

**Telecommunication Software**

Second Practical Exercise

Full name: **Mithin kumar Ananthula**

ID: **241AEM014**

**Riga 2024**

**Contents**

Table of Contents

task 1)Python ADT Stack and Graphs 3

task 2) Bellman-Ford Algorithm 5

task 3) Dijkstra's Algorithm 10

task 4) Prim's Algorithm 15

task 5) Q-Learning and Shortest Path 22

task 6) RIP Experiment and Requirements (Cisco Packet Tracer)25

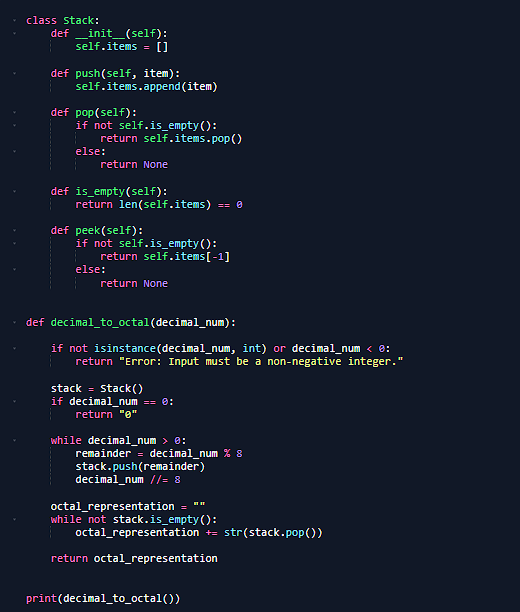
task 7) OSPF Experiment and Requirements (Cisco Packet Tracer 28

task 828

giit hub link28

# Task1 Python ADT Stack and Graphs:

**1.1:** Implement a function to convert a decimal number to its octal representation using a Python Stack.



thedecimal number to octal representation output examples

print(decimal\_to\_octal(10))

# Output: 12

print(decimal\_to\_octal(64))

# Output: 100

print(decimal\_to\_octal(233))

# Output: 351

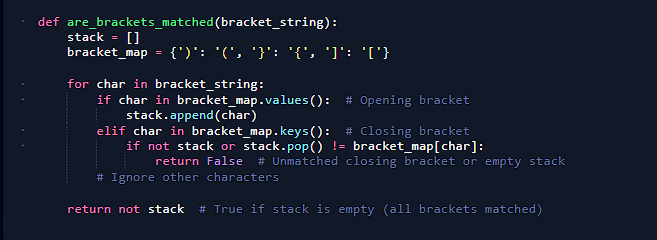
print(decimal\_to\_octal(0))

# Output: 0

print(decimal\_to\_octal(1234))

# Output: 2322

**1.2** : Write a function to check if a series of brackets (square brackets, curly braces, and parentheses) are properly matched.



The output examples

print(are\_brackets\_matched("()[]{}")) # True

print(are\_brackets\_matched("([{}])")) # True

print(are\_brackets\_matched("{[()]")) # True

print(are\_brackets\_matched("{[(])}")) # False

print(are\_brackets\_matched("((()))")) # True

print(are\_brackets\_matched("(()")) # False

print(are\_brackets\_matched(")(")) # False

print(are\_brackets\_matched("")) # True

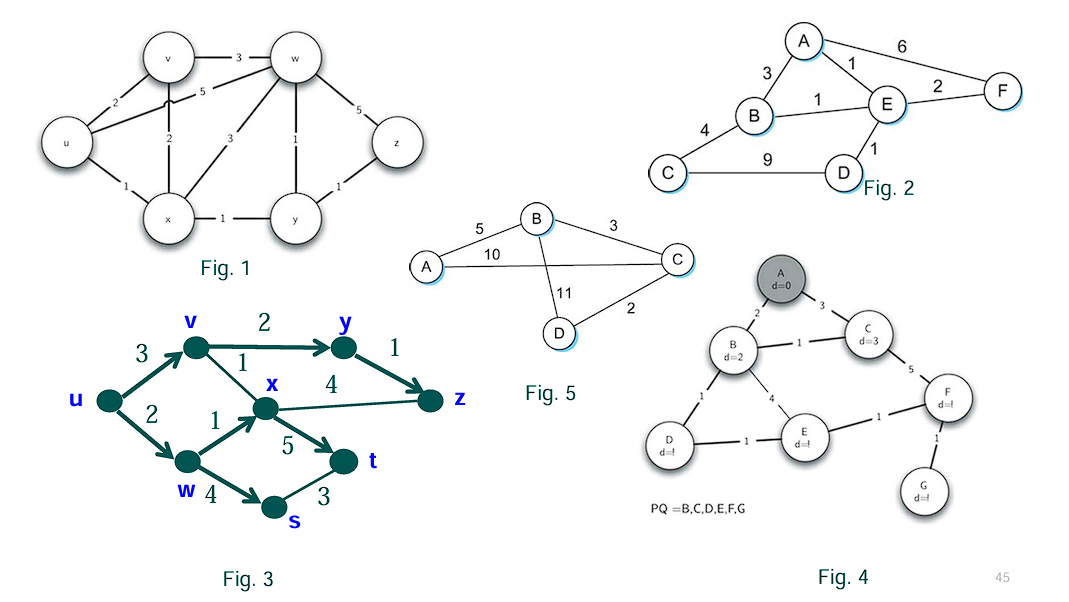
print(are\_brackets\_matched("abc")) # True (ignores non-brackets)

print(are\_brackets\_matched("{([])}")) # False

## Task 2 Bellman-Ford Algorithm:

Implement the Bellman-Ford algorithm to traverse 5 given graphs (from the course slides). For each graph, find the shortest path from all nodes to their neighboring nodes. Plot the graph, including nodes, edges, and weights.

given graphs from the course slides

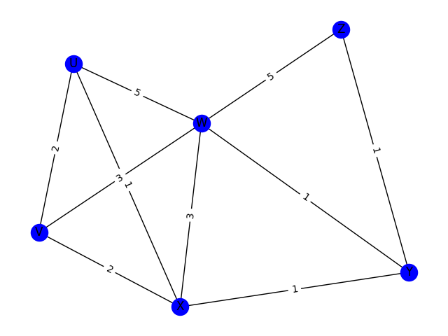


2.1 Bellman ford algorithm for the first figure graph:



out : Shortest paths from Y: {'U': 2, 'V': 3, 'W': 1, 'X': 1, 'Y': 0, 'Z': 1}

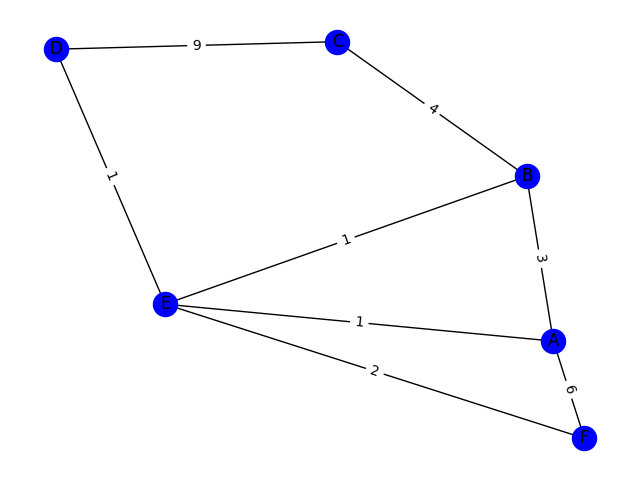
Shortest paths from Y: {'U': inf, 'V': inf, 'W': inf, 'X': inf, 'Y': 0, 'Z': 1}



2.2 Bellman ford algorithm for the Second figure graph:



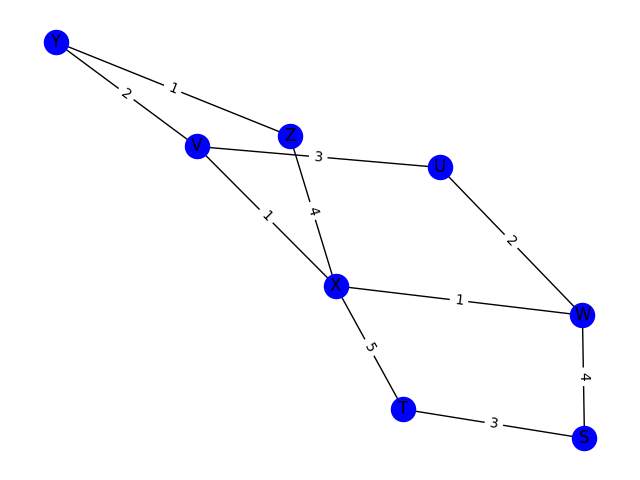
Shortest paths from A: {'A': 0, 'B': 2, 'E': 1, 'F': 3, 'C': 6, 'D': 2}



2.3 Bellman ford algorithm for the third figure graph:



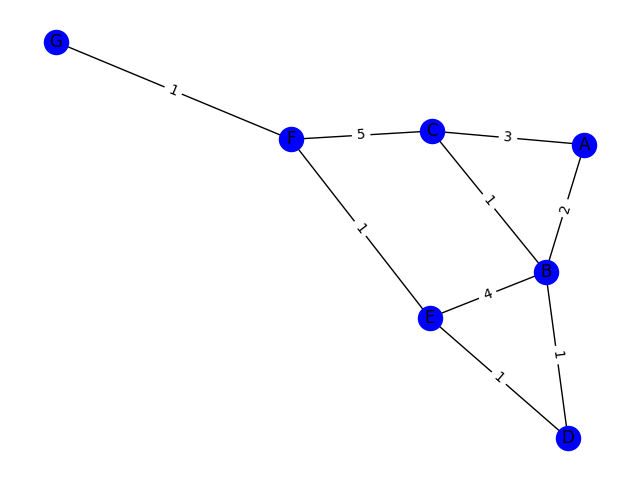
Shortest paths from U: {'U': 0, 'V': 3, 'W': 2, 'X': 3, 'Y': 5, 'Z': 6, 'T': 8, 'S': 6}



2.4 Bellman ford algorithm for the forth figure graph:



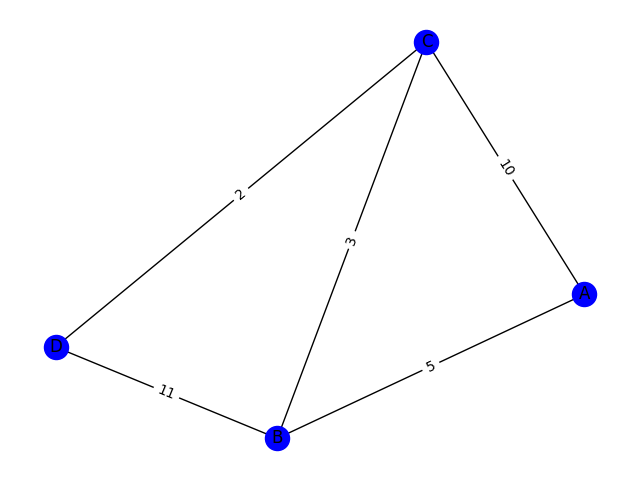
Shortest paths from G: {'A': 6, 'B': 4, 'C': 5, 'D': 3, 'E': 2, 'F': 1, 'G': 0}



2.5 Bellman ford algorithm for the fifth figure graph:



Shortest paths from A: {'A': 0, 'B': 5, 'C': 8, 'D': 10}



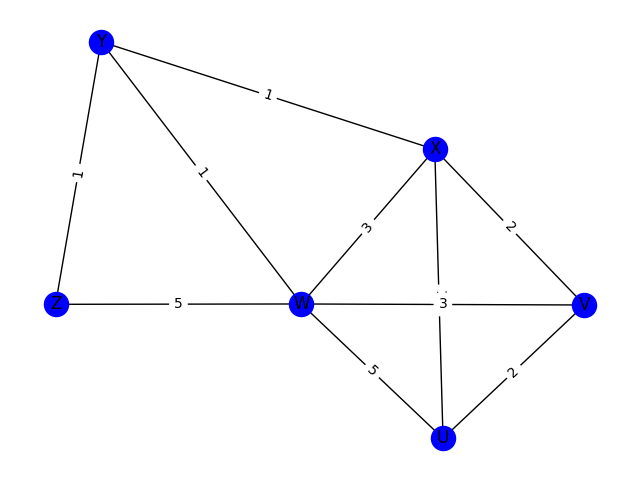
#### Task 3 Dijkstra's Algorithm:

Implement Dijkstra’s algorithm to traverse the same 5 graphs. For each, find the shortest path from all nodes to their neighboring nodes, and plot the graph with nodes, edges, and weights.

3.1 Dijkstra’s algorithm for 1st figure graph:



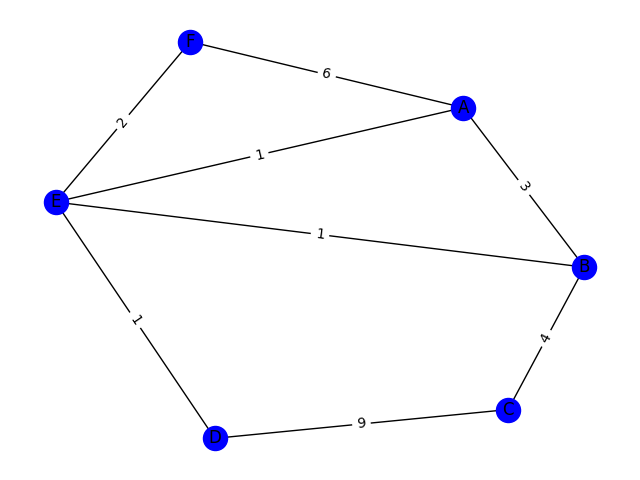
Shortest paths from Z: {'U': 3, 'V': 4, 'W': 2, 'X': 2, 'Y': 1, 'Z': 0}



3.2 Dijkstra’s algorithm for 2nd figure graph:



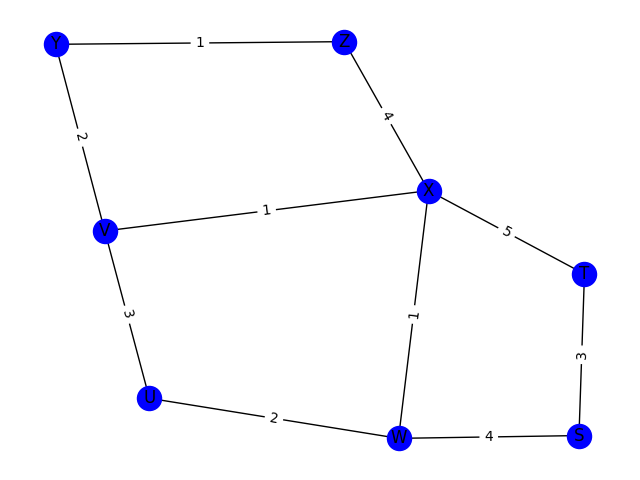
Shortest paths from A: {'A': 0, 'B': 2, 'E': 1, 'F': 3, 'C': 6, 'D': 2}



3.3 Dijkstra’s algorithm for 3rd figure graph:



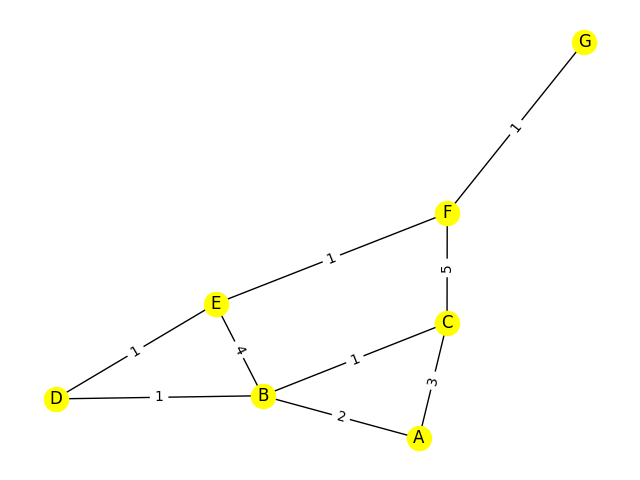
Shortest paths from U: {'U': 0, 'V': 3, 'W': 2, 'X': 3, 'Y': 5, 'Z': 6, 'T': 8, 'S': 6}



3.4 Dijkstra’s algorithm for 4th figure graph:



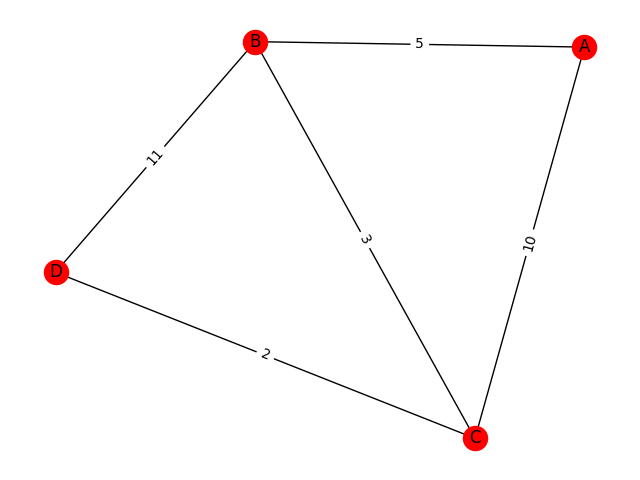
Shortest paths from A: {'A': 0, 'B': 2, 'C': 3, 'D': 3, 'E': 4, 'F': 5, 'G': 6}



3.5 Dijkstra’s algorithm for 5th figure graph:



Shortest paths from A: {'A': 0, 'B': 5, 'C': 8, 'D': 10}



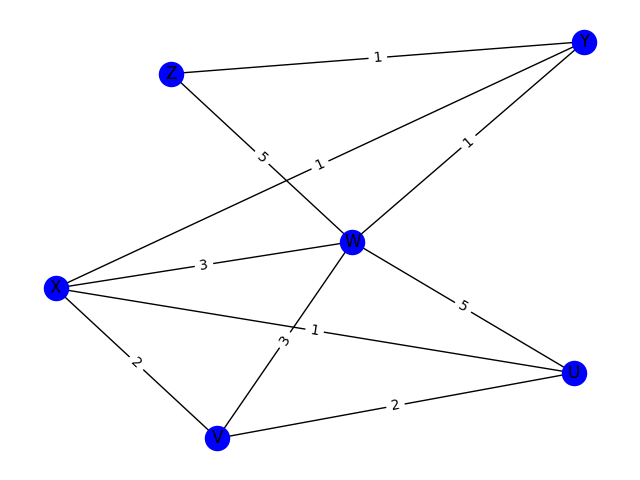
##### Task 4 Prim's Algorithm:

Implement Prim's algorithm for the same 5 graphs and traverse all nodes to find the minimum weight cost.

4.1 Prim's algorithm for 1st figure graph:



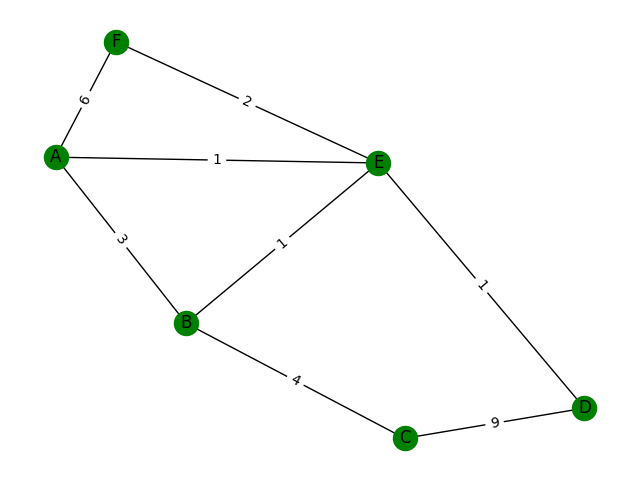
Minimum spanning tree edges: [('U', 'X'), ('U', 'V'), ('X', 'Y'), ('Y', 'W'), ('Y', 'Z')]



4.2 Prim's algorithm for 2nd figure graph:



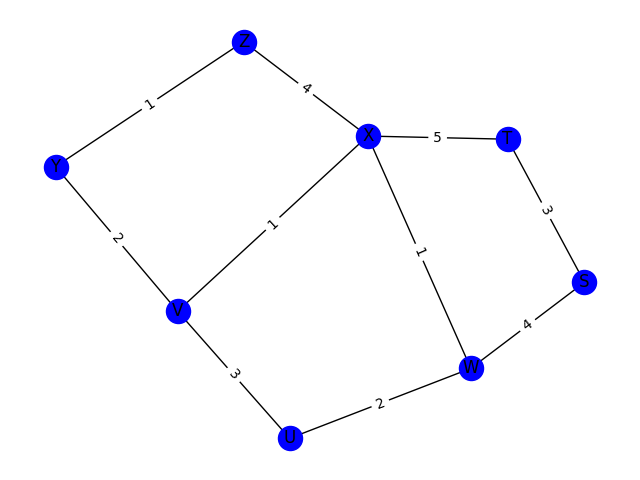
Minimum spanning tree edges: [('A', 'E'), ('E', 'B'), ('E', 'D'), ('E', 'F'), ('B', 'C')]



4.3 Prim's algorithm for 3rd figure graph:



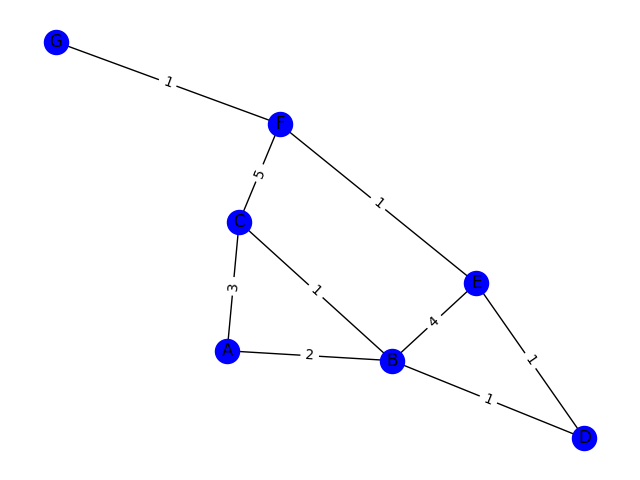
Minimum spanning tree edges: [('U', 'W'), ('W', 'X'), ('W', 'S'), ('X', 'V'), ('V', 'Y'), ('Y', 'Z'), ('S', 'T')]



4.4 Prim's algorithm for 4th figure graph:



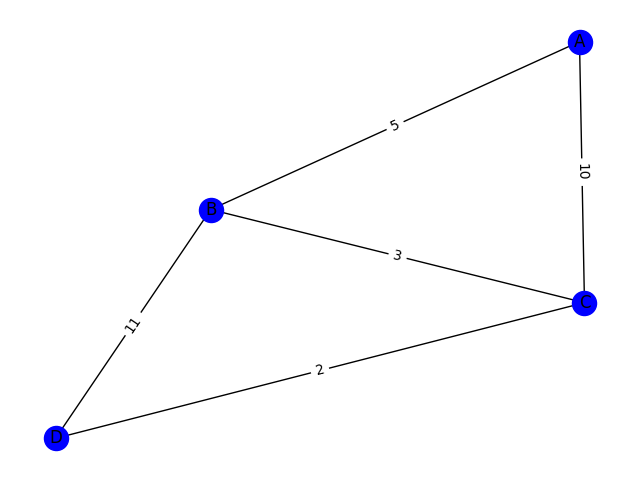
Minimum spanning tree edges: [('A', 'B'), ('B', 'C'), ('B', 'D'), ('D', 'E'), ('E', 'F'), ('F', 'G')]



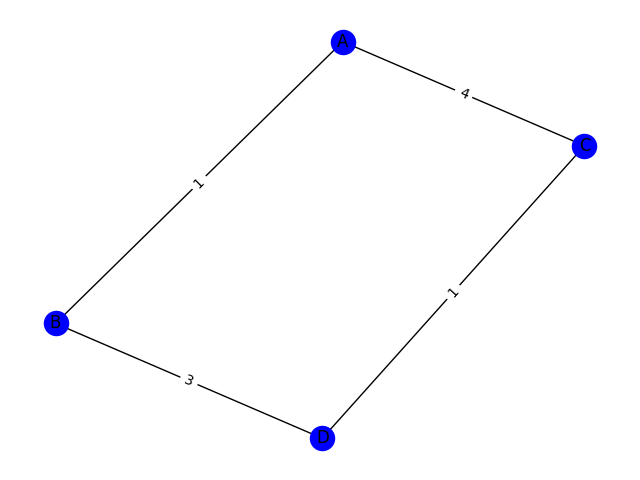
4.5 Prim's algorithm for 5th figure graph:



Minimum spanning tree edges: [('A', 'B'), ('B', 'C'), ('C', 'D')]



4.6 Compare Bellman-Ford and Dijkstra’s algorithms on the same graphs and note any path differences.



**Bellman-Ford Result**:

*A*→*B*: A→C→D→B*A*→*C*→*D*→ *B* (cost = 7).

**Dijkstra Result**:

*A*→*B*: A→B*A*→*B* (cost = 1).

| **Feature** | **Bellman-Ford** | **Dijkstra** |
| --- | --- | --- |
| **Purpose** | Finds shortest paths, supports negative weights. | Finds shortest paths, assumes non-negative weights. |
| **Relaxation Technique** | Repeats edge relaxation for V−1 iterations (where Vis the number of vertices). | Relaxes edges in a greedy manner using a priority queue. |
| **Complexity** | O(VE) | O((V+E)logV) with a priority queue |
| **Negative Cycles** | Detects and reports negative cycles. | Cannot handle graphs with negative weights. |
| **Graph Type** | Works on directed/undirected graphs, with or without negative weights. | Works only for non-negative edge weights. |

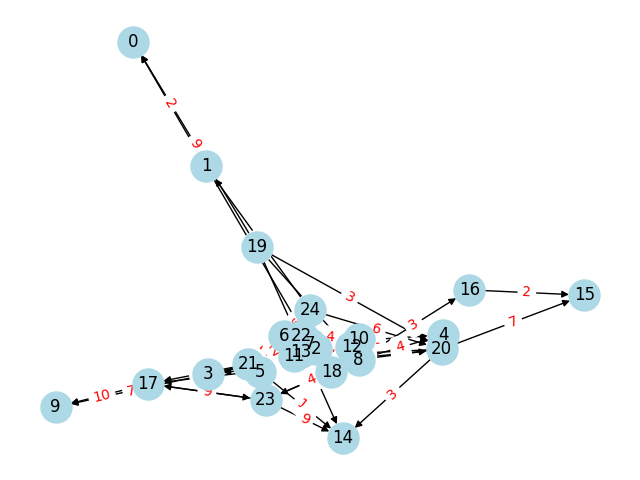
4.7 Discuss the differences between Dijkstra’s and Prim's algorithm

| **Aspect** | **Dijkstra’s Algorithm** | **Prim’s Algorithm** |
| --- | --- | --- |
| **Purpose** | Finds the shortest path from a source node to all other nodes. | Finds the minimum spanning tree (MST) of a graph. |
| **Graph Type** | Works on weighted graphs with non-negative weights. | Works on weighted graphs (negative weights allowed). |
| **Input Requirements** | Requires a source vertex. | Does not require a specific source vertex. |
| **Output** | Shortest path tree (SPT) with distances from the source. | Minimum spanning tree (MST) connecting all vertices. |
| **Optimization Goal** | Minimizes the sum of path weights from the source to each node. | Minimizes the sum of edge weights in the spanning tree. |
| **Algorithm Type** | Greedy algorithm for shortest path. | Greedy algorithm for spanning tree. |
| **Data Structures Used** | Priority queue (e.g., binary heap). | Priority queue (e.g., binary heap). |
| **Cycle Handling** | Avoids re-visiting nodes using a priority queue. | Avoids forming cycles by ensuring unvisited nodes only. |
| **Applicability** | Suitable for pathfinding problems (e.g., navigation). | Suitable for network design problems (e.g., wiring, infrastructure). |
| **Edge Weights** | Requires non-negative edge weights. | Works with positive or negative edge weights. |

###### Task 5 Q-Learning and Shortest Path:

Review the Q-learning example on GitHub, and apply the reinforcement learning algorithm to find the shortest path between nodes in a graph of your design (at least 25 nodes). Apply Dijkstra’s and Prim’s algorithms to the same graph, and compare their results with those from the Q-learning. What are the differences in the results and performance?







Dijkstra's Shortest Path Results:

Path from 10 to 10: [10] with total weight 0

Path from 10 to 8: [10, 8] with total weight 3

Path from 10 to 12: [10, 12] with total weight 5

Path from 10 to 20: [10, 8, 4, 20] with total weight 10

Path from 10 to 4: [10, 8, 4] with total weight 7

Path from 10 to 23: [10, 8, 17, 23] with total weight 8

Path from 10 to 17: [10, 8, 17] with total weight 4

Path from 10 to 16: [10, 8, 16] with total weight 6

Path from 10 to 9: [10, 8, 17, 9] with total weight 14

Path from 10 to 15: [10, 8, 16, 15] with total weight 8

Path from 10 to 14: [10, 8, 4, 20, 14] with total weight 13

Path from 10 to 3: [10, 8, 4, 20, 3] with total weight 11

Path from 10 to 11: [10, 8, 4, 20, 11] with total weight 13

Path from 10 to 2: [10, 8, 4, 20, 3, 2] with total weight 18

Path from 10 to 18: [10, 8, 4, 20, 11, 18] with total weight 17

Path from 10 to 7: [10, 8, 4, 20, 11, 7] with total weight 16

Path from 10 to 21: [10, 8, 4, 20, 11, 7, 21] with total weight 22

Path from 10 to 6: [10, 8, 4, 20, 3, 2, 6] with total weight 28

Path from 10 to 13: [10, 8, 4, 20, 11, 7, 21, 13] with total weight 32

Path from 10 to 22: [10, 8, 4, 20, 3, 2, 6, 22] with total weight 37

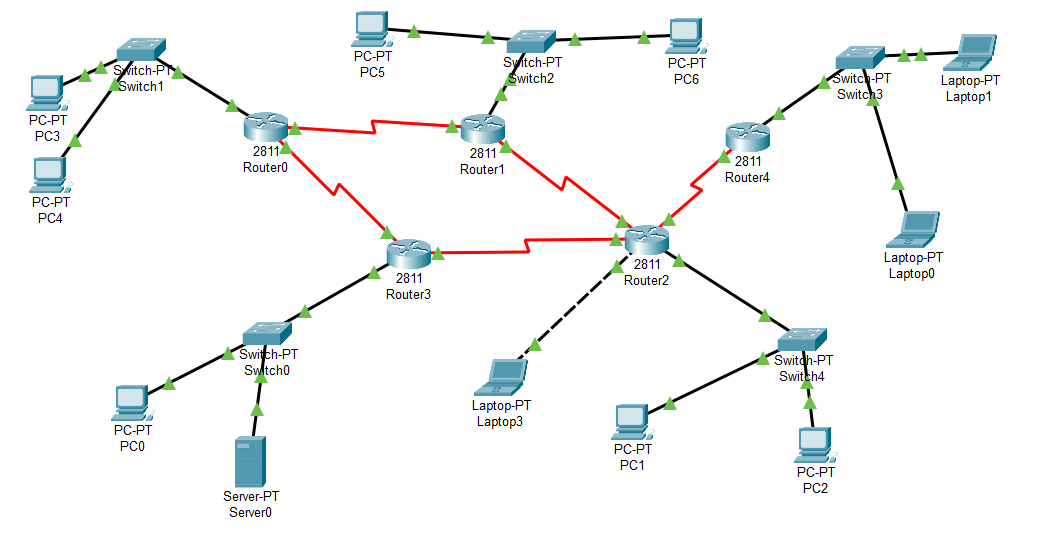
Path from 10 to 24: [10, 8, 4, 20, 3, 2, 6, 24] with total weight 34

Path from 10 to 5: [10, 8, 4, 20, 3, 2, 6, 24, 5] with total weight 41

Path from 10 to 1: [10, 8, 4, 20, 3, 2, 6, 24, 1] with total weight 36

Path from 10 to 0: [10, 8, 4, 20, 3, 2, 6, 24, 1, 0] with total weight 38

Task 6 RIP Experiment and Requirements (Cisco Packet Tracer):

****

All routers and endpoints should be able to ping each other successfully.

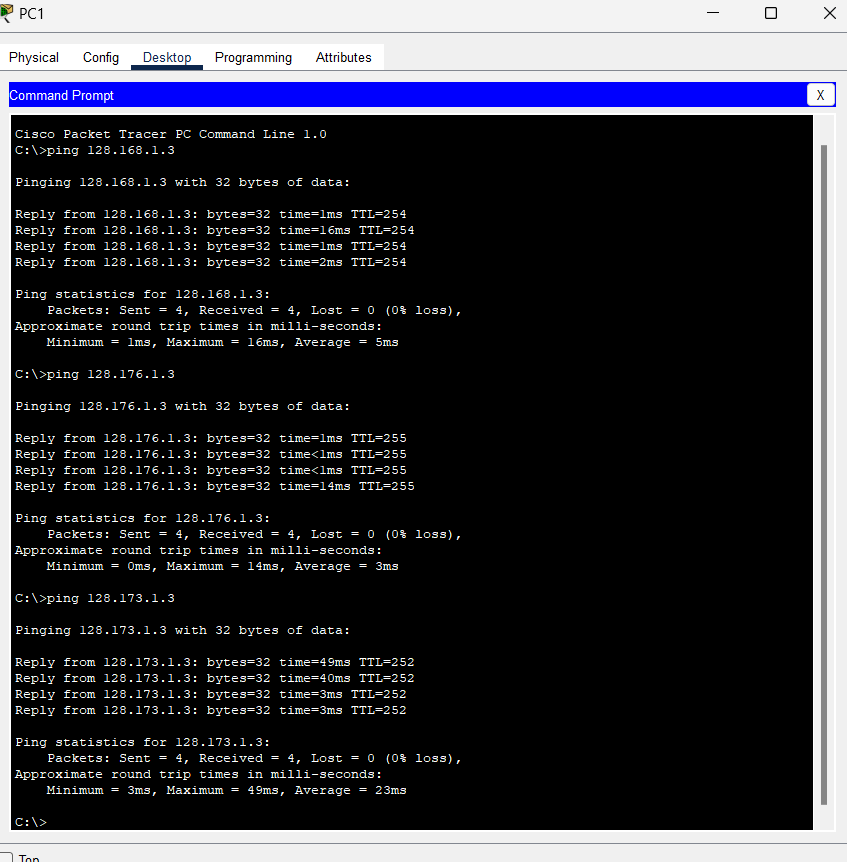
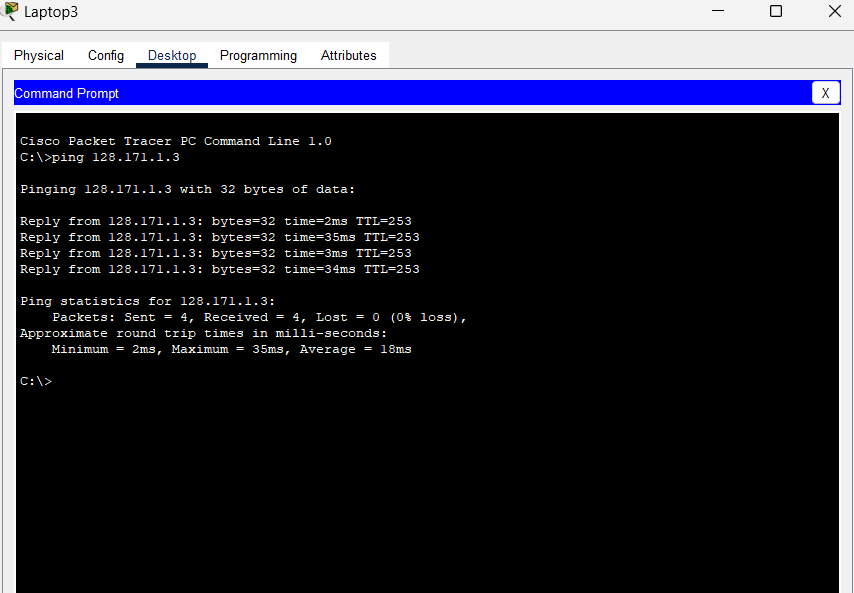
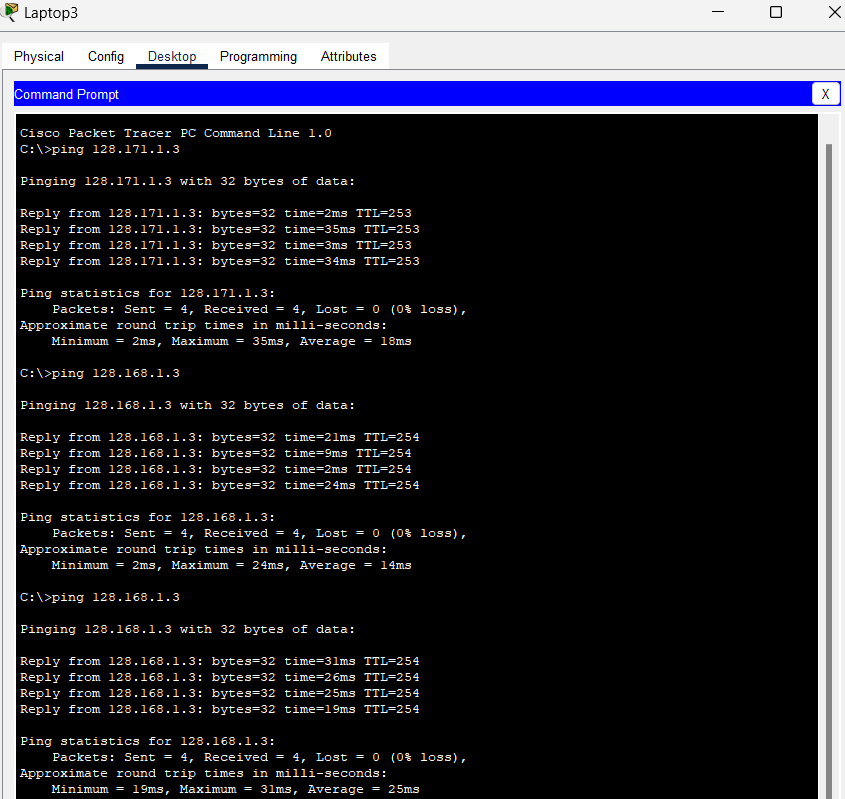
Configure and advertise all network segments of adjacent devices and display RIP information.

Check routers’ routing tables to verify IP address entries.

Configure LoopBack0 on core routers and verify its operation. Understand why the loopback IP is necessary.

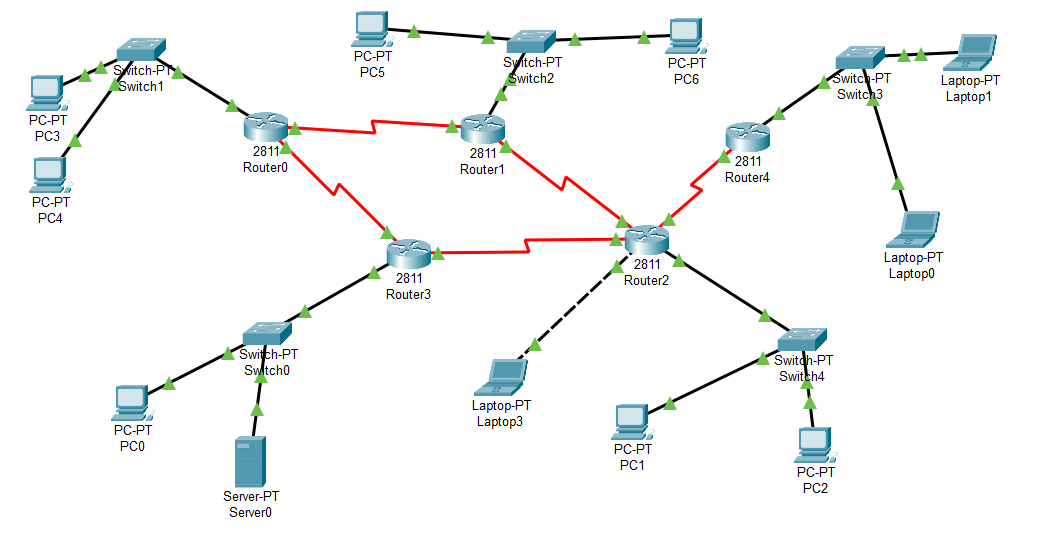
Optimize router convergence times: update time (15 sec), age time (90 sec), and garbage collection time (60 sec).

Use simulation mode and the traffic generator to test communication between PC0 and Laptop0 (e.g., via Ping, TFTP, FTP, and HTTP). Both devices should be able to communicate. Analyze the event list, PDU details, and the protocol's working process.



\

Task 7 OSPF Experiment and Requirements (Cisco Packet Tracer):

****

Task 8 INET Framework Simulation: Run the INET Framework’s RIP and OSPF examples, plot your simulation results, and explain the outputs.

I could no instal the software because of my laptop tech.

Git hub link :- <https://github.com/Mithin-RTU/Telecommunication_software_practical_2>