nodes = nodes

        self.vertexNum = vertexNum

        self.MST = []



    def printSolution(self):

        print("Minimum Spanning Tree (MST):")
```

```
        print("Edge : Weight")
        for s, d, w in self.MST:
            print("%s -> %s: %s" % (s, d, w))

    def primsAlgo(self):
        visited = [0] * self.vertexNum
        edgeNum = 0
        visited[0] = True

        while edgeNum < self.vertexNum - 1:
            min_weight = sys.maxsize
            s, d = 0, 0

            for i in range(self.vertexNum):
                if visited[i]:
                    for j in range(self.vertexNum):
                        if not visited[j] and self.edges[i][j] and min_weight > self.edges[i][j]:
                            min_weight = self.edges[i][j]
                            s, d = i, j

            self.MST.append([self.nodes[s], self.nodes[d], self.edges[s][d]])
            visited[d] = True
            edgeNum += 1

            print(f"Selected edge: {self.nodes[s]} - {self.nodes[d]} ({self.edges[s][d]})")
```

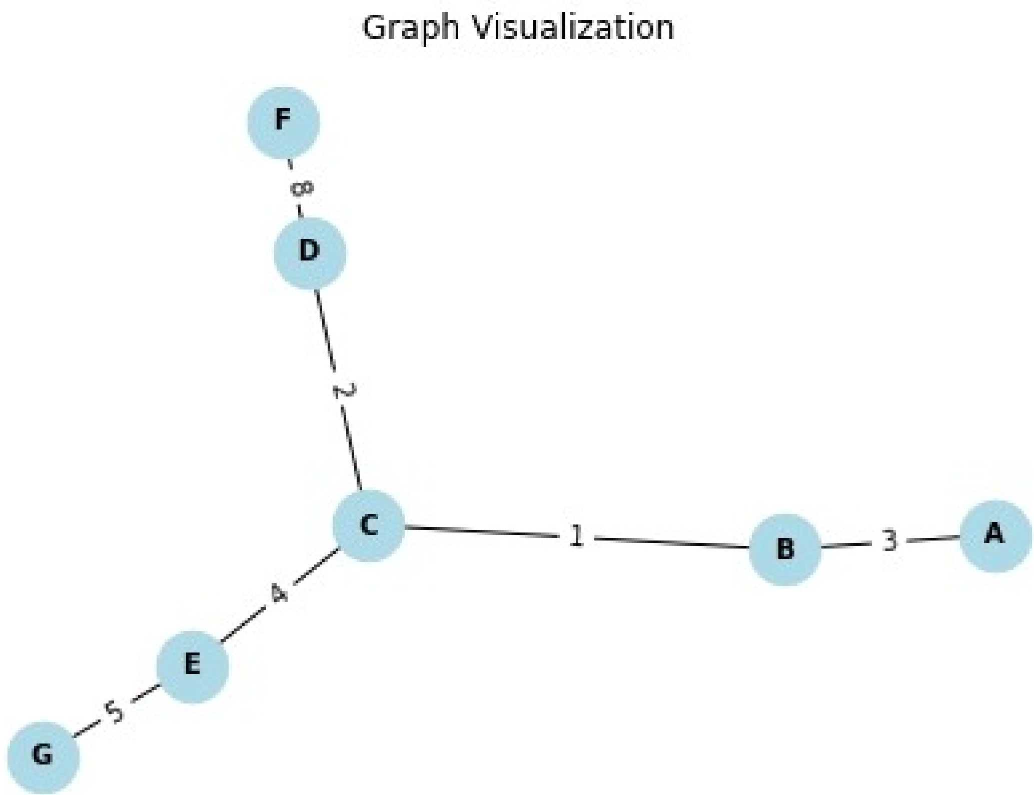
```
            print(f"Visited nodes: {self.get_visited_nodes(visited)}")


        self.printSolution()

        self.plot_graph()


    def get_visited_nodes(self, visited):

        return [self.nodes[i] for i in range(self.vertexNum) if visited[i]]


    def plot_graph(self):

        G = nx.Graph()

        for node in self.nodes:

            G.add_node(node)

        for s, d, w in self.MST:

            G.add_edge(s, d, weight=w)


        pos = nx.spring_layout(G)

        nx.draw(G, pos, with_labels=True, font_weight='bold', arrowsize=15, node_size=700,
node_color='lightblue', font_size=10)

        edge_labels = {(s, d): w for s, d, w in self.MST}

        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)


        plt.title("Graph Visualization")

        plt.show()




different_edges = [
```

```
    [0, 2, 4, 0, 0],

    [2, 0, 1, 5, 0],

    [4, 1, 0, 3, 6],

    [0, 5, 3, 0, 8],

    [0, 0, 6, 8, 0]

]


different_nodes = ["X", "Y", "Z", "W", "K"]

different_g = Graph(5, different_edges, different_nodes)

different_g.primsAlgo()
```

Output:

```
Selected edge: X - Y (2)
Visited nodes: ['X', 'Y']
Selected edge: Y - Z (1)
Visited nodes: ['X', 'Y', 'Z']
Selected edge: Z - W (3)
Visited nodes: ['X', 'Y', 'Z', 'W']
Selected edge: Z - K (6)
Visited nodes: ['X', 'Y', 'Z', 'W', 'K']
Minimum Spanning Tree (MST):
Edge : Weight
X -> Y: 2
Y -> Z: 1
Z -> W: 3
Z -> K: 6
```

6. Kruskal’s Algorithm:



Objective:

Find the minimum spanning tree (MST) of a connected, undirected graph.

Algorithm:

```
def kruskalAlgo(self):
    edge_count, mst_count = 0, 0
```

```
    while mst_count < self.V - 1:
        source, destination, weight = self.graph[edge_count]
        set_source, set_destination = ds.find(source), ds.find(destination)

        if set_source != set_destination:
            mst_count += 1
            self.MST.append([source, destination, weight])
            ds.union(set_source, set_destination)

        edge_count += 1
```

<u>Code:</u>

```
import networkx as nx
import matplotlib.pyplot as plt

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

    def addEdge(self, s, d, w):
        self.graph.append([s, d, w])

    def addNode(self, value):
        self.nodes.append(value)

    def printSolution(self, s, d, w):
        print("Minimum Spanning Tree (MST):")
        for s, d, w in self.MST:
            print("%s - %s: %s" % (s, d, w))
```

```
def kruskalAlgo(self):
    i, e = 0, 0
    ds = DisjointSet(self.nodes)
    self.graph = sorted(self.graph, key=lambda item: item[2])

    print("Edges sorted by weight:")
    for edge in self.graph:
        print(f"{edge[0]} - {edge[1]}: {edge[2]}")

    while e < self.V - 1:
        s, d, w = self.graph[i]
        i += 1
        x = ds.find(s)
        y = ds.find(d)

        print(f"Checking edge: {s} - {d} ({w})")

        if x != y:
            e += 1
            self.MST.append([s, d, w])
            ds.union(x, y)
            print(f"Selected edge: {s} - {d} ({w}), Added to MST")
            print(f"Sets after adding edge: {ds.get_sets()}")

    self.printSolution(s, d, w)
    self.plot_graph()

def plot_graph(self):
    G = nx.Graph()
    for node in self.nodes:
        G.add_node(node)
```

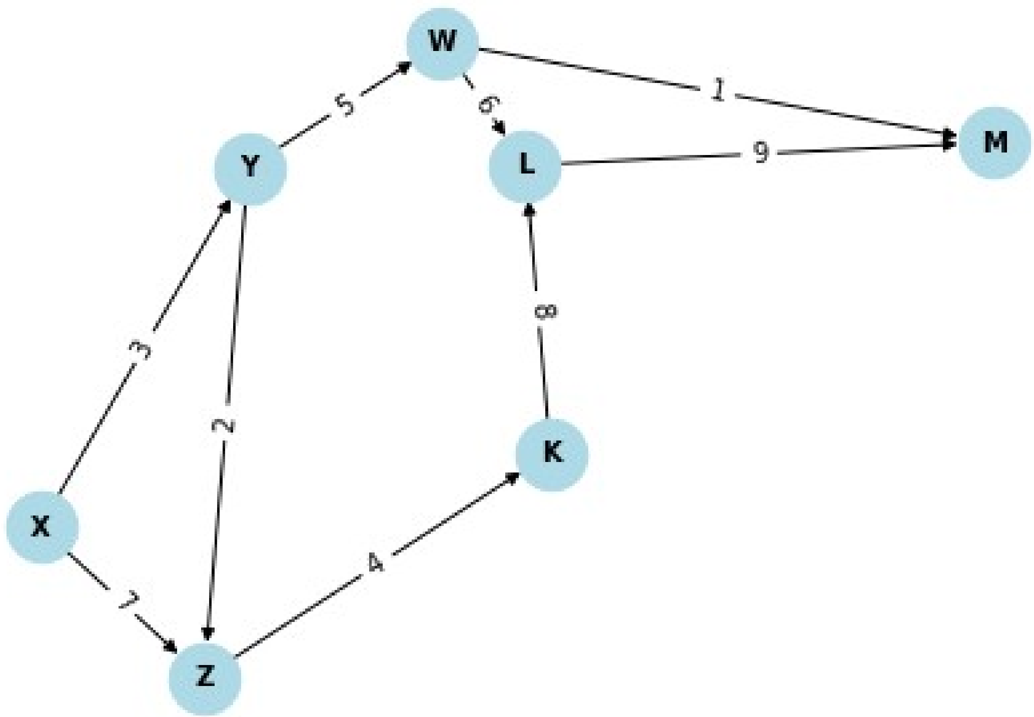
```
        for s, d, w in self.MST:
            G.add_edge(s, d, weight=w)

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, font_weight='bold', arrowsize=15, node_size=700,
node_color='lightblue', font_size=10)
        edge_labels = {(s, d): w for s, d, w in self.MST}
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        plt.title("Graph Visualization")
        plt.show()


class DisjointSet:
    def __init__(self, nodes):
        self.parent = {}
        for node in nodes:
            self.parent[node] = node

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

    def union(self, x, y):
        x_set = self.find(x)
        y_set = self.find(y)
        self.parent[y_set] = x_set

    def get_sets(self):
        sets = {}
        for node in self.parent:
```

```
            root = self.find(node)
            if root in sets:
                sets[root].append(node)
            else:
                sets[root] = [node]
        return sets


# Create a new graph
new_g = Graph(7)
new_g.addNode("A")
new_g.addNode("B")
new_g.addNode("C")
new_g.addNode("D")
new_g.addNode("E")
new_g.addNode("F")
new_g.addNode("G")

new_g.addEdge("A", "B", 3)
new_g.addEdge("A", "C", 5)
new_g.addEdge("B", "C", 1)
new_g.addEdge("B", "D", 6)
new_g.addEdge("C", "D", 2)
new_g.addEdge("C", "E", 4)
new_g.addEdge("D", "E", 7)
new_g.addEdge("D", "F", 8)
new_g.addEdge("E", "F", 9)
new_g.addEdge("E", "G", 5)
new_g.addEdge("F", "G", 10)

new_g.kruskalAlgo()
```

Output:

```
Edges sorted by weight:
B - C: 1
C - D: 2
A - B: 3
C - E: 4
A - C: 5
E - G: 5
B - D: 6
D - E: 7
D - F: 8
E - F: 9
F - G: 10
Checking edge: B - C (1)
Selected edge: B - C (1), Added to MST
Sets after adding edge: {'A': ['A'], 'B': ['B', 'C'], 'D': ['D'], 'E': ['E'], 'F': ['F'], 'G': ['G']}
Checking edge: C - D (2)
Selected edge: C - D (2), Added to MST
Sets after adding edge: {'A': ['A'], 'B': ['B', 'C', 'D'], 'E': ['E'], 'F': ['F'], 'G': ['G']}
Checking edge: A - B (3)
Selected edge: A - B (3), Added to MST
Sets after adding edge: {'A': ['A', 'B', 'C', 'D'], 'E': ['E'], 'F': ['F'], 'G': ['G']}
Checking edge: C - E (4)
Selected edge: C - E (4), Added to MST
Sets after adding edge: {'A': ['A', 'B', 'C', 'D', 'E'], 'F': ['F'], 'G': ['G']}
Checking edge: A - C (5)
Checking edge: E - G (5)
Selected edge: E - G (5), Added to MST
Sets after adding edge: {'A': ['A', 'B', 'C', 'D', 'E', 'G'], 'F': ['F']}
Checking edge: B - D (6)
Checking edge: D - E (7)
Checking edge: D - F (8)
Selected edge: D - F (8), Added to MST
Sets after adding edge: {'A': ['A', 'B', 'C', 'D', 'E', 'F', 'G']}
Minimum Spanning Tree (MST):
B - C: 1
C - D: 2
A - B: 3
C - E: 4
E - G: 5
D - F: 8
```

7. Bellman Ford



Objective:

To find the shortest paths in a graph from a source vertex, handling negative weights and detecting negative cycles.

Algorithm:

```
def bellman_ford(graph, num_vertices, source):
    dist = [float('inf')] * num_vertices
    dist[source] = 0

    for _ in range(num_vertices - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    negative_cycle = any(dist[u] != float('inf') and dist[u] + w < dist[v] for u, v, w in graph)

    return dist, negative_cycle
```

Code:

```
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = {}
        self.nodes = set()

    def add_edge(self, s, d, w):
        if s not in self.graph:
            self.graph[s] = []
        self.graph[s].append((d, w))
        self.nodes.update([s, d])

    def addNode(self, value):
        self.nodes.add(value)

    def visualize_graph(self):
        G = nx.DiGraph()
        for s in self.graph:
            for d, w in self.graph[s]:
                G.add_edge(s, d, weight=w)

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700,
node_color='lightblue', font_size=10)
        edge_labels = {(s, d): w for s in self.graph for d, w in self.graph[s]}
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
        plt.show()

    def bellmanFord(self, src):
        distances = {node: float("Inf") for node in self.nodes}
        distances[src] = 0

        for _ in range(self.V - 1):
            for s in self.graph:
                for d, w in self.graph[s]:
                    if distances[s] != float("Inf") and distances[s] + w < distances[d]:
                        distances[d] = distances[s] + w

        for s in self.graph:
            for d, w in self.graph[s]:
                if distances[s] != float("Inf") and distances[s] + w < distances[d]:
                    print("Graph contains a negative cycle")
```

```
                return

        self.print_solution(distances)

    def print_solution(self, distances):
        print("\nVertex Distance from Source:")
        for node, distance in distances.items():
            print(f"{node}: {distance}" if distance != float("Inf") else f"{node}: ∞")

# Example usage with a new graph
g = Graph(6)
g.addNode("X")
g.addNode("Y")
g.addNode("Z")
g.addNode("W")
g.addNode("K")
g.addNode("L")

g.add_edge("X", "Y", 3)
g.add_edge("X", "Z", 7)
g.add_edge("Y", "Z", 2)
g.add_edge("Y", "W", 5)
g.add_edge("Z", "K", 4)
g.add_edge("W", "L", 6)
g.add_edge("W", "M", 1)
g.add_edge("K", "L", 8)
g.add_edge("L", "M", 9)

# Visualize the new graph
print("Graph Visualization:")
g.visualize_graph()

# Perform Bellman-Ford algorithm
print("\nBellman-Ford Algorithm:")
g.bellmanFord("X")
```

<u>Output:</u>

```
Graph Visualization:

Bellman-Ford Algorithm:

Vertex Distance from Source:
Z: 5
W: 8
X: 0
Y: 3
K: 9
L: 14
M: 9
```

https://github.com/Rayyhhaann/19CCE202-Algorithms-on-graphs.git