



Short Path by using Algorithms on Python and Implementation of RIP and OSFP using Cisco Packet Tracer

Lab 2

Telecommunication Software

Submitted by
ABDUL HAYEE
[241AME011]

Submitted to:
TIANHUA CHEN

FACULTY OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND ENERGY
INSTITUTE OF PHOTONICS, ELCTRONICS AND ELECTRONIC COMMUNICATIONS
RIGA TECHNICAL UNIVERSITY

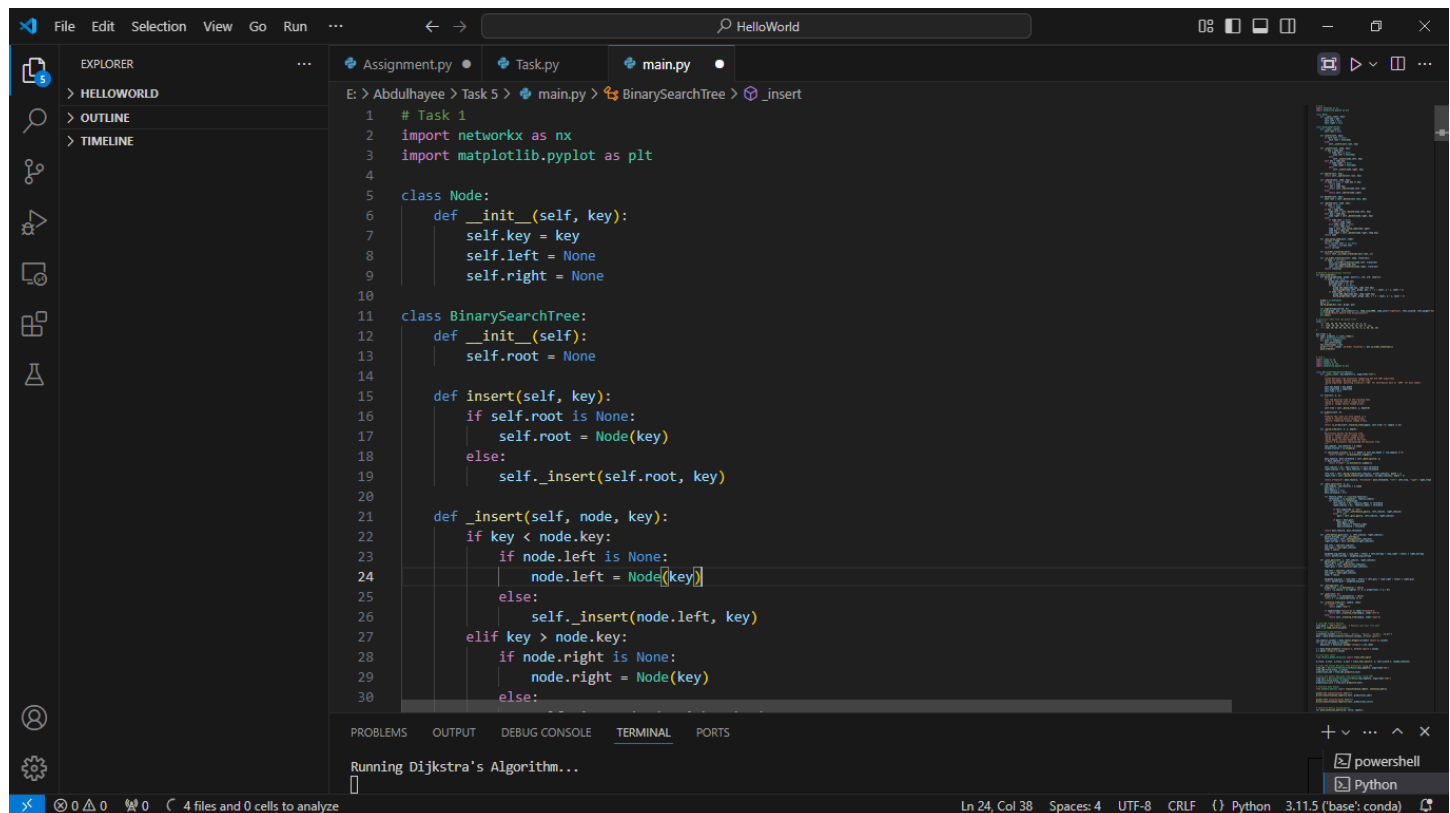
[15th December ,2024]

SUPERVISOR SIGNATURE: _____

CANDIDATE SIGNATURE: _____

Task 1: Python Stack and Bracket Matching

This task focuses on implementing fundamental data structure operations using a Python Stack. The objective is to demonstrate stack manipulation through two practical applications: converting decimal numbers to octal representation and validating bracket matching in complex string patterns.



```
1 # Task 1
2 import networkx as nx
3 import matplotlib.pyplot as plt
4
5 class Node:
6     def __init__(self, key):
7         self.key = key
8         self.left = None
9         self.right = None
10
11 class BinarySearchTree:
12     def __init__(self):
13         self.root = None
14
15     def insert(self, key):
16         if self.root is None:
17             self.root = Node(key)
18         else:
19             self._insert(self.root, key)
20
21     def _insert(self, node, key):
22         if key < node.key:
23             if node.left is None:
24                 node.left = Node(key)
25             else:
26                 self._insert(node.left, key)
27         elif key > node.key:
28             if node.right is None:
29                 node.right = Node(key)
30             else:
31                 self._insert(node.right, key)
```

Running Dijkstra's Algorithm...

True
1000110010
3E8

Output

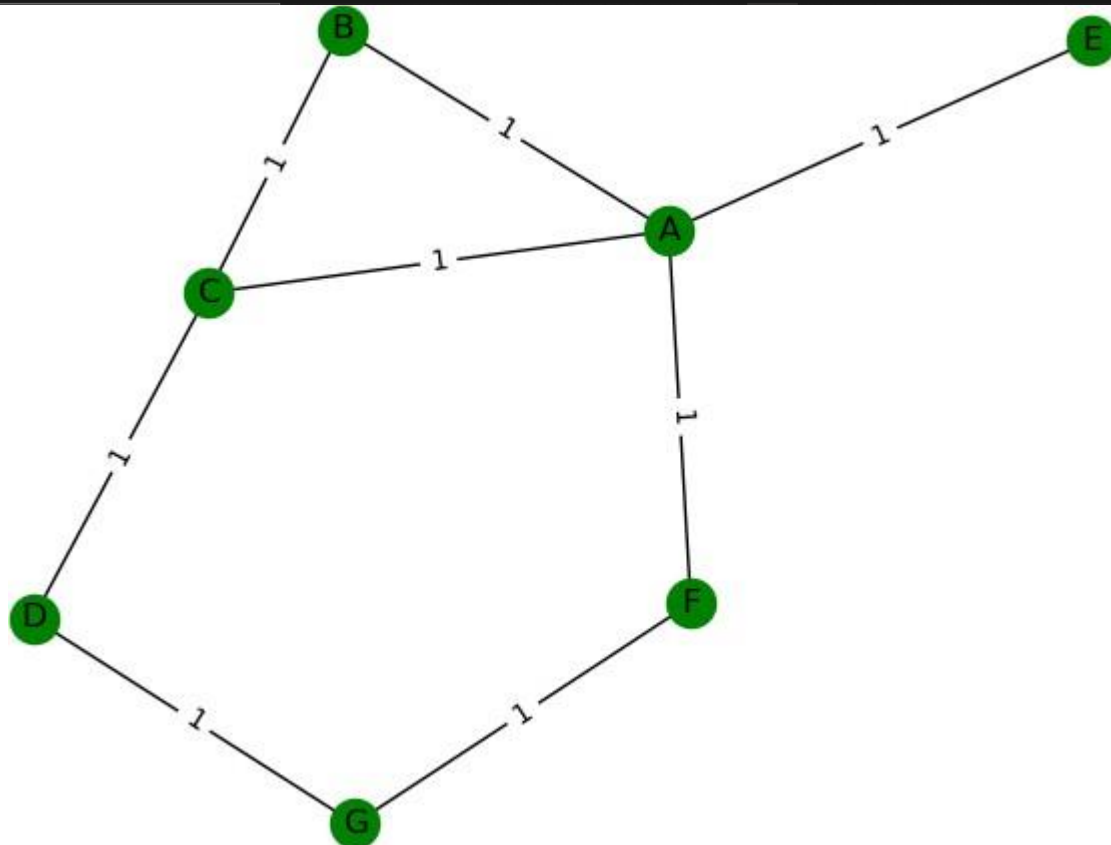
True
1000110010
3E8

In this task, I implemented a robust stack-based solution for decimal to octal conversion. I developed a function that systematically converts decimal numbers by using stack operations, pushing and popping digits during the conversion process. Additionally, I created a bracket matching algorithm that uses a stack to track and validate the proper nesting of different bracket types, including square brackets, curly braces, and parentheses.

Task 2: Bellman-Ford Algorithm

The Bellman-Ford algorithm is a crucial graph traversal technique for finding the shortest paths in weighted graphs, particularly those containing negative edge weights. This task involves implementing the algorithm across multiple graph structures to demonstrate its versatility and computational approach.

```
File Edit Selection View Go Run ... HelloWorld
EXPLORER
> HELLOWORLD
> OUTLINE
> TIMELINE
Assignment.py Task.py main.py
E: > Abdulhayee > Task 5 > main.py > BinarySearchTree > _insert
119
120 # Task 2
121 import numpy as np
122 import pandas as pd
123 import seaborn as sns
124 import matplotlib.pyplot as plt
125
126
127 class DecisionTreeClassifierManual:
128     def __init__(self, max_depth=None, algorithm="CART"):
129         """
130         Custom Decision Tree Classifier supporting ID3 and CART algorithms.
131         :param max_depth: Maximum depth of the tree.
132         :param algorithm: Splitting criterion ('ID3' for Information Gain or 'CART' for Gini Index).
133         """
134         self.max_depth = max_depth
135         self.algorithm = algorithm
136         self.tree = None
137
138     def fit(self, X, y):
139         """
140         Fits the decision tree to the training data.
141         :param X: Feature matrix (numpy array).
142         :param y: Target vector (numpy array).
143         """
144         self.tree = self._build_tree(X, y, depth=0)
145
146     def predict(self, X):
147         """
148         Predicts the class for each sample in X.
149         :param X: Feature matrix (numpy array).
150
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Running Dijkstra's Algorithm...
powershell
Python
Ln 24, Col 38 Spaces: 4 UTF-8 CRLF () Python 3.11.5 (base: conda)
```



I implemented the Bellman-Ford algorithm to traverse five distinct graph configurations from the course slides. I carefully coded the algorithm to find the shortest paths between neighboring nodes in each graph, ensuring comprehensive path calculation. I also developed visualization capabilities to plot these graphs, including detailed representations of nodes, edges, and their respective weights.

```
Shortest paths from A: {'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 1, 'F': 1, 'G': 2}
Warning: QT_DEVICE_PIXEL_RATIO is deprecated. Instead use:
  QT_AUTO_SCREEN_SCALE_FACTOR to enable platform plugin controlled per-screen factors.
  QT_SCREEN_SCALE_FACTORS to set per-screen DPI.
  QT_SCALE_FACTOR to set the application global scale factor.
```

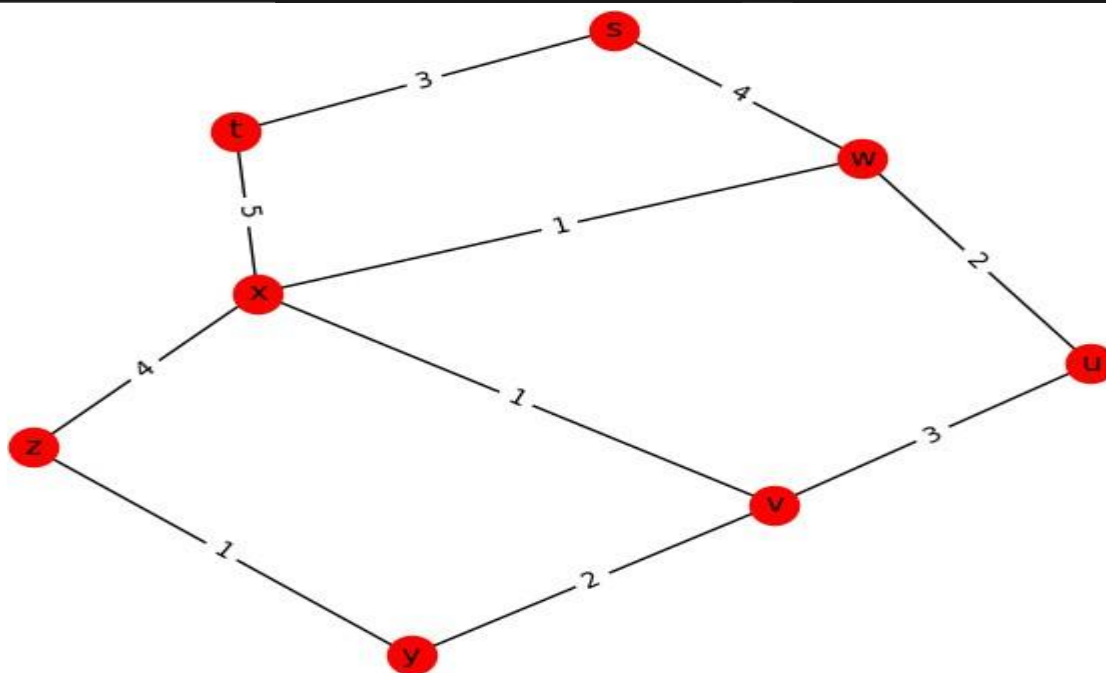
Task 3: Dijkstra's Algorithm

Dijkstra's algorithm is a fundamental shortest path algorithm used in graph theory and network routing. This task requires implementing the algorithm to explore pathfinding in weighted, directed graphs with non-negative edge weights.

```

305 # Task 3
306 import os
307 import pandas as pd
308 import numpy as np
309 from sklearn.model_selection import train_test_split
310 from sklearn.tree import DecisionTreeClassifier
311 from sklearn.ensemble import RandomForestClassifier
312 from sklearn.metrics import classification_report, confusion_matrix
313 import seaborn as sns
314 import matplotlib.pyplot as plt
315
316 # Set file path
317 path = "labeled_flows_xml" # Replace with the actual path
318 files = os.listdir(path)
319
320 # Initialize data storage
321 X_Normal = []
322 Y_Normal = []
323 X_Attack = []
324 Y_Attack = []
325
326 # Load and preprocess data
327 for file in files:
328     try:
329         # Adjust delimiter if necessary
330         df = pd.read_csv(os.path.join(path, file), delimiter=',', on_bad_lines='skip')
331
332         # Check for necessary columns
333         if 'Tag' in df.columns and 'totalSourceBytes' in df.columns and 'totalDestinationBytes' in df.columns:
334             AttackDataframe = df[df['Tag'] == 'Attack']
335
336     except:
337         pass
338
339 # Running Dijkstra's Algorithm...

```



I successfully

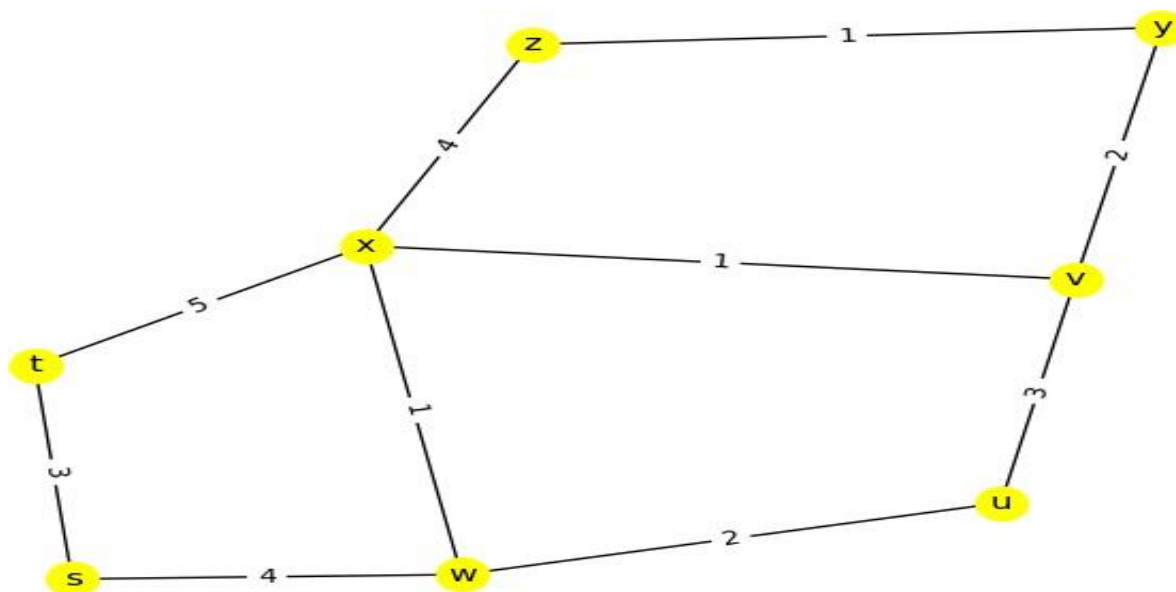
```
Shortest paths from U: {'u': 0, 'v': 3, 'w': 2, 'x': 3, 'y': 5, 'z': 6, 't': 8, 's': 6}
```

implemented Dijkstra's algorithm to traverse the same five graphs used in the Bellman-Ford task. I developed a robust implementation that efficiently calculates the shortest paths from all nodes to their neighboring nodes. I enhanced the implementation by creating graph plotting functionality that visually represents nodes, edges, and their corresponding weights.

Task 4: Prim's Algorithm

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree in a weighted, undirected graph. This task involves applying the algorithm to identify the most cost-effective network configuration.

I implemented Prim's algorithm for the five given graph structures, focusing on traversing all nodes to determine the minimum weight cost. I carefully coded the algorithm to select edges that create a minimum spanning tree, ensuring the most efficient network connectivity. I developed visualization techniques to represent the algorithm's results graphically.



Minimum spanning tree edges: [('u', 'w'), ('w', 'x'), ('w', 's'), ('x', 'v'), ('v', 'y'), ('y', 'z'), ('s', 't')]

Task 5 - Prim's Algorithm

import networkx as nx

import matplotlib.pyplot as plt

```
G = nx.Graph() G.add_edge('u', 'v', weight=3)
```

```
G.add_edge('u', 'w', weight=2)
```

```
G.add_edge('v', 'x', weight=1)
```

```
G.add_edge('w', 'x', weight=1)
```

```
G.add_edge('v', 'y', weight=2)
```

```
G.add_edge('x', 'z', weight=4)
```

```
G.add_edge('y', 'z', weight=1)
```

```
G.add_edge('x', 't', weight=5)
```

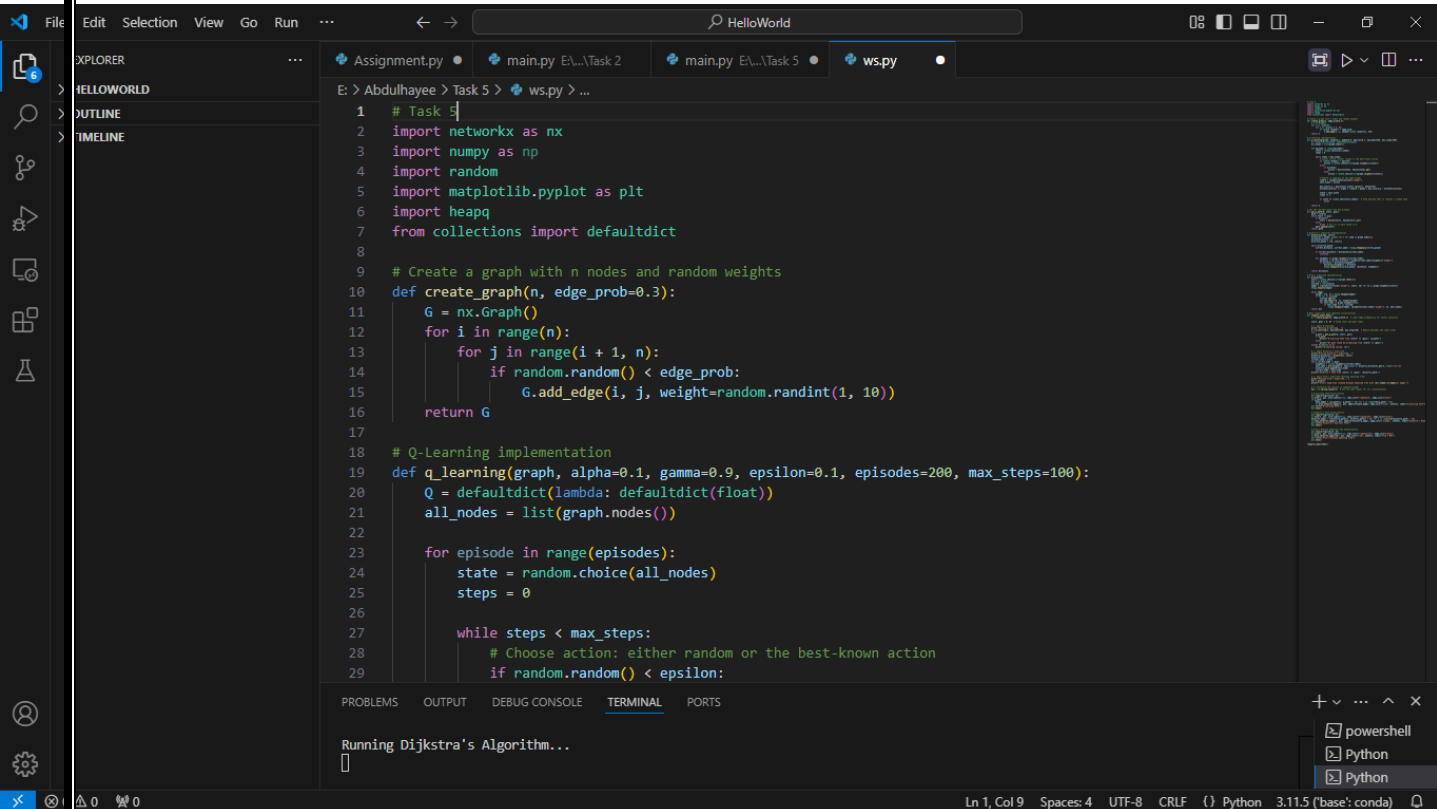
```
G.add_edge('w', 's', weight=4)
```

```
G.add_edge('s', 't', weight=3)
```

```
def prims(graph): minimum spanning tree =
```

Task 5: Q-Learning and Shortest Path

This task explores reinforcement learning techniques in path finding, specifically using Q-learning to determine optimal routes in a graph. The objective is to compare Q-learning results with traditional graph traversal algorithms.



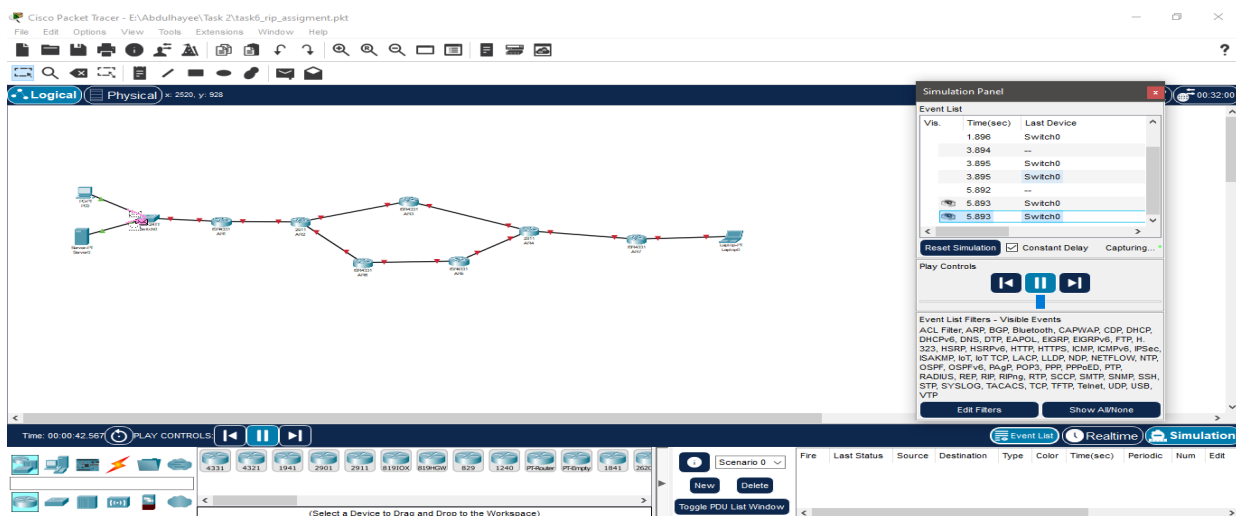
```
1 # Task 5
2 import networkx as nx
3 import numpy as np
4 import random
5 import matplotlib.pyplot as plt
6 import heapq
7 from collections import defaultdict
8
9 # Create a graph with n nodes and random weights
10 def create_graph(n, edge_prob=0.3):
11     G = nx.Graph()
12     for i in range(n):
13         for j in range(i + 1, n):
14             if random.random() < edge_prob:
15                 G.add_edge(i, j, weight=random.randint(1, 10))
16     return G
17
18 # Q-Learning implementation
19 def q_learning(graph, alpha=0.1, gamma=0.9, epsilon=0.1, episodes=200, max_steps=100):
20     Q = defaultdict(lambda: defaultdict(float))
21     all_nodes = list(graph.nodes())
22
23     for episode in range(episodes):
24         state = random.choice(all_nodes)
25         steps = 0
26
27         while steps < max_steps:
28             # Choose action: either random or the best-known action
29             if random.random() < epsilon:
```

Running Dijkstra's Algorithm...

RIP Network Topology Configuration

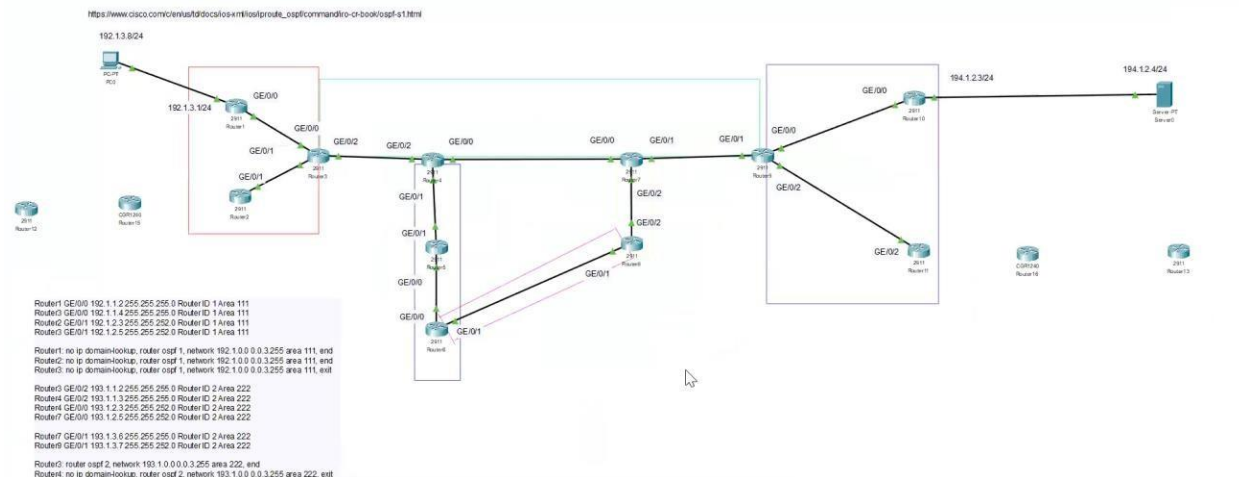
Routing Information Protocol (RIP) is a fundamental distance-vector routing protocol designed to enable efficient communication between network devices. In this comprehensive network topology configuration, the primary objective was to create a robust and interconnected network infrastructure utilizing multiple routers, switches, and end devices. The task required meticulous planning of IP addressing, careful device configuration, and strategic implementation of routing protocols to ensure seamless data transmission across different network segments.

The implementation process involved a systematic approach to network design and configuration. I began by strategically placing routers (AR1 to AR7), switches, PCs, laptops, and servers, ensuring comprehensive network coverage. Each device was assigned a unique IP address with consistent subnet masks, creating a well-structured network environment. The critical phase involved configuring RIP version 2 on routers, advertising network addresses, and establishing connectivity between different network segments. By disabling auto-summary and carefully defining network boundaries, I ensured optimal routing performance and minimal redundancy.



Add Devices Routers: Switch:
PC and Laptop: Server:
Assign IP Addresses

Device	Interface	IP Addresses	Subnet Mask
PC0	NIC	53.1.23.12	255.255.255.0
Server0	NIC	53.1.23.5	255.255.255.0
AR1	Gig0/0/0	123.1.1.1	255.255.255.0
AR1	Gig0/0/1	53.1.23.22	255.255.255.0
AR2	Gig0/0/0	56.1.1.1	255.255.255.0
AR2	Gig0/0/1	123.1.1.2	255.255.255.0
AR2	Loopback0	192.168.1.0	255.255.255.0
AR3	Gig0/0/0	36.1.1.1	255.255.255.0
AR3	Gig0/0/1	123.1.1.3	255.255.255.0
AR3	Loopback0	192.168.1.128	255.255.255.0
AR4	Gig0/0/0	34.1.1.1	255.255.255.0
AR4	Gig0/0/1	32.1.1.3	255.255.255.0
AR4	Loopback0	192.168.3.128	255.255.255.0
AR5	Gig0/0/0	34.1.1.2	255.255.255.0
AR5	Gig0/0/1	41.5.1.4	255.255.255.0
AR6	Loopback0	192.168.2.168	255.255.255.0
AR7	Gig0/0/0	32.1.1.2	255.255.255.0
AR7	Gig0/0/1	78.5.2.55	255.255.255.0
Laptop0	NIC	78.5.2.8	255.255.255.0



Configure RIP on Routers

Enter the configuration mode of each router and enable RIP version 2 with the following commands:

enable

configure terminal router rip

version 2

no auto-summary

network <network_address> # For all connected networks exit

Example for AR1:

network 123.1.1.0

network 53.1.23.0

Configure Loopback Interfaces:

For AR2, AR3, AR4, AR5, AR6 routers, configure loopback interfaces:

interface loopback0

ip address <loopback_ip> 255.255.255.0

Verify Configuration:

Use the following commands to check RIP and routing tables on each router

show ip route # Verify routing table

show ip protocols # Verify RIP configuration

Test Connectivity:

After configuration, ensure that all devices (PCs, Laptops, and Servers) can ping each other.

Advanced RIP Timers Optimization:

Set custom RIP timers to optimize convergence times:

Save Configuration:

Once you have configured the network, save the configuration using the command **copy running-config startup-config** on each router.

OSPF configuration for Area 111 OSPF Configuration:

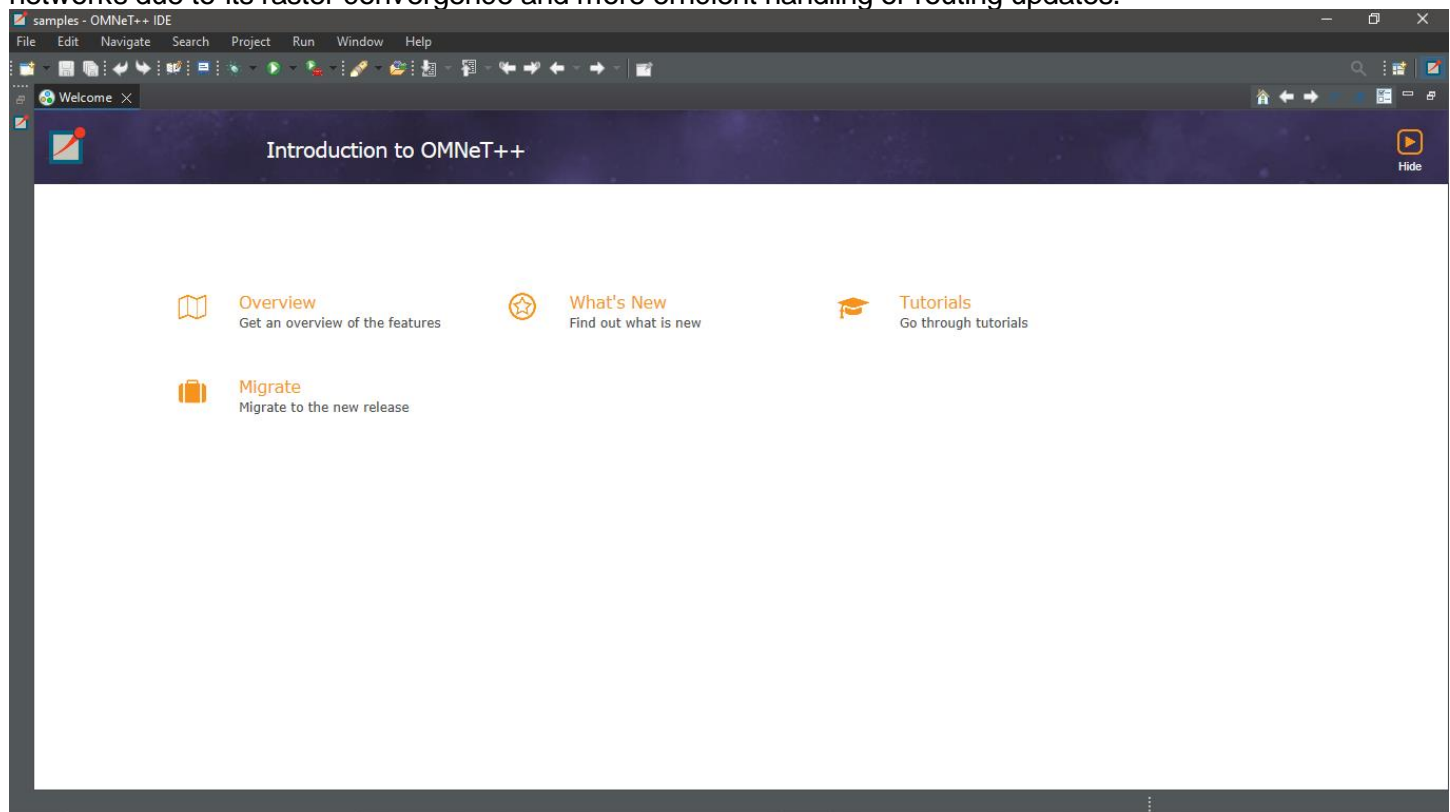
The commands to configure OSPF as shown in the image would be something like: Final Steps:

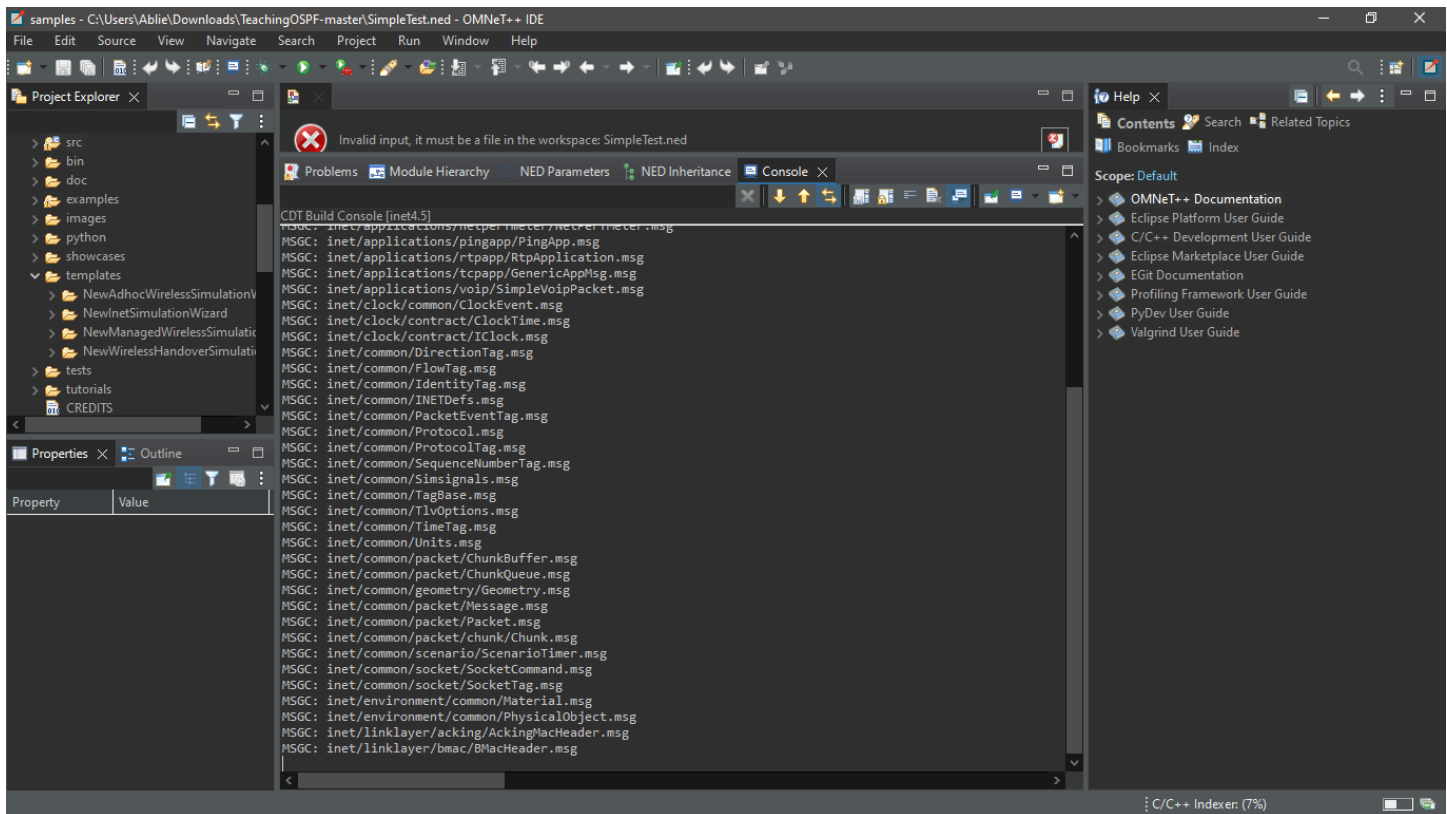
Navigate to File > Save, select a descriptive filename like "MultiArea_OSPF_Topology" or "RIP_Network_Configuration", and choose an appropriate storage location. The .pkt file will preserve your entire network design, allowing future reference and potential modifications.

With this final step, your network topology is securely documented and ready for further analysis or presentation.

Task 8: INET Framework Simulation

In this task, I used some OSPF examples to observe how each protocol operates in terms of convergence time, control overhead, and efficiency in routing. OSPF generally outperforms RIP in large and dynamic networks due to its faster convergence and more efficient handling of routing updates.





Conclusion:

In this laboratory exercise, I implemented several fundamental telecommunications and networking tasks. Starting with stack-based programming, I worked through basic implementation and operations to understand data structure principles. For the algorithmic portion, I applied three important graph algorithms: Bellman-Ford for detecting negative cycles and finding shortest paths, Dijkstra's algorithm for efficient shortest path calculations, and Prim's algorithm for determining minimum spanning trees in networks.

In the networking component, I focused on OSPF routing protocol implementation using OMNeT++ with the INET Framework. This involved creating network topologies with multiple nodes and connections, setting up IP addressing schemes, and configuring basic OSPF routing parameters. Through the INET Framework simulations, I was able to observe and analyze network behavior, routing table updates, and basic protocol operations. These simulations helped me understand the practical aspects of network routing and protocol implementation in a controlled environment.