

ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLGY

SUBMITTED BY: SAYYED ZAIN UL ABIDIN SHAH

SUBMITTED TO: MR JAMAL ABDUL AHAD

SECTION: C

ROLL NO: 12387

ASSIGNMENT:5

SUBJECT: DATA STRUCTURES AND ALGORTHMS

Question :1

import networkx as nx

import matplotlib.pyplot as plt

# Create an empty graph

G = nx.Graph()

# Add vertices

G.add\_node(1)

G.add\_node(2)

G.add\_node(3)

# Add edges

G.add\_edge(1, 2)

G.add\_edge(2, 3)

# Remove vertices and edges

G.remove\_node(3)

G.remove\_edge(1, 2)

# Visualize the graph

nx.draw(G, with\_labels=True)

plt.show()

Question 2

# Depth-First Search (DFS) - Recursive

def dfs\_recursive(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

for neighbor in graph[start]:

if neighbor not in visited:

dfs\_recursive(graph, neighbor, visited)

return visited

# Breadth-First Search (BFS)

from collections import deque

def bfs(graph, start):

visited, queue = set(), deque([start])

visited.add(start)

while queue:

vertex = queue.popleft()

for neighbor in graph[vertex]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

return visited

# Applications:

# - Finding connected components, identifying cycles, determining if bipartite

Question : 3

[11:53 pm, 28/01/2024] Wajdan Uni SE: # Iterative DFS for finding paths between vertices

def iterative\_dfs(graph, start, end):

stack = [(start, [start])]

while stack:

(vertex, path) = stack.pop()

for next in set(graph[vertex]) - set(path):

if next == end:

yield path + [next]

else:

stack.append((next, path + [next]))

# Dijkstra's Algorithm for finding shortest paths

import heapq

def dijkstra(graph, start):

distances = {vertex: float('infinity') for vertex in graph}

distances[start] = 0

queue = [(0, start)]

while queue:

current\_distance, current\_vertex = heapq.heappop(queue)

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(queue, (distance, neighbor))

return distances

# Applications:

# - Finding shortest routes on maps, determining efficient task sequences in projects

[11:53 pm, 28/01/2024] Wajdan Uni SE: # Calculate graph properties

# Degree of vertices

degrees = dict(G.degree())

# Density of graph

density = nx.density(G)

# Diameter of graph

diameter = nx.diameter(G)

# Analyze network characteristics

# You can use the calculated properties to analyze the structure and patterns of different networks.

Question :4

import heapq

def prim(graph):

mst = []

visited = set()

start\_vertex = next(iter(graph))

visited.add(start\_vertex)

edges = [(weight, start\_vertex, to) for to, weight in graph[start\_vertex]]

heapq.heapify(edges)

while edges:

weight, frm, to = heapq.heappop(edges)

if to not in visited:

visited.add(to)

mst.append((frm, to, weight))

for next\_to, next\_weight in graph[to]:

if next\_to not in visited:

heapq.heappush(edges, (next\_weight, to, next\_to))

return mst

# Example usage:

graph = {

'A': [('B', 2), ('C', 3)],

'B': [('A', 2), ('C', 1), ('D', 1)],

'C': [('A', 3), ('B', 1), ('D', 2)],

'D': [('B', 1), ('C', 2)]

}

minimum\_spanning\_tree = prim(graph)

print(minimum\_spanning\_tree)

Question :4

def kruskal(graph):

mst = []

edges = [(weight, frm, to) for frm in graph for to, weight in graph[frm]]

edges.sort()

parent = {v: v for v in graph}

def find(node):

if parent[node] != node:

parent[node] = find(parent[node])

return parent[node]

def union(v1, v2):

root1 = find(v1)

root2 = find(v2)

parent[root2] = root1

for weight, frm, to in edges:

if find(frm) != find(to):

mst.append((frm, to, weight))

union(frm, to)

return mst

# Example usage:

graph = {

'A': [('B', 2), ('C', 3)],

'B': [('A', 2), ('C', 1), ('D', 1)],

'C': [('A', 3), ('B', 1), ('D', 2)],

'D': [('B', 1), ('C', 2)]

}

minimum\_spanning\_tree = kruskal(graph)

print(minimum\_spanning\_tree)