Sri Lanka Institute of Information Technology

Lab Submission 08

IT24101541

Peiris M. W. U. L.

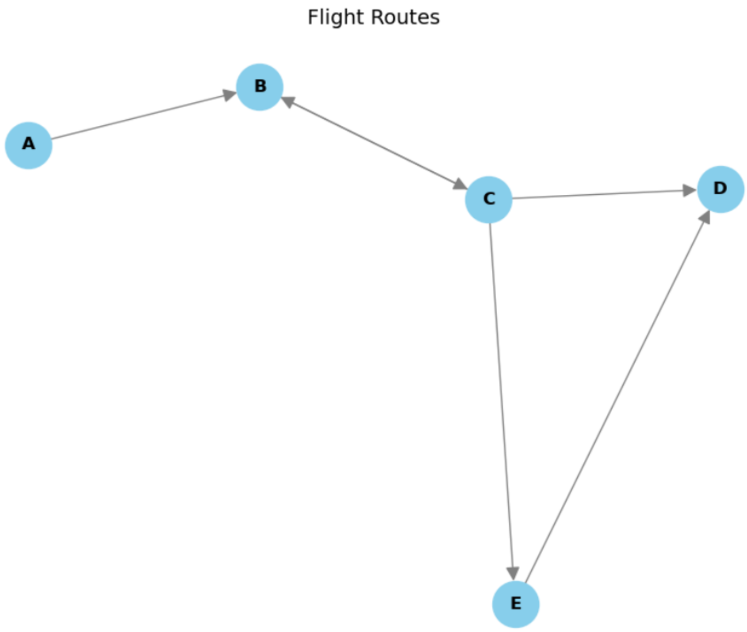
Discrete Mathematics | IT1160

B.Sc. (Hons) in Information Technology

Part A

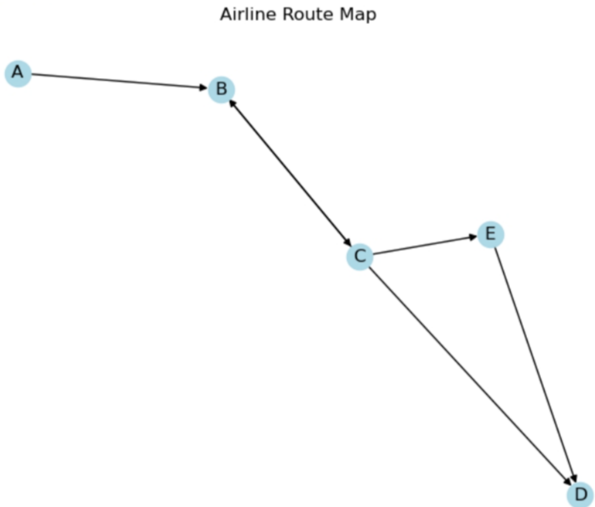
1.

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

cities = ['A','B','C','D','E']
flights = [('A','B'),('B','C'),('C','D'),('C','E'),('E','D'),('C','B')]

G = nx.DiGraph()
G.add_nodes_from(cities)
G.add_edges_from(flights)

plt.figure(figsize=(8,6))
pos = nx.spring_layout(G,seed=123)

nx.draw(G, pos,
       with_labels = True,
       node_size = 1000,
       node_color = 'skyblue',
       edge_color = 'gray',
       arrowsize = 20,
       font_weight = 'bold')

plt.title("Flight Routes",fontsize = 14)
plt.show()
```




```python
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]]
```




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

cities = ['A','B','C','D','E']
flights = [('A','B'),('B','C'),('C','D'),('C','E'),('E','D'),('C','B')]

G = nx.DiGraph()
G.add_nodes_from(cities)
G.add_edges_from(flights)


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
```

2.

```
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:
[['B', 'C']]
```

3.

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

4.

```python
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
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}
```

5.

```python
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]
```

6.

```python
def is_full_binary_tree(tree):
    n = len(tree)
    for i in range(n):
        if tree[i] is not None:
            left_child = 2*i+1
            right_child = 2*i+2
            has_left = left_child < n and tree[left_child] is not None
            has_right = right_child < n and tree[right_child] is not None
            if has_left != has_right:
                return False
    return True

tree1 = [1,2,3,4,5,None,None] #Full Binary Tree
tree2 = [1,2,3,4,None,None,None] # Not full
tree3 = [1,2,3,None,None,None,None] #FBT
tree4 = [1,2,3,4,5,6,None] #Not full

print(is_full_binary_tree(tree2))
```

```
False
```

7.

```python
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]
```

8.

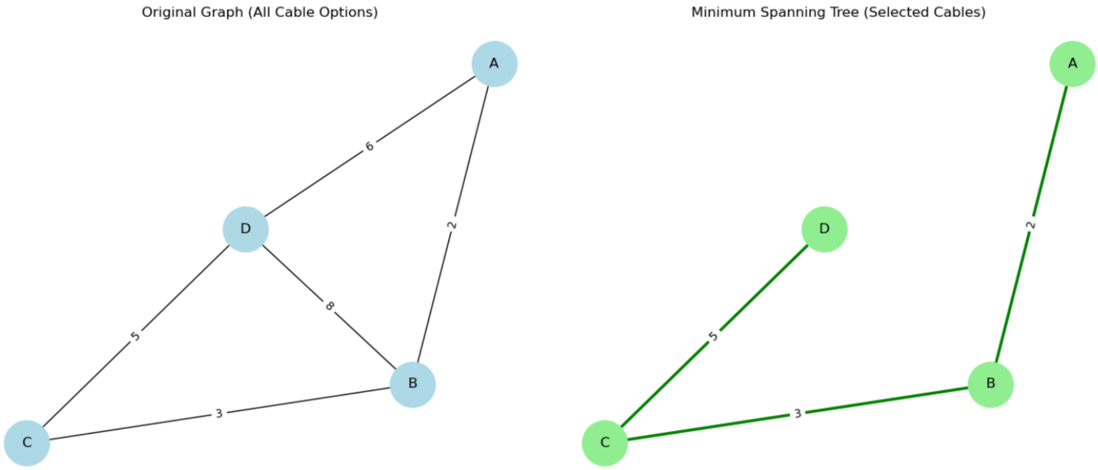
9.

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

# a) Model the problem as a weighted undirected graph
graph = defaultdict(list)

edges = [
    ('A', 'B', 2),
    ('A', 'D', 6),
    ('B', 'C', 3),
    ('B', 'D', 8),
    ('C', 'D', 5)
]

# Add edges to the graph (undirected)
for u, v, cost in edges:
    graph[u].append((v, cost))
    graph[v].append((u, cost))

print("a) Graph model as an adjacency list (weighted undirected graph):")
for node in sorted(graph):
    print(f"   {node} -> {[(neighbor, cost) for neighbor, cost in graph[node]]}")

# b) Find the Minimum Spanning Tree using Prim's Algorithm
def prims_algorithm(start):
    visited = set()
    min_heap = []
    mst = []
    total_cost = 0
```

```python
    visited.add(start)
    for neighbor, cost in graph[start]:
        heapq.heappush(min_heap, (cost, start, neighbor))

    while min_heap and len(visited) < len(graph):
        cost, u, v = heapq.heappop(min_heap)
        if v not in visited:
            visited.add(v)
            mst.append((u, v, cost))
            total_cost += cost
            for next_neighbor, next_cost in graph[v]:
                if next_neighbor not in visited:
                    heapq.heappush(min_heap, (next_cost, v, next_neighbor))

    return mst, total_cost

mst_result, total_cost = prims_algorithm('A')

# c) Display the selected cable connections and their costs
print("\nb) Minimum Spanning Tree using Prim's Algorithm calculated.")
print("\nc) Selected cable connections and their costs:")
for u, v, cost in mst_result:
    print(f"   {u} - {v}: {cost} million")

# d) Show the total installation cost
print(f"\n\nd) Total installation cost: {total_cost} million")
```

```python
# ======== Visualization ========

# Create original full graph
G = nx.Graph()
G.add_weighted_edges_from(edges)

# Create MST graph
MST = nx.Graph()
MST.add_weighted_edges_from(mst_result)

# Use same layout for consistency
pos = nx.spring_layout(G, seed=42)

# Plotting
plt.figure(figsize=(14, 6))

# Plot original graph
plt.subplot(1, 2, 1)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1500, font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G, 'weight'))
plt.title("Original Graph (All Cable Options)")

# Plot MST
plt.subplot(1, 2, 2)
nx.draw(MST, pos, with_labels=True, node_color='lightgreen', node_size=1500, font_size=12, edge_color='green', width=2.5)
nx.draw_networkx_edge_labels(MST, pos, edge_labels=nx.get_edge_attributes(MST, 'weight'))
plt.title("Minimum Spanning Tree (Selected Cables)")

plt.tight_layout()
plt.show()
```

```
a) Graph model as an adjacency list (weighted undirected graph):
   A -> [('B', 2), ('D', 6)]
   B -> [('A', 2), ('C', 3), ('D', 8)]
   C -> [('B', 3), ('D', 5)]
   D -> [('A', 6), ('B', 8), ('C', 5)]

b) Minimum Spanning Tree using Prim's Algorithm calculated.

c) Selected cable connections and their costs:
   A - B: 2 million
   B - C: 3 million
   C - D: 5 million

d) Total installation cost: 10 million
```



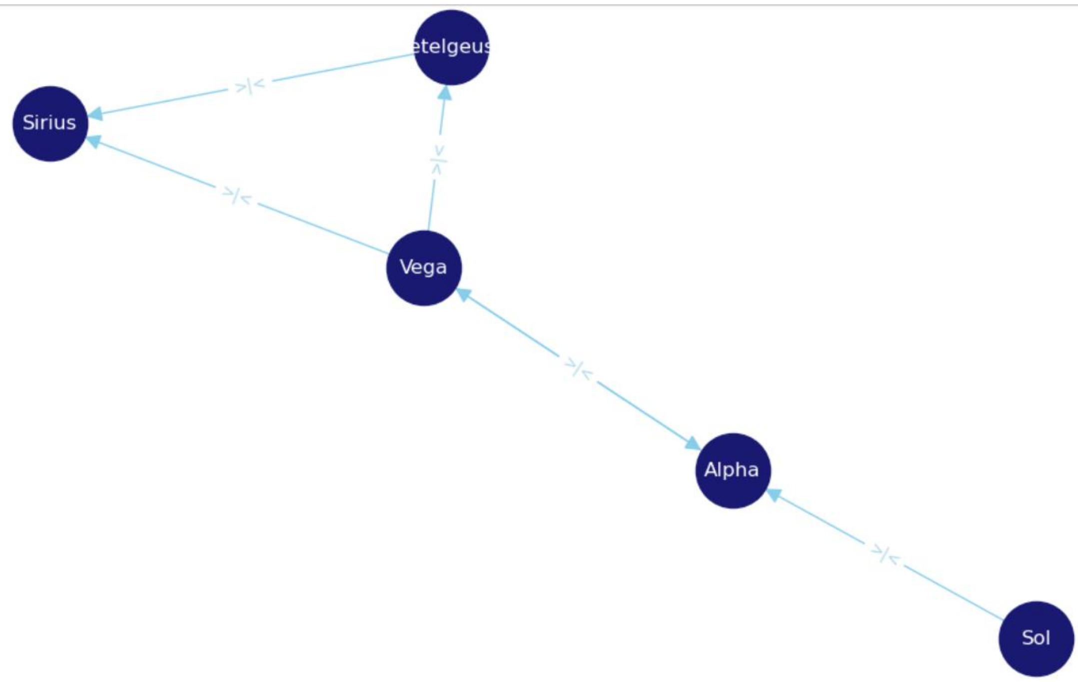
Part B

6.

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

# Star systems and routes
star_systems = ["Sol", "Alpha", "Vega", "Sirius", "Betelgeuse"]
travel_routes = [
    ("Sol", "Alpha"),
    ("Alpha", "Vega"),
    ("Vega", "Sirius"),
    ("Vega", "Betelgeuse"),
    ("Betelgeuse", "Sirius"),
    ("Vega", "Alpha")
]

# a) Build the star map as a directed graph
G = nx.DiGraph()
G.add_nodes_from(star_systems)
G.add_edges_from(travel_routes)

# b) Display adjacency list
print("06-b) Adjacency List:")
for node in G.nodes():
    print(f"  {node} -> {list(G.successors(node))}")

# Adjacency matrix
print("\nAdjacency Matrix:")
matrix = nx.adjacency_matrix(G, nodelist=star_systems).todense()
print(np.array(matrix))
```

```
06-b) Adjacency List:
  Sol -> ['Alpha']
  Alpha -> ['Vega']
  Vega -> ['Sirius', 'Betelgeuse', 'Alpha']
  Sirius -> []
  Betelgeuse -> ['Sirius']

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]]
```

```python
# c) Visualization with a space-themed aesthetic
plt.figure(figsize=(10, 6))
pos = nx.spring_layout(G, seed=42)

# Draw nodes and edges
nx.draw(G, pos, with_labels=True, node_color='midnightblue', edge_color='skyblue',
        font_color='white', node_size=2000, font_size=12, arrowsize=20)

# Edge labels
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): ">|<" for u, v in G.edges()}, font_color='lightblue')
plt.title("Intergalactic Star Route Map", fontsize=15, color='white')
plt.gca().set_facecolor("black")
plt.axis("off")
plt.show()
```



7.

```python
# a) Compute in-degree and out-degree
print("\n07-a) Starport Traffic (In-degree & Out-degree):")
for node in G.nodes():
    print(f"  {node}: In-degree = {G.in_degree(node)}, Out-degree = {G.out_degree(node)}")

# b) Identify self-loops
print("\n07-b) Self-loops (Wormholes to self):")
self_loops = list(nx.selfloop_edges(G))
if self_loops:
    for u, v in self_loops:
        print(f"  {u} has a self-loop.")
else:
    print("  No self-loops found.")
```

```
07-a) Starport Traffic (In-degree & Out-degree):
  Sol: In-degree = 0, Out-degree = 1
  Alpha: In-degree = 2, Out-degree = 1
  Vega: In-degree = 1, Out-degree = 3
  Sirius: In-degree = 2, Out-degree = 0
  Betelgeuse: In-degree = 1, Out-degree = 1

07-b) Self-loops (Wormholes to self):
  No self-loops found.
```

8.

```python
# a) Check if Sirius is reachable from Sol
print("\n08-a) Reachability:")
is_reachable = nx.has_path(G, source="Sol", target="Sirius")
print(f"  Sirius is {'reachable' if is_reachable else 'not reachable'} from Sol.")

# b) List all simple paths from Sol to Sirius
print("\n08-b) All Simple Paths from Sol to Sirius:")
paths = list(nx.all_simple_paths(G, source="Sol", target="Sirius"))
for i, path in enumerate(paths, 1):
    print(f"  Path {i}: {' -> '.join(path)}")

# c) Detect cycles
print("\n08-c) Cycle Detection:")
cycles = list(nx.simple_cycles(G))
if cycles:
    for i, cycle in enumerate(cycles, 1):
        print(f"  Cycle {i}: {' -> '.join(cycle)}")
else:
    print("  No cycles found in the network.")
```

```
08-a) Reachability:
  Sirius is reachable from Sol.

08-b) All Simple Paths from Sol to Sirius:
  Path 1: Sol -> Alpha -> Vega -> Sirius
  Path 2: Sol -> Alpha -> Vega -> Betelgeuse -> Sirius

08-c) Cycle Detection:
  Cycle 1: Vega -> Alpha
```

9.

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    bfs_tree = nx.DiGraph()
    visit_order = []

    visited.add(start)

    while queue:
        node = queue.popleft()
        visit_order.append(node)

        for neighbor in graph.successors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                bfs_tree.add_edge(node, neighbor)

    return visit_order, bfs_tree

# Perform BFS from Sol
visit_order_bfs, bfs_tree = bfs(G, "Sol")

# Output results
print("\n09) Breadth-First Search (BFS) from Sol")
print("  Visit Order:", visit_order_bfs)
print("  BFS Tree Edges:")
for u, v in bfs_tree.edges():
    print(f"    {u} → {v}")
```

```
09) Breadth-First Search (BFS) from Sol
  Visit Order: ['Sol', 'Alpha', 'Vega', 'Sirius', 'Betelgeuse']
  BFS Tree Edges:
    Sol → Alpha
    Alpha → Vega
    Vega → Sirius
    Vega → Betelgeuse
```

10.

```python
time = 0
discovery = {}
finishing = {}
dfs_tree = nx.DiGraph()

def dfs_recursive(graph, node, visited, parent=None):
    global time
    visited.add(node)
    time += 1
    discovery[node] = time
    for neighbor in graph.successors(node):
        if neighbor not in visited:
            if parent:
                dfs_tree.add_edge(node, neighbor)
            dfs_recursive(graph, neighbor, visited, node)

    time += 1
    finishing[node] = time

# Run DFS from Sol
visited_dfs = set()
time = 0  # Reset global clock
dfs_recursive(G, "Sol", visited_dfs)
# Output timestamps
print("\n10) Depth-First Search (DFS) with Discovery and Finishing Times")
for node in discovery:
    print(f"  {node}: Discovery = {discovery[node]}, Finishing = {finishing[node]}")
# DFS Tree
print("\n  DFS Tree Edges:")
for u, v in dfs_tree.edges():
    print(f"    {u} → {v}")
```

```
10) Depth-First Search (DFS) with Discovery and Finishing Times
  Sol: Discovery = 1, Finishing = 10
  Alpha: Discovery = 2, Finishing = 9
  Vega: Discovery = 3, Finishing = 8
  Sirius: Discovery = 4, Finishing = 5
  Betelgeuse: Discovery = 6, Finishing = 7

  DFS Tree Edges:
    Alpha → Vega
    Vega → Sirius
    Vega → Betelgeuse
```

11.

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

# Initial nearly max-heap
gpa_list = [4.0, 3.8, 2.5, 3.2, 3.9, 2.0]

print("Original list:", gpa_list)

# Apply heapify at index 1
heapify(gpa_list, len(gpa_list), 1)

print("After heapify at index 1:", gpa_list)
```

```
Original list: [4.0, 3.8, 2.5, 3.2, 3.9, 2.0]
After heapify at index 1: [4.0, 3.9, 2.5, 3.2, 3.8, 2.0]
```

12.

```python
def sift_up(arr, i):
    # Bubble up the updated element if it's larger than its parent
    parent = (i - 1) // 2
    while i > 0 and arr[i] > arr[parent]:
        arr[i], arr[parent] = arr[parent], arr[i]
        i = parent
        parent = (i - 1) // 2

ratings = [4.9, 4.7, 4.5, 3.8, 4.8]

print("Original ratings:", ratings)
ratings[3] = 4.85
sift_up(ratings, 3)

print("After updating index 3 and heapifying:", ratings)
```

```
Original ratings: [4.9, 4.7, 4.5, 3.8, 4.8]
After updating index 3 and heapifying: [4.9, 4.85, 4.5, 4.7, 4.8]
```

13. 14. 15.

```python
import networkx as nx

# Create a weighted undirected graph
G = nx.Graph()

# Add edges along with their costs
edges = [
    ('P', 'Q', 4),
    ('P', 'S', 7),
    ('Q', 'R', 2),
    ('Q', 'S', 6),
    ('R', 'S', 3)
]

G.add_weighted_edges_from(edges)

# Compute the Minimum Spanning Tree (MST) using Prim's Algorithm
mst = nx.minimum_spanning_tree(G)

# Display the selected route connections and costs
print("Selected Route Connections (Minimum Spanning Tree):")
total_cost = 0
for edge in mst.edges(data=True):
    print(f"{edge[0]} - {edge[1]}: Cost = {edge[2]['weight']} million")
    total_cost += edge[2]['weight']

print(f"\nTotal Cost of Construction: {total_cost} million")
```

```
Selected Route Connections (Minimum Spanning Tree):
P - Q: Cost = 4 million
Q - R: Cost = 2 million
S - R: Cost = 3 million

Total Cost of Construction: 9 million
```