Experiment-1

Commands to do

ping www.google.com

Theory :-

The ping command is a network diagnostic tool used to:

Test if a host is reachable on a network.
Measure the round-trip time (RTT) for data packets.
It's useful for:

Checking internet connectivity.
Diagnosing network issues.
Identifying packet loss or latency problems.
Stress testing networks.

ipconfig

Theory :-

ipconfig is a command-line utility used in Windows operating systems to display detailed information about the network interfaces (such as Ethernet, Wi-Fi, etc.) on a computer. It stands for "Internet Protocol Configuration."

Here are some of the key things ipconfig can do:

Displaying IP Address Information: It shows the current IP address, subnet mask, and default gateway for all active network interfaces on the computer.

Viewing DNS Information: It provides details about the DNS (Domain Name System) servers that the computer is currently configured to use.

Showing MAC Address: It displays the physical address (Media Access Control address) of the network adapter.

Checking DHCP Configuration: It indicates whether the network interface is configured to obtain an IP address automatically from a DHCP (Dynamic Host Configuration Protocol) server.

Releasing and Renewing IP Addresses: It can release the current IP address assigned to a network interface and request a new one from the DHCP server.

Flushing DNS Cache: It can clear the DNS resolver cache, which stores recent DNS lookups.

Displaying Additional Information: Depending on the options used with ipconfig, it can provide additional information, such as tunnel adapter details, etc.

tracert www.google.com

Theory :-

The tracert (or traceroute in some systems) command is a network diagnostic tool used to trace the route that data packets take from your computer to a specified destination, like a website or server. Here's a short explanation:

Tracing Packet Route: tracert sends a series of packets towards the target, each with an increasing time-to-live (TTL) value.

Hop-by-Hop Information: As each packet traverses the network, intermediate routers are required to decrease the TTL. When the TTL reaches zero, the router sends an error message back. This allows tracert to identify each hop the packet takes.

Displaying Route and Delays: tracert then displays a list of all the routers (or "hops") the packet passed through, along with their IP addresses and round-trip times. This information helps diagnose network issues and measure latency.

tracert is useful for:

Identifying network congestion or delays.
Locating network failures or outages.
Troubleshooting connectivity problems.
Verifying if data is taking an expected path.

nslookup www.google.com

Theory:-

The nslookup command is a network tool used to query DNS (Domain Name System) servers for information about domain names, IP addresses, and related DNS records. Here's a short explanation:

Resolving DNS Information: nslookup helps resolve and display information about domain names and their associated IP addresses.

Querying DNS Servers: It allows you to interactively query specific DNS servers or use the default DNS server configured on your system.

Checking DNS Records: You can use nslookup to check various DNS records like A (address), MX (mail exchange), or PTR (reverse DNS) records for a given domain.

Diagnosing DNS Issues: It's a valuable tool for troubleshooting DNS-related problems and verifying DNS configurations.

netstat

Theory :-

The netstat command is a network utility used to display information about network connections, routing tables, and network interface statistics on a computer. Here's a short explanation:

Displaying Network Connections: netstat shows a list of all active network connections, including the local and remote addresses, as well as the state of the connection (listening, established, etc.).

Viewing Routing Information: It provides information about the computer's routing table, which contains details about how data packets are directed through the network.

Showing Network Interface Statistics: netstat can display statistics for each network interface, including data transfer rates, error counts, and more.

Monitoring Open Ports: It can be used to identify which ports are open and listening for incoming connections.

Diagnosing Network Issues: netstat is a useful tool for diagnosing network problems and understanding how data is flowing in and out of a computer.

arp -a

Theory:-

The command arp -a displays the current ARP (Address Resolution Protocol) table for a device. ARP is used to map an IP address to a physical MAC (Media Access Control) address on a local network.

In short, arp -a shows a list of known IP addresses and their corresponding MAC addresses that the device has recently communicated with. This information is important for establishing network connections.

Experiment - 2

Theory

1) Fiber-optic cable

- These cables mostly consist of center glass and different layers of protective materials surrounding them. Fiber-Optic cabling transmits light in place of electronic signals, which removes the issue of electrical interference. This makes it an ideal selection for environments that contain a large amount of electrical interference.

- This type of network cable offers the ability to transmit signals over longer distances. It also provides the ability to carry information in a faster space.

Two types of fiber-optic cables are:

- Single-mode fiber (SMF)– This type of fiber optic cable uses only a single ray of light to carry data. Used for larger distance wiring.

- Multimode fiber (MMF)– This type of fiber-optic uses multiple rays of light to carry data. Less Expensive than SMF.

2) Coaxial cable

- Coaxial Cable is standard for 10 Mbps Ethernet cables. These types of cables consist of an inner copper wire cover with insulation and other shielding.

- It has a plastic layer that offers insulation between the braided metal shield and the center conductor. Coaxial cabling has a single copper conductor in its center.

- Types of coaxial cable are a) RG58 b) RG8 c) RG6, and d) RG59

3) Twisted-pair cable

- Twisted-Pair Cabling is a type of cabling in which pairs of wires are twisted together to stop electromagnetic interference (EMI) from other wire pairs.

- Two types of twisted pair cables are a) Unshielded Twisted Pair b)

Shielded Twisted pair

4) Connectors

• Connectors in networking are essential for establishing physical connections between devices.

• Common connectors include RJ-45 for Ethernet, fiber optic connectors like SC and LC, BNC for video transmission, USB for peripheral devices, and HDMI for audio/video. Coaxial connectors are used in cable and satellite TV connections.

• Choosing the right connector ensures reliable data transmission.

5) Hubs

• A hub is a basic networking device that connects multiple Ethernet devices together.

• It operates at the physical layer and broadcasts data to all connected devices. Hubs amplify and regenerate signals but create a single collision domain, leading to performance degradation.

• They have limited bandwidth and have been largely replaced by switches in modern networks.

6) Switches

• Switches are networking devices that connect multiple devices in a LAN and facilitate communication between them.

• They operate at the data link layer and use packet switching to forward data based on MAC addresses.

• Switches support unicast, broadcast, and multicast traffic and can also implement VLANs for network segmentation.

• Switches provide better performance and management capabilities compared to hubs.

7) Bridges

• Bridges are networking devices that connect LANs or network Segments.

• They operate at the data link layer and forward traffic based on MAC addresses.

• Bridges segment networks and improve performance by reducing Congestion.

• While switches have largely replaced bridges, the term "bridge" is sometimes used interchangeably with switches.

8) Network Interface Card

• A Network Interface Card (NIC), also known as a network adapter or network card, is a hardware component that allows a computer or other device to connect to a network.

• It provides the physical interface between the device and the network medium, enabling communication with other devices on the Network.

9) Gateway and Firewalls

• Gateway, is a device or software component that serves as an entry or exit point for data traffic between different networks.

• It acts as a bridge, facilitating communication and data transfer between networks that use different protocols, addressing schemes, or communication technologies.

• Firewalls are network security devices or software that monitor and control incoming and outgoing network traffic based on predetermined security rules.

• They act as a barrier between trusted internal networks and untrusted external networks, providing protection against unauthorized access, malicious activities, and potential threats.

Experiment - 3

1) Bus Topology

In this topology, all devices are connected to a single communication line, often called a "bus."

Each device shares the same communication medium and can transmit and receive data.

However, the entire network can be affected if there is a break or failure in the main communication line.

2) Mesh Topology:

A mesh topology provides a direct point-to-point connection between every device in the network.

Each device has a dedicated connection to every other device, resulting in redundant paths.

This redundancy enhances fault tolerance and ensures that if one path fails, data can still be transmitted through alternative routes.

Mesh topologies can be fully connected (every device connected to every other device) or partially connected (selected devices have direct connections).

3) Ring Topology:

In a ring topology, devices are connected in a closed-loop or ring configuration. Each device is connected directly to two other devices, forming a circular pathway for data transmission.

Data travels in one direction around the ring, passing through each device until it reaches its destination.

Ring topologies are relatively easy to install and manage, but if one device or connection fails, the entire network can be disrupted.

4) Star Topology:

In a star topology, all devices are connected directly to a central hub or switch. The central hub acts as a central point of communication, facilitating data exchange between devices.

If one device wants to communicate with another, it sends the data to the central hub, which then forwards it to the intended recipient.

Star topologies are easy to set up and manage, and a failure in one device or connection typically does not affect the rest of the network. However, the central hub is a critical point of failure, and if it malfunctions, the entire network may be affected.

Experiment - 4

Networks play a vital role in modern networking by enabling communication and resource sharing among devices. Various types of networks serve different purposes to meet the diverse needs of users and organizations. The most prominent types of networks include Local Area Networks (LANs), Wide Area Networks (WANs), Metropolitan Area Networks (MANs), Storage Area Networks (SANs), Virtual Private Network (VPN), Wireless Network, and Cloud Network.

Three most common types of networks:
• Local Area Network (LAN)
• Wide Area Network (WAN)
• Metropolitan Area Network (MAN)

LANs

• A LAN is a network infrastructure that spans a small geographical area.
• Interconnect end devices in a limited area.
• Administered by a single organization or individual.
• Provide high-speed bandwidth to internal devices.

WANs

• A WAN is a network infrastructure that spans a wide geographical area.
• Interconnect LANs over wide geographical areas.
• Typically administered by one or more service providers.
• Typically provide slower speed links between LANs.

MANs

• A MAN is a network that spans a larger geographical area than a LAN.
• It connects multiple Local Area Networks over high speed links such as fiber optic cables.
• A MAN can be built by a single organization to connect multiple offices spanning a few building that are not very far from each other or it might be operated as public utility.

Experiment - 5

The Internet

The internet is a worldwide collection of interconnected LANs and WANs.
LANs are connected to each other using WANs.
WANs may use copper wires, fiber optic cables, and wireless transmissions.
The internet is not owned by any individual or group. IETF

Intranets and Extranets

An intranet is a private collection of LANs and WANs internal to an organization that is meant to be accessible only to the organizations members or others with authorization.

An organization might use an extranet to provide secure access to their network for individuals who work for a different organization that need access to their data on their network.

Experiment - 6

Algorithm Go-Back-N:-

Short Algorithm for Go-Back-N ARQ:

1. **Input**:
   - `tf`: Total number of frames to be transmitted.
   - `N`: Window size, representing the maximum number of frames that can be sent before waiting for an acknowledgment.
   - `i`: Starting frame number.

2. **Transmission**:
   - Initialize `tt` to 0 (total transmitted frames counter).

   - Loop while `i <= tf`:
     - Initialize `z` to 0 (acknowledged frames counter).

     - Transmit frames `i` to `min(i + N, tf + 1)`.

     - For each frame `k` in the current window:
       - Simulate acknowledgment (`f`):
         - If acknowledged (`f == 0`), increment `z`.
         - If not acknowledged (`f == 1`), initiate retransmission.

     - Increment `i` by `z`.
     - Increment `tt` by the number of frames transmitted in the current window.

3. **Output**:
   - Return `tt` (total transmitted frames).

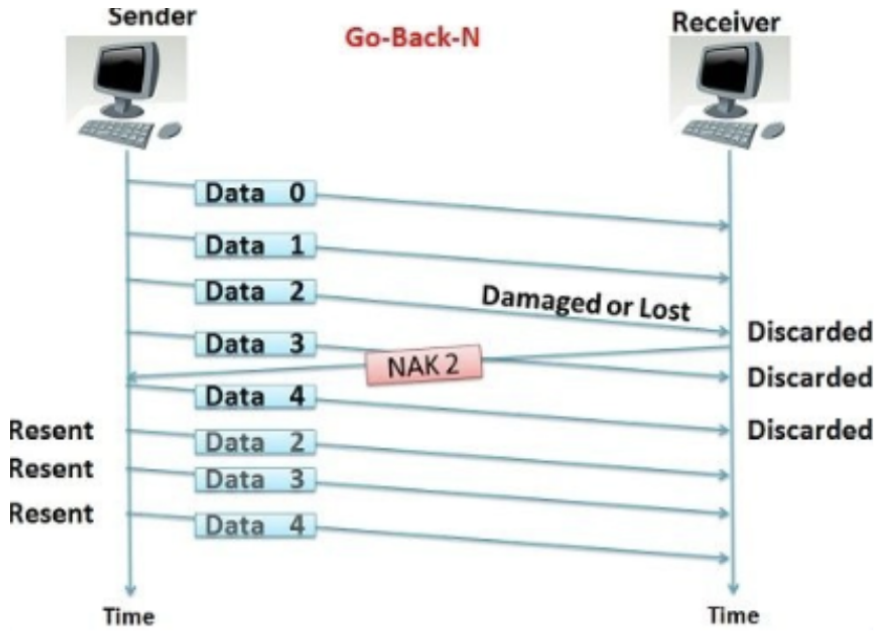
4. **Main Function**:
   - Prompt user for `tf` and `N`.

   - Set `i` to 1.

- Call Transmission function with `i`, `N`, and `tf`.

- Print total transmitted frames.

5. **Execution**:
   - User inputs total frames `tf` and window size `N`.
   - Frames are transmitted and acknowledged or retransmitted.
   - Total transmitted frames are displayed.



Theory Go-Back-N :-

In this protocol, the frames are sent again if they are lost while being transmitted.
The size of the receiver buffer will be one, while the size of the sender buffer is predefined.
The receiver cancels the frame if it gets corrupted; when the sender's timer for an acknowledgment to be received expires, the sender sends the same frame again without moving on with different frames.

Code for Go-Back-N :-

```
import random
import time

def transmission(i, N, tf):
    tt = 0
    while i <= tf:
        z = 0
        for k in range(i, min(i + N, tf + 1)):
```

```
            print(f"Sending Frame {k}...")
            tt += 1
        for k in range(i, min(i + N, tf + 1)):
            f = random.randint(0, 1)
            if not f:
                print(f"Acknowledgment for Frame {k}...")
                z += 1
            else:
                print(f"Timeout!! Frame Number: {k} Not Received")
                print("Retransmitting Window...")
                break
        print()
        i += z
    return tt

def main():
    tf = int(input("Enter the Total number of frames: "))
    N = int(input("Enter the Window Size: "))
    i = 1
    tt = transmission(i, N, tf)
    print(f"Total number of frames which were sent and resent are: {tt}")

if __name__ == "__main__":
    main()
```

Algorithm Selective Repeat :-

1. **Input**:
   - `N`: Window size, representing the maximum number of frames that can be sent before waiting for an acknowledgment.
   - `tf`: Total number of frames to be transmitted.

2. **Transmitting Frames**:
   - Initialize `tt` (total transmitted frames counter) to 0.
   - Create a list `frames` to keep track of frame statuses (0 for unsent, 1 for acknowledged).

   - Loop over frames `i` from 1 to `tf`:
     - Initialize an empty list `acknowledgment_messages`.

     - For each frame `k` in the current window:
       - If the frame is unsent, print a message indicating the transmission of frame `k`.

     - For each frame `k` in the current window:
       - If the frame is unsent, simulate acknowledgment (`f`):
         - If acknowledged (`f == 0`), append acknowledgment message and update frame status.

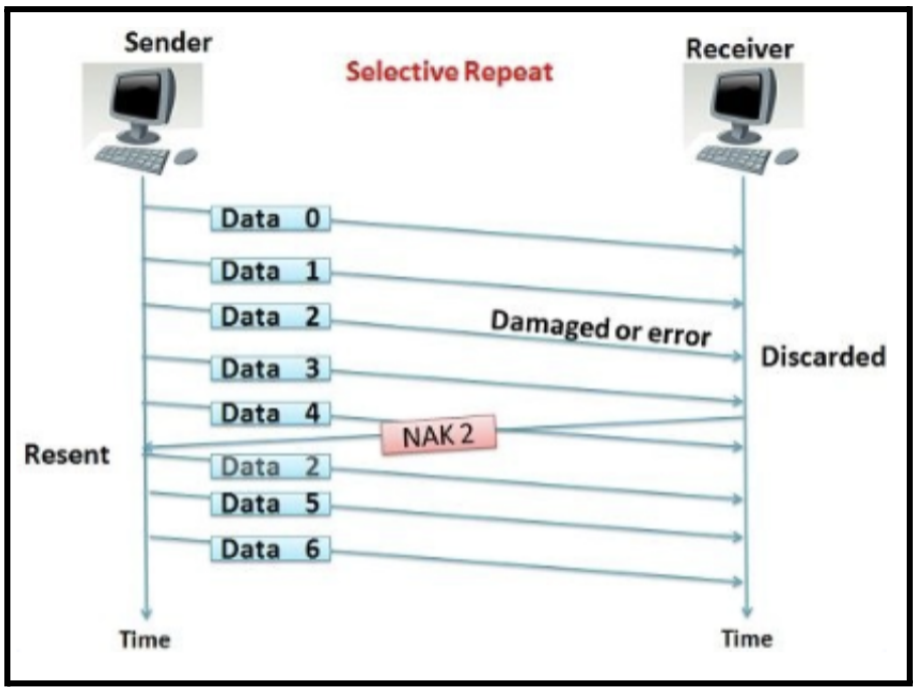
- Print acknowledgment messages for frames sent.

- Increment `i` by 1.
- Increment `tt` by the number of frames transmitted in the current window.

3. **Output**:
   - Return `tt`.

4. **Main Function**:
   - Prompt user for `tf` and `N`.
   - Call Transmission function with `N` and `tf`.
   - Print total transmitted frames.

5. **Execution**:
   - User inputs total frames `tf` and window size `N`.
   - Frames are transmitted and acknowledged.
   - Total transmitted frames are displayed.



Theory Selective Repeat :-

Selective repeat automatic repeat request works with the data link layer and uses the sliding window method to send frames.

Only the corrupted frame during transmission is sent again in its execution, while the further requests get acknowledged.

The window size in both the sender and receiver is the same.

This protocol helps in saving on bandwidth as compared to Go Back-N ARQ, which processed the whole frames again without selectively choosing to send the faulty frames.

Code for Selective Repeat :-

```
        import random

def transmission(N, tf):
    tt = 0
    frames = [0] * (tf + 1)

    for i in range(1, tf + 1, N):
        z = 0
        acknowledgment_messages = []  # Store acknowledgment messages

        # First loop to send frames and record acknowledgment status
        for k in range(i, min(i + N, tf + 1)):
            if frames[k] == 0:
                print(f"Sending Frame {k}...")
                tt += 1

        for k in range(i, min(i + N, tf + 1)):
            if frames[k] == 0:
                f = random.randint(0, 1)
                if not f:
                    acknowledgment_messages.append(f"Acknowledgment for Frame {k}
received.")
                    z += 1
                    frames[k] = 1
                else:
                    acknowledgment_messages.append(f"Acknowledgment for Frame {k} not
received.")
                    frames[k] = 2  # Mark frame for retransmission

        # Print acknowledgment messages for frames sent
        for msg in acknowledgment_messages:
            print(msg)

        # Print retransmission messages
        for k in range(i, min(i + N, tf + 1)):
            if frames[k] == 2:
                print(f"Retransmitting Frame {k}...")
                f = random.randint(0, 1)
                if not f:
                    acknowledgment_messages.append(f"Acknowledgment for Frame {k}
(retransmitted) received.")
```

```
                frames[k] = 1

        # Print acknowledgment messages for retransmitted frames that were not received
        for k in range(i, min(i + N, tf + 1)):
            if frames[k] == 2:
                print(f"Acknowledgment for Frame {k} (retransmitted) not received.")

        print()

    return tt

def main():
    tf = int(input("Enter the Total number of frames: "))
    N = int(input("Enter the Window Size: "))
    tt = transmission(N, tf)
    print("Total number of frames which were sent and acknowledged are:", tt)

if __name__ == "__main__":
    main()
```

Experiment - 7 :-

Links :-

http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html

http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html

http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html

http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wiresharkfile5.html

Username :- wireshark-students Password :- network

Experiment - 8 :-

Code:-

```
import heapq
```

```
def dijkstra(graph, start):
    # Initialize distance and predecessor dictionaries
    distance = {vertex: float('infinity') for vertex in graph}
    previous = {vertex: None for vertex in graph}

    # Set the distance to the starting node to 0
    distance[start] = 0

    # Create a priority queue to store vertices and their distances
    priority_queue = [(0, start)]

    while priority_queue:
        # Get the vertex with the smallest distance
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # If the current distance is greater than the recorded distance, skip
        if current_distance > distance[current_vertex]:
            continue

        # Explore the neighbors of the current vertex
        for neighbor, weight in graph[current_vertex].items():
            distance_to_neighbor = distance[current_vertex] + weight

            # If the new distance is shorter, update the distance and predecessor
            if distance_to_neighbor < distance[neighbor]:
                distance[neighbor] = distance_to_neighbor
                previous[neighbor] = current_vertex
                heapq.heappush(priority_queue, (distance_to_neighbor, neighbor))

    return distance, previous

# Example usage:
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
start_vertex = 'A'
distances, predecessors = dijkstra(graph, start_vertex)

# Print the shortest distances from the starting vertex
for vertex, distance in distances.items():
    print(f"Shortest distance from {start_vertex} to {vertex} is {distance}")

# Print the shortest path from the starting vertex to a specific vertex
end_vertex = 'D'
```

```
path = []
while end_vertex:
    path.insert(0, end_vertex)
    end_vertex = predecessors[end_vertex]

print(f"Shortest path from {start_vertex} to {path[-1]}: {' -> '.join(path)}")

graph3 = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1},
    'C': {'D': 3},
    'D': {'E': 2},
    'E': {'F': 1},
    'F': {'D': 5}
}

start_vertex3 = 'A'
distances3, predecessors3 = dijkstra(graph3, start_vertex3)

print("Shortest distances from", start_vertex3, ":", distances3)


end_vertex3 = 'F'
path3 = []
while end_vertex3:
    path3.insert(0, end_vertex3)
    end_vertex3 = predecessors3[end_vertex3]

print("Shortest path from", start_vertex3, "to", path3[-1], ":", ' -> '.join(path3))
# Expected output:
# Shortest path from A to F: A -> B -> C -> D -> E -> F
import networkx as nx
import matplotlib.pyplot as plt

# Test Case 1: Basic Example
graph1 = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# Create a directed graph
G1 = nx.DiGraph()
for node, edges in graph1.items():
    G1.add_node(node)
    for neighbor, weight in edges.items():
```

```
        G1.add_edge(node, neighbor, weight=weight)

# Visualize the graph
pos1 = nx.spring_layout(G1)
labels1 = nx.get_edge_attributes(G1, 'weight')
nx.draw_networkx_nodes(G1, pos1, node_size=700)
nx.draw_networkx_edges(G1, pos1)
nx.draw_networkx_labels(G1, pos1)
nx.draw_networkx_edge_labels(G1, pos1, edge_labels=labels1)
plt.title("Test Case 1: Basic Example")
plt.axis('off')
plt.show()

# Test Case 2: Directed Graph
graph2 = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'A': 4, 'D': 1},
    'D': {}
}

# Create a directed graph
G2 = nx.DiGraph()
for node, edges in graph2.items():
    G2.add_node(node)
    for neighbor, weight in edges.items():
        G2.add_edge(node, neighbor, weight=weight)

# Visualize the graph
pos2 = nx.spring_layout(G2)
labels2 = nx.get_edge_attributes(G2, 'weight')
nx.draw_networkx_nodes(G2, pos2, node_size=700)
nx.draw_networkx_edges(G2, pos2)
nx.draw_networkx_labels(G2, pos2)
nx.draw_networkx_edge_labels(G2, pos2, edge_labels=labels2)
plt.title("Test Case 2: Directed Graph")
plt.axis('off')
plt.show()

# Test Case 3: Disconnected Graph
graph3 = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1},
    'C': {'D': 3},
    'D': {'E': 2},
    'E': {'F': 1},
    'F': {'D': 5}
```

```
}

# Create an undirected graph
G3 = nx.Graph()
for node, edges in graph3.items():
    G3.add_node(node)
    for neighbor, weight in edges.items():
        G3.add_edge(node, neighbor, weight=weight)

# Visualize the graph
pos3 = nx.spring_layout(G3)
labels3 = nx.get_edge_attributes(G3, 'weight')
nx.draw_networkx_nodes(G3, pos3, node_size=700)
nx.draw_networkx_edges(G3, pos3)
nx.draw_networkx_labels(G3, pos3)
nx.draw_networkx_edge_labels(G3, pos3, edge_labels=labels3)
plt.title("Test Case 3: Disconnected Graph")
plt.axis('off')
plt.show()
```

Algorithm :-

Dijkstra's Algorithm:

Input:

graph: A weighted directed or undirected graph represented as a dictionary of dictionaries. The outer dictionary contains nodes as keys, and the inner dictionaries have neighboring nodes as keys and edge weights as values.

start: The starting node for the algorithm.

Initialization:

Create two dictionaries: distance and previous.

distance: Initialize all nodes with a distance of infinity, except the starting node which is set to 0.

previous: Initialize all nodes with a None predecessor.

Priority Queue:

Create a priority queue (min-heap) called priority_queue to keep track of nodes and their distances. Initialize it with the start node and its distance.

Main Loop:

While the priority_queue is not empty:

Pop the node with the smallest distance from the priority_queue.

For each neighbor of the current node:

Calculate the tentative distance to the neighbor through the current node.

If this distance is less than the recorded distance, update the distance and predecessor.

Add the neighbor and its updated distance to the priority_queue.

Output:

After the loop, return the distance and previous dictionaries.

Usage:

You can then use the output to find the shortest path from the starting node to any other node.