IT1160-DiscreteMathematics

LabSheet 08

Part A

Graphs

A small airline company maps its flight routes as a directed graph. Cities are vertices, and flights are directed edges. The company wants to analyze this graph for optimization.

Cities (Vertices): A, B, C, D, E
Flights (Edges): (A → B), (B → C), (C → D), (C → E), (E → D), (C → B)

01) Write a Python program to,

a) Represent this graph as both an adjacency list and an adjacency matrix

b) Display the graph using a visualization

02) The airline wants to analyze the degree of connectivity of each city. Write a Python program to,

a) Find the in-degree and out-degree of each city

b) Check if there are any self-loops

03) The airline wants to check if it's possible to fly from City A to City D, and also find all simple paths from A to D. Write a Python program to,

a) Check if D is reachable from A
b) Print all simple paths from A to D
c) Check for any cycles in the graph

04) A travel app wants to discover the shortest flight route (in terms of stops) from A to other cities. Implement Breadth-First Search (BFS) starting from A. Find,

a) The order of visited nodes
b) The BFS tree structure

05) The network admin wants to explore all possible paths deeply from City A. Implement Depth-First Search (DFS) from vertex A, and,

a) Record discovery and finishing times of each city

b) Display the DFS forest if disconnected

Trees

06) A school's student ranking list (based on GPA) got jumbled due to a system glitch. Use the HEAPIFY algorithm to correct the ranking starting from a specific position in the heap.

GPA_list = [4.0, 3.8, 2.5, 3.2, 3.9, 2.0]

07) You're given a binary tree represented as a list. Write a function to check if it is a full binary tree (i.e., every node has 0 or 2 children).

Binary Tree (as array) - [1, 2, 3, 4, 5, None, None]

08) A school's student ranking list (based on GPA) got jumbled due to a system glitch. Use the HEAPIFY algorithm to correct the ranking starting from a specific position in the heap.

GPA_list = [4.0, 3.8, 2.5, 3.2, 3.9, 2.0]

09) The government of Tech Island is planning to set up a communication network between four towns: A, B, C, and D. Since the island has limited resources, the goal is to connect all towns using underground fiber-optic cables in a way that:

- All towns are connected
- The total cost is minimized
- There is no cyclic connection (no redundant paths)

The engineering team has conducted a survey and estimated the following costs (in millions) to lay cables between the towns:

From	To	Cost
A	B	2
A	D	6
B	C	3
B	D	8
C	D	5

As the lead engineer for this project, you are assigned to design the most cost-efficient plan to connect the towns using a Minimum Spanning Tree (MST) approach.

a) Model the problem as a weighted undirected graph
b) Find the Minimum Spanning Tree using Prim's algorithm
c) Display the selected cable connections and their costs
d) Show the total installation cost

Part B

You are part of the Intergalactic Navigation Bureau, mapping routes between star systems in the Milky Way. Each star system is a node, and travel routes are directed edges. Your mission begins by mapping the network.

Star Systems: Sol, Alpha, Vega, Sirius, Betelgeuse
Travel Routes: (Sol → Alpha), (Alpha → Vega), (Vega → Sirius), (Vega → Betelgeuse), (Betelgeuse → Sirius), (Vega → Alpha)

06) Write a Python program to:

a) Build the star map as a directed graph

b) Display the adjacency list and adjacency matrix

c) Visualize the network using a space-themed style

07) To avoid space traffic congestion, mission control wants to analyze how busy each starport is. Write a Python program to,

a) Compute the in-degree and out-degree of each star system

b) Identify any self-loops (loops in wormholes)

08) You are plotting a mission from Earth's system (Sol) to reach Sirius. Mission control needs all possible safe routes. Write a Python program to,

a) Check if Sirius is reachable from Sol

b) List all simple paths from Sol to Sirius

c) Identify if there are any cycles in the network

09) Your exploration drone performs a breadth-first scan from Sol to map the galaxy with the least number of fuel jumps. Implement Breadth-First Search (BFS) from Sol. Find,

- The visit order
- The BFS tree

10) To analyze deep space mysteries, the crew uses Depth-First Search to explore all reachable star systems and record discovery timestamps. Write a recursive DFS function that,

a) Records discovery and finishing times of each star system

b) Prints the DFS tree (or forest)

11) Given a nearly max-heap GPA list [4.0, 3.8, 2.5, 3.2, 3.9, 2.0], apply HEAPIFY from index 1 to fix the heap.

12) Given a list of product ratings [4.9, 4.7, 4.5, 3.8, 4.8], apply HEAPIFY after updating the rating at index 3 to maintain heap structure.

SmartLogi Inc., a logistics and supply chain company, is setting up a centralized delivery network to connect its four key distribution centers located in cities P, Q, R, and S. The objective is to ensure:

- All centers are interconnected
- The overall road construction cost is minimized
- There are no redundant road paths (i.e., no loops)

After a ground survey, the estimated construction costs (in millions) for possible direct roads between centers are:

From	To	Cost
P	Q	4
P	S	7
Q	R	2
Q	S	6
R	S	3

As the system analyst responsible for route optimization, perform the following:

13. Model the above scenario as a weighted undirected graph using an appropriate Python library
14. Use Prim's Algorithm to compute the Minimum Spanning Tree (MST)
15. Display the selected route connections and costs, and print the total cost

Python Collections Module

from collections import namedtuple, deque, Counter, OrderedDict, defaultdict, ChainMap

namedtuple - Factory function for creating tuple subclasses with named fields

Point = namedtuple('Point', 'x y')

p = Point(1, 2)

print(p.x, p.y) # Output: 1 2

deque - List-like container with fast appends and pops on either end

d = deque([1, 2, 3])

d.append(4)

d.appendleft(0)

print(d) # Output: deque([0, 1, 2, 3, 4])

Counter - Dict subclass for counting hashable objects

cnt = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'apple'])

print(cnt) # Output: Counter({'apple': 3, 'banana': 2, 'orange': 1})

OrderedDict - Dict subclass that remembers the order entries were added

od = OrderedDict()

od['a'] = 1

od['b'] = 2

od['c'] = 3

print(od) # Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)])

defaultdict - Dict subclass that calls a factory function for missing keys

dd = defaultdict(int)

```python
dd['a'] += 1
print(dd['a'])  # Output: 1
print(dd['b'])  # Output: 0 (default value since 'b' doesn't exist)
```

ChainMap - Groups multiple dictionaries into a single view

```python
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
cm = ChainMap(dict1, dict2)
print(cm['b'])  # Output: 2 (from dict1, as it comes first)
print(cm['c'])  # Output: 4 (from dict2)
```

Python networkx Library

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

Create an empty Graph (undirected)

```python
G = nx.Graph()
G.add_node(1)
G.add_nodes_from([2, 3])
G.add_edge(1, 2)
G.add_edges_from([(2, 3), (3, 1)])

print("Nodes:", G.nodes())  # Output: Nodes: [1, 2, 3]
print("Edges:", G.edges())  # Output: Edges: [(1, 2), (1, 3), (2, 3)]
```

Create a Directed Graph

```
DG = nx.DiGraph()
DG.add_edges_from([(1, 2), (2, 3)])
print("Directed Edges:", DG.edges())  # Output: [(1, 2), (2, 3)]
```

Add node and edge attributes

```
G.nodes[1]['label'] = 'Start'
G.edges[1, 2]['weight'] = 4.2
print(G.nodes(data=True))  # Output includes attributes
print(G.edges(data=True))  # Output includes edge weights
```

Visualize the Graph

```
nx.draw(G, with_labels=True)
plt.show()
```

Shortest Path

```
shortest = nx.shortest_path(G, source=1, target=3)
print("Shortest path from 1 to 3:", shortest)  # Output: [1, 3]
```

Degree and Centrality Measures

```
degree_dict = dict(G.degree())
print("Degrees:", degree_dict)
centrality = nx.degree_centrality(G)
print("Degree Centrality:", centrality)
```

Cycle Detection

```
has_cycle = nx.cycle_basis(G)
print("Cycles in the graph:", has_cycle)  # Output: [[1, 2, 3]]
```

Connected Components

```
components = list(nx.connected_components(G))
print("Connected Components:", components)  # Output: [{1, 2, 3}]
```

Graph from Edgelist or Adjacency List

```
edge_list = [(1, 2), (2, 3), (3, 4)]
G_from_edges = nx.from_edgelist(edge_list)
print("Graph from edge list:", G_from_edges.edges())
```

Convert Graph to Adjacency Matrix

```
adj_matrix = nx.adjacency_matrix(G).todense()
print("Adjacency Matrix:\n", adj_matrix)
```

PageRank (for Directed Graphs)

```
pagerank = nx.pagerank(DG)
print("PageRank:", pagerank)
```

Clustering Coefficient

```
clustering = nx.clustering(G)
print("Clustering Coefficient:", clustering)
```

Minimum Spanning Tree (for weighted graphs)

```
MST = nx.minimum_spanning_tree(G)
print("Minimum Spanning Tree Edges:", MST.edges())
```

Community Detection (Greedy Modularity)

```python
from networkx.algorithms.community import greedy_modularity_communities
communities = greedy_modularity_communities(G)
print("Communities:", [list(c) for c in communities])
```

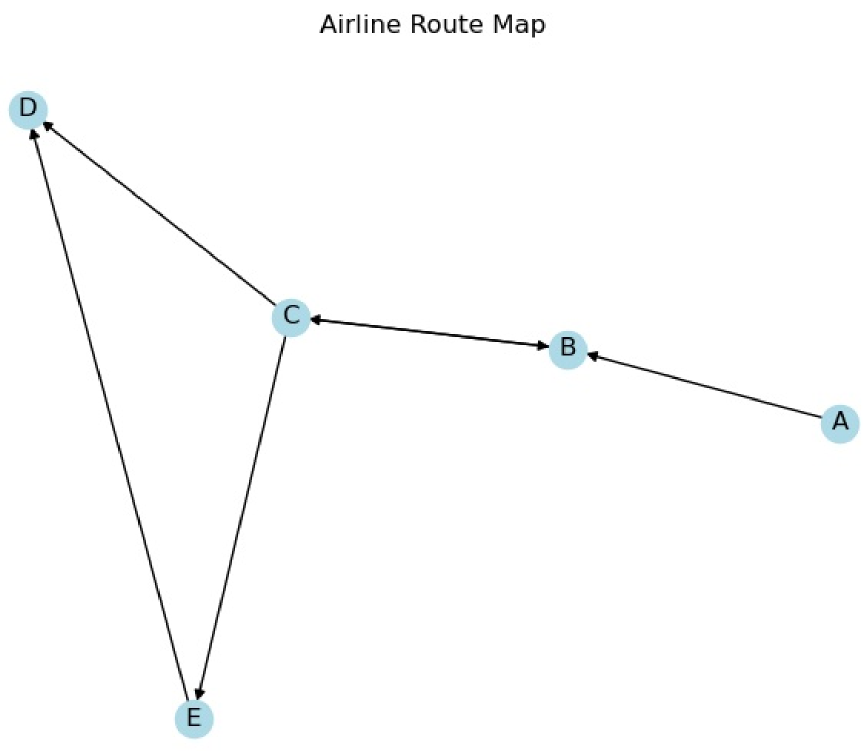
Export Graph to GraphML

```python
nx.write_graphml(G, "example.graphml")
```

Question 01

In [2]:
```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph()
edges = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('C', 'E'), ('E', 'D'), ('C', 'B')]
G.add_edges_from(edges)

print("Adjacency List:")

for node in G.nodes():
 print(f"{node}: {list(G.adj[node])}")

print("\nAdjacency Matrix:")
print(nx.adjacency_matrix(G).todense())

nx.draw(G, with_labels=True, node_color='lightblue', arrows=True)
plt.title("Airline Route Map")
plt.show()
```

```
Adjacency List:
A: ['B']
B: ['C']
C: ['D', 'E', 'B']
D: []
E: ['D']

Adjacency Matrix:
[[0 1 0 0 0]
 [0 0 1 0 0]
 [0 1 0 1 1]
 [0 0 0 0 0]
 [0 0 0 1 0]]
```



Question 02

In [3]:
```
print("In-Degree and Out-Degree:")
for node in G.nodes():
    print(f"{node} - In-degree: {G.in_degree(node)}, Out-degree: {G.out_degree(node)}")

print("\nSelf-loops:")
self_loops = list(nx.selfloop_edges(G))
print("None" if not self_loops else self_loops)
```

```
In-Degree and Out-Degree:
A - In-degree: 0, Out-degree: 1
B - In-degree: 2, Out-degree: 1
C - In-degree: 1, Out-degree: 3
D - In-degree: 2, Out-degree: 0
E - In-degree: 1, Out-degree: 1

Self-loops:
None
```

Question 03

In [5]:
```
print("Is D reachable from A?", nx.has_path(G, 'A', 'D'))
print("\nAll Simple Paths from A to D:")

for path in nx.all_simple_paths(G, source='A', target='D'):
 print(path)
print("\nCycles in the Graph:")
cycles = list(nx.simple_cycles(G))
print("No cycles" if not cycles else cycles)
```

```
Is D reachable from A? True

All Simple Paths from A to D:
['A', 'B', 'C', 'D']
['A', 'B', 'C', 'E', 'D']

Cycles in the Graph:
[['C', 'B']]
```

Question 04

In [6]:
```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    bfs_tree = {start: []}
    parent = {start: None}

    while queue:
        node = queue.popleft()
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited and neighbor not in queue:
                queue.append(neighbor)
                parent[neighbor] = node
                if parent[neighbor] not in bfs_tree:
                    bfs_tree[parent[neighbor]] = []
                bfs_tree[parent[neighbor]].append(neighbor)
    return visited, bfs_tree

visited, bfs_tree = bfs(G.adj, 'A')
print("BFS Visit Order:", visited)
print("BFS Tree:", bfs_tree)
```

```
BFS Visit Order: {'D', 'B', 'A', 'C', 'E'}
BFS Tree: {'A': ['B'], 'B': ['C'], 'C': ['D', 'E']}
```

Question 05

In [7]:
```
import networkx as nx

time = 0
discovery = {}
finishing = {}
visited = set()

def dfs(graph, node):
    global time
    visited.add(node)
    time += 1
    discovery[node] = time

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor)

    time += 1
    finishing[node] = time
```

```python
for node in G.nodes():
    if node not in visited:
        dfs(G.adj, node)

print("Discovery Times:", discovery)
print("Finishing Times:", finishing)
```

```
Discovery Times: {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 6}
Finishing Times: {'D': 5, 'E': 7, 'C': 8, 'B': 9, 'A': 10}
```

Question 07

In [8]:
```python
def is_full_binary_tree(tree):
    n = len(tree)
    for i in range(n):
        left = 2 * i + 1
        right = 2 * i + 2
        if left < n and tree[left] is not None or right < n and tree[right] is not None:
            if not (left < n and tree[left] is not None and right < n and tree[right] is not None):
                return False
    return True

tree = [1, 2, 3, 4, 5, None, None]
print("Is full binary tree?", is_full_binary_tree(tree))
```

```
Is full binary tree? True
```

Question 06

In [9]:
```python
#Method 01
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

GPA_list = [4.0, 3.8, 2.5, 3.2, 3.9, 2.0]
heapify(GPA_list, len(GPA_list), 1)
print("Heap after HEAPIFY from index 1:", GPA_list)
```

```
Heap after HEAPIFY from index 1: [4.0, 3.9, 2.5, 3.2, 3.8, 2.0]
```

In [10]:
```python
#Method 02
import heapq

GPA_list = [-4.0, -3.8, -2.5, -3.2, -3.9, -2.0]
heapq.heapify(GPA_list)

GPA_list = [-x for x in GPA_list]
print("Max-heapified GPA list:", GPA_list)
```

```
Max-heapified GPA list: [4.0, 3.9, 2.5, 3.2, 3.8, 2.0]
```

Question 08

In [1]:
```python
import networkx as nx

G = nx.Graph()

G.add_edge('A', 'B', weight=2)
G.add_edge('A', 'D', weight=6)
G.add_edge('B', 'C', weight=3)
G.add_edge('B', 'D', weight=8)
G.add_edge('C', 'D', weight=5)

mst = nx.minimum_spanning_tree(G, algorithm='prim')

print("Minimum Spanning Tree edges:")
for u, v, data in mst.edges(data=True):
    print(f"{u} - {v} \t weight: {data['weight']}")

total_weight = sum(data['weight'] for u, v, data in mst.edges(data=True))
print("Total weight of MST:", total_weight)
```

```
Minimum Spanning Tree edges:
A - B    weight: 2
B - C    weight: 3
D - C    weight: 5
Total weight of MST: 10
```

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js