Computer Communication Networks Lab File

Group Members:

Vedant Milind Athavale (181090071) (+91 9004188281) (vmathavale_b18@et.vjti.ac.in)
Aditi Patil (181091003) (+91 9326566734) (aapatil_b18@et.vjti.ac.in)
Tejas Adhikari (181090068) (+91 9987117186) (tsadhikari_b18@et.vjti.ac.in)
Dhruv Singh (181090064) (+91 9527723605) (dysingh_b18@et.vjti.ac.in)

EXPERIMENT 1

Aim:

To study the stuffing and de-stuffing of the entered binary code. By stuffing a 0 after five consecutive 1’s.

Requirements:

MATLAB R2020a

Theory:

Bit stuffing (also known as **positive justification**) is the insertion of non-information bits into data. Stuffed bits should not be confused with overhead bits. Bit stuffing is used for various purposes, such as for bringing bit streams that do not necessarily have the same or rationally related bit rates up to a common rate, or to fill buffers or frames. The location of the stuffing bits is communicated to the receiving end of the data link, where these extra bits are removed to return the bit streams to their original bit rates or form. Bit stuffing may be used to synchronize several channels before multiplexing or to rate-match two single channels to each other.

Another use of bit stuffing is for run length limited coding: to limit the number of consecutive bits of the same value in the data to be transmitted. A bit of the opposite value is inserted after the maximum allowed number of consecutive bits. Since this is a general rule the receiver doesn't need extra information about the location of the stuffing bits in order to do the de-stuffing. This is done to create additional signal transitions to ensure reliable reception or to escape special reserved code words such as frame sync sequences when the data happens to contain them. Bit stuffing does not ensure that the payload is intact (*i.e.* not corrupted by transmission errors); it is merely a way of attempting to ensure that the transmission starts and ends at the correct places. Error detection and correction techniques are used to check the frame for corruption after its delivery and, if necessary, the frame will be re-sent.

<u>Algorithm:</u>

Stuffing:

1. Start
2. Initialize the array for transmitted stream with the special bit pattern.
3. Get the bit stream to be transmitted in to the array.
4. Check for five consecutive ones and if they occur, stuff a bit 0
5. Display the data transmitted as it appears on the data line after

De–stuffing:

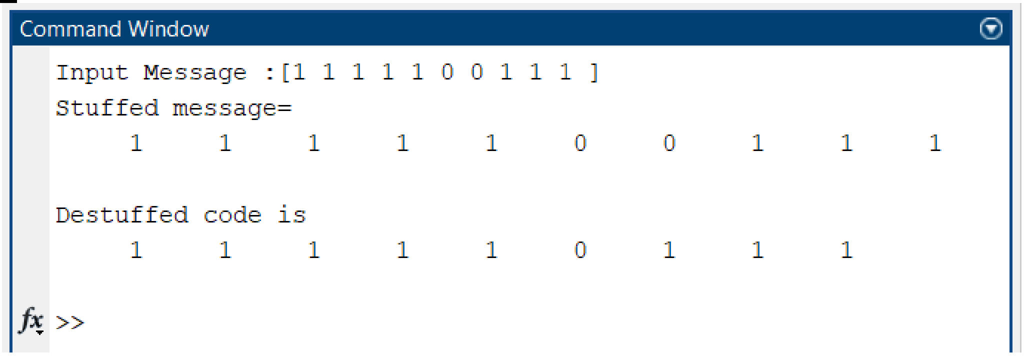
1. Copy the transmitted data to another array after detecting the stuffed bits
2. Display the received bit stream
3. Stop

<u>Code:</u>

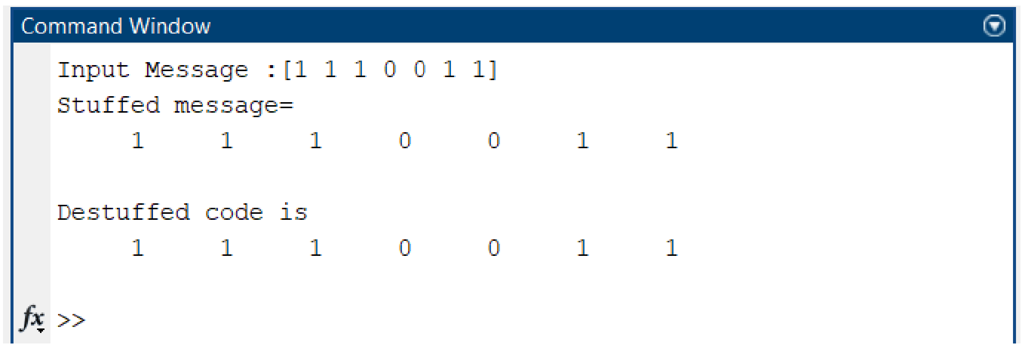
```
clc;
msg=input('Input Message :');
count=0;
stuffcount=0;
[M,N]=size(msg);
for i=1:1:N+stuffcount

    if msg(i)==1
        count=count+1;
    elseif msg(i)==0
        count=0;
    end
    if count==5
        msg=[msg(1:i) 0 msg(i+2:end)];
        stuffcount=stuffcount+1;
        count=0;
    else
        continue
    end

end
disp("Stuffed message=")
disp(msg)

[M,N]=size(msg);
```

```
des = [];
i=1;
j=1;
count=0;
while j <= N-stuffcount
    if(msg(i) == 1)
        count = count + 1;
    elseif(msg(i) == 0)
        count = 0;
    end
    des(j) = msg(i);
    if(count == 5)
        i = i + 1;
        count = 0;
    end
    j = j + 1;
    i = i + 1;
end

disp("Destuffed code is")
disp(des)
```

Output:



```
Command Window
Input Message :[1 1 1 1 1 0 0 1 1 1 ]
Stuffed message=
     1     1     1     1     1     0     0     1     1     1

Destuffed code is
     1     1     1     1     1     0     1     1     1

>>
```



```
Command Window
Input Message :[1 1 1 0 0 1 1]
Stuffed message=
     1     1     1     0     0     1     1

Destuffed code is
     1     1     1     0     0     1     1

>>
```

Conclusion:

We studied the stuffing and de-stuffing of the entered binary code. By stuffing a 0 after five consecutive 1’s using the MATLAB.

EXPERIMENT 2

Aim:

To study Cyclic Redundancy Code, and to understand it's use in detecting errors.

Requirements:

MATLAB

Theory:

When bits are transmitted over the computer network, they are subject to get corrupted due to interference and network problems. The corrupted bits lead to spurious data being received by the receiver and are called errors.
Error detection techniques are responsible for checking whether any error has occurred or not in the frame that has been transmitted via network. It does not take into account the number of error bits and the type of error.
For error detection, the sender needs to send some additional bits along with the data bits. The receiver performs necessary checks based upon the additional redundant bits. If it finds that the data is free from errors, it removes the redundant bits before passing the message to the upper layers.
There are three main techniques for detecting errors in data frames: Parity Check, Checksum and Cyclic Redundancy Check (CRC).

Cyclic Redundancy Check (CRC) is a block code invented by W. Wesley Peterson in 1961. It is commonly used to detect accidental changes to data transmitted via telecommunications networks and storage devices.

CRC involves binary division of the data bits being sent by a predetermined divisor agreed upon by the communicating system. The divisor is generated using polynomials. So, CRC is also called polynomial code checksum.

Encoding using CRC:
The communicating parties agrees upon the size of message block and the CRC divisor. For example, the block chosen may be CRC (7, 4), where 7 is the total length of the block and 4 is the number of bits in the data segment. The divisor chosen may be 1011.

The sender performs binary division of the data segment by the divisor.
It then appends the remainder called CRC bits to the end of data segment. This makes the resulting data unit exactly divisible by the divisor.
Decoding:
The receiver divides the incoming data unit by the divisor.
If there is no remainder, the data unit is assumed to be correct and is accepted.
Otherwise, it is understood that the data is corrupted and is therefore rejected. The receiver may then send an erroneous acknowledgement back to the sender for retransmission.

<u>**Algorithm:**</u>

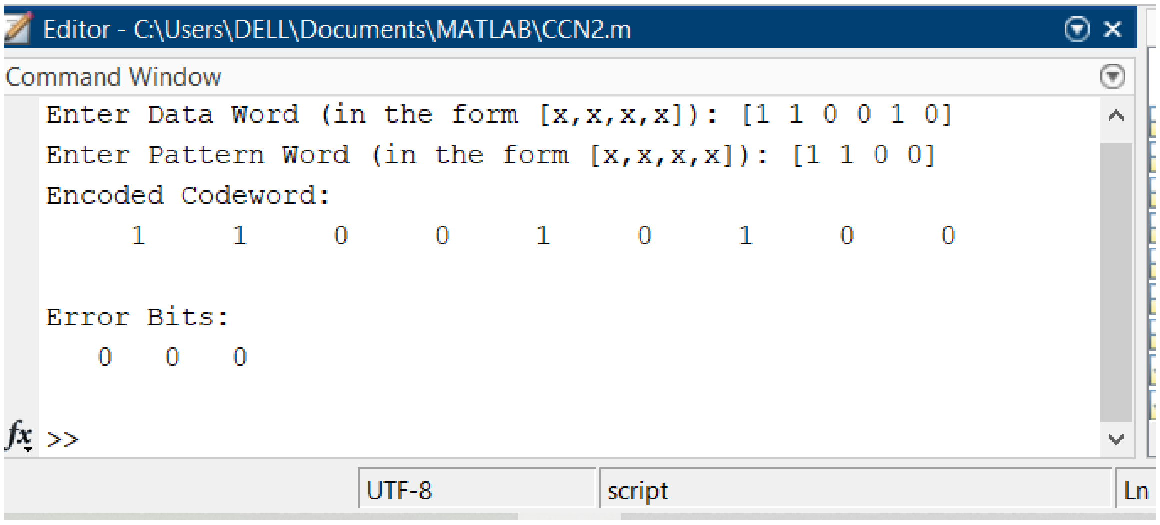
- ***Algorithm for Encoding using CRC***
    - The communicating parties agrees upon the size of message, $M(x)$ and the generator polynomial, $G(x)$.
    - If r is the order of $G(x)$, r, bits are appended to the low order end of $M(x)$. This makes the block size bits, the value of which is $x^r M(x)$.
    - The block $x^r M(x)$ is divided by $G(x)$ using modulo 2 division.
    - The remainder after division is added to $x^r M(x)$ using modulo 2 addition. The result is the frame to be transmitted, $T(x)$. The encoding procedure makes exactly divisible by $G(x)$.
- ***Algorithm for Decoding using CRC***
    - The receiver divides the incoming data frame T(x) unit by G(x) using modulo 2 division. Mathematically, if E(x) is the error, then modulo 2 division of [$M(x) + E(x)$] by $G(x)$ is done.
    - If there is no remainder, then it implies that $E(x)$. The data frame is accepted.
    - A remainder indicates a non-zero value of E(x), or in other words presence of an error. So, the data frame is rejected. The receiver may then send an erroneous acknowledgment back to the sender for retransmission.

Code:

```
clc;
close all;
clear all;
x = input('Enter Data Word (in the form [x,x,x,x]): ');
y = input('Enter Pattern Word (in the form [x,x,x,x]): ');
p = length(y);
k = length(x);
n = p+k-1;
a = x;
%input data matrix
for z=1:(n-k)
a(k+z) = 0;
end
%parity bits
for z=1:p
td(z)= a(z);
tr2(z)=0;
tr3(z)=1;
end
tr = xor(td,y);
%division process
while(1)
z=z+1;
if(z>n)
break;
end

tr(p+1)=a(z);
for l=2:(p+1)
tr1(l-1)=tr(l);
end

tr = tr1;
if(tr(1)==0)
tr=xor(tr,tr2);
else
tr=xor(tr,y);
end
end
code=x;
for z=2:p
code(k+z-1)=tr(z);
end
disp("Encoded Codeword: ")
disp(code)
```

```
x=code;
p = length(y);
k = length(x);
a = x;
%input data matrix
for z=1:(n-k)
a(k+z) = 0;
end
%parity bits
for z=1:p
td(z)= a(z);
tr2(z)=0;
tr3(z)=1;
end
tr = xor(td,y);
%division process
while(1)
z=z+1;
if(z>n)
break;
end

tr(p+1)=a(z);
for l=2:(p+1)
tr1(l-1)=tr(l);
end

tr = tr1;
if(tr(1)==0)
tr=xor(tr,tr2);
else
tr=xor(tr,y);
end
end
code=x;
for z=2:p
code(k+z-1)=tr(z);
end
disp("Error Bits: ")
disp(tr(2:p))
```

Output:



```
Enter Data Word (in the form [x,x,x,x]): [1 1 0 0 1 0]
Enter Pattern Word (in the form [x,x,x,x]): [1 1 0 0]
Encoded Codeword:
     1     1     0     0     1     0     1     0     0

Error Bits:
   0   0   0

>>
```

Conclusion:

We studied about Cyclic Redundancy Code (CRC) and its use in detecting errors.

EXPERIMENT 3

Aim:

To study the Bellman Ford Algorithm and use it to find the minimum distance from each node to the destination node on the given graph.

Requirements:

C++

Theory:

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.
Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges (and possibly some paths longer than i edges). Since the longest possible path without a cycle can be |V|-1 edges, the edges must be scanned |V|-1 times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length |V| edges has been found which can only occur if at least one negative cycle exists in the graph.

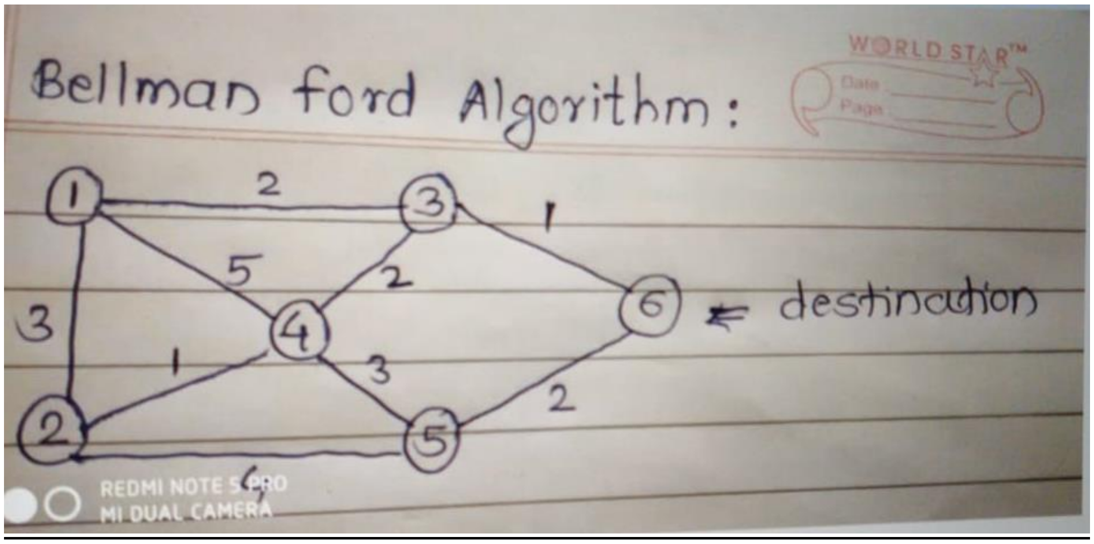
Algorithm:

- Create a Class Graph with the member variables as node and graph, and member functions as addnode(), printTab() and BellmanFord().
- The variable node is for storing the total number of nodes and variable graph is used to store all the edges the destination node is taken as input from the user.
- The addnode() is used to add nodes and edges in the graph.
- The printTab() is used to print the resultant table.
- BellmanFord() is used to implement the algorithm on the variable graph.
- In the BellmanFord() we create a dist[] array of size node and initialize all elements to infinity and set the value of distance of destination node to zero.
- Now using nested for loops update the dist[] array with the minimum edge required.
- Then print the resultant dist[] array according to the nodes using the printTab().

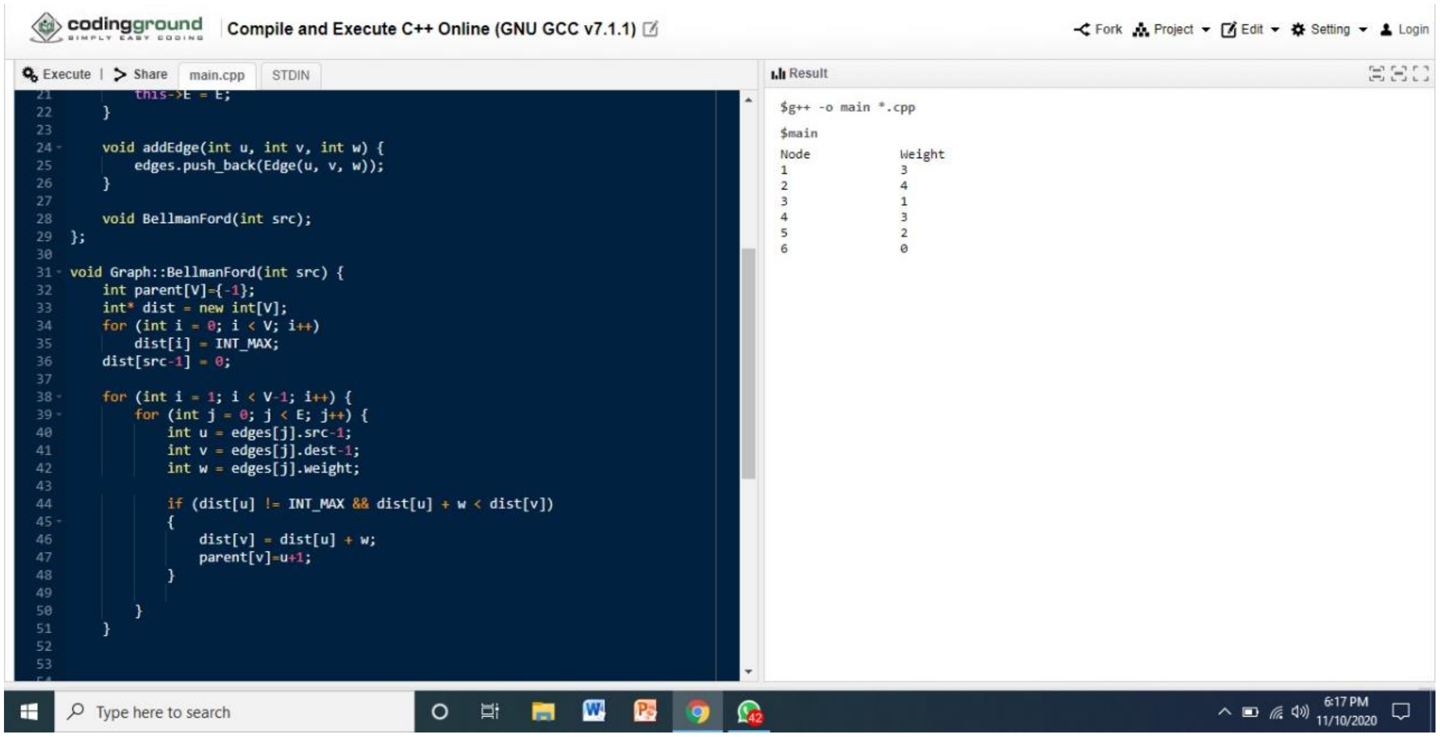
Code:

```
#include <bits/stdc++.h>
using namespace std;
class Edge{
   public:
      int src, dest, weight;
      Edge(int src, int dest, int weight) {
         this->src = src;
         this->dest = dest;
         this->weight = weight;
      }
};
class Graph {
    int V, E;
    vector<Edge> edges;
public:
    Graph(int V, int E) {
        this->V = V;
        this->E = E;
    }
    void addEdge(int u, int v, int w) {
        edges.push_back(Edge(u, v, w));
    }
    void BellmanFord(int src);
};
void Graph::BellmanFord(int src) {
    int parent[V]={-1};
    int* dist = new int[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src-1] = 0;

    for (int i = 1; i < V-1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].src-1;
            int v = edges[j].dest-1;
            int weight = edges[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            {
                dist[v] = dist[u] + weight;
                parent[v]=u+1;
            }
        }
    }
    cout << "Node"<<"\t\t"<<    "Weight" << endl;
    for (int i = 0; i < V; i++){
        cout<<i+1<<"\t\t"<<dist[i]<<"\n";
    }
    return;
}
int main() {
    int V = 6, E = 18;
    Graph g(V, E);
    g.addEdge(3, 6, 1); g.addEdge(6, 3, 1);
    g.addEdge(5, 6, 2); g.addEdge(6, 5, 2);
    g.addEdge(3, 4, 2); g.addEdge(4, 3, 2);
    g.addEdge(4, 5, 3); g.addEdge(5, 4, 3);
    g.addEdge(1, 3, 2); g.addEdge(3, 1, 2);
    g.addEdge(1, 5, 4); g.addEdge(5, 1, 4);
    g.addEdge(2, 4, 1); g.addEdge(4, 2, 1);
    g.addEdge(2, 5, 4); g.addEdge(5, 2, 4);
    g.addEdge(1, 2, 3); g.addEdge(2, 1, 3);
    g.BellmanFord(6);
    return 0;
}
```

Given Graph:



Output:



```cpp
        this->E = E;
    }

    void addEdge(int u, int v, int w) {
        edges.push_back(Edge(u, v, w));
    }

    void BellmanFord(int src);
};

void Graph::BellmanFord(int src) {
    int parent[V]={-1};
    int* dist = new int[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src-1] = 0;

    for (int i = 1; i < V-1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].src-1;
            int v = edges[j].dest-1;
            int w = edges[j].weight;

            if (dist[u] != INT_MAX && dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
                parent[v]=u+1;
            }

        }
    }


```

```
$g++ -o main *.cpp
$main
Node            Weight
1               3
2               4
3               1
4               3
5               2
6               0
```

Conclusion:

We studied the Bellman-Ford algorithm and used it to find the minimum distance from all nodes to the destination node and printed the resultant table using the python3 programming language

EXPERIMENT 4

Aim:

Connecting two networks using router using packet tracer.

Requirements:

Cisco Packet Tracer 7.3.1

Theory:

A router is connected to two or more data lines from different IP networks.[b] When a data packet comes in on one of the lines, the router reads the network address information in the packet header to determine the ultimate destination. Then, using information in its routing table or routing policy, it directs the packet to the next network on its journey.

When multiple routers are used in interconnected networks, the routers can exchange information about destination addresses using a routing protocol. Each router builds up a routing table, a list of routes, between two computer systems on the interconnected networks.

A router has two types of network element components organized onto separate processing planes:

- Control plane: A router maintains a routing table that lists which route should be used to forward a data packet, and through which physical interface connection. It does this using internal pre-configured directives, called static routes, or by learning routes dynamically using a routing protocol. Static and dynamic routes are stored in the routing table. The control-plane logic then strips non-essential directives from the table and builds a forwarding information base (FIB) to be used by the forwarding plane.
- Forwarding plane: The router forwards data packets between incoming and outgoing interface connections. It forwards them to the correct network type using information that the packet header contains matched to entries in the FIB supplied by the control plane.

Routing is the process of selecting a path for traffic in a network or between or across multiple networks. Broadly, routing is performed in many types of networks, including circuit-switched networks, such as the public switched telephone network (PSTN), and computer networks, such as the Internet.

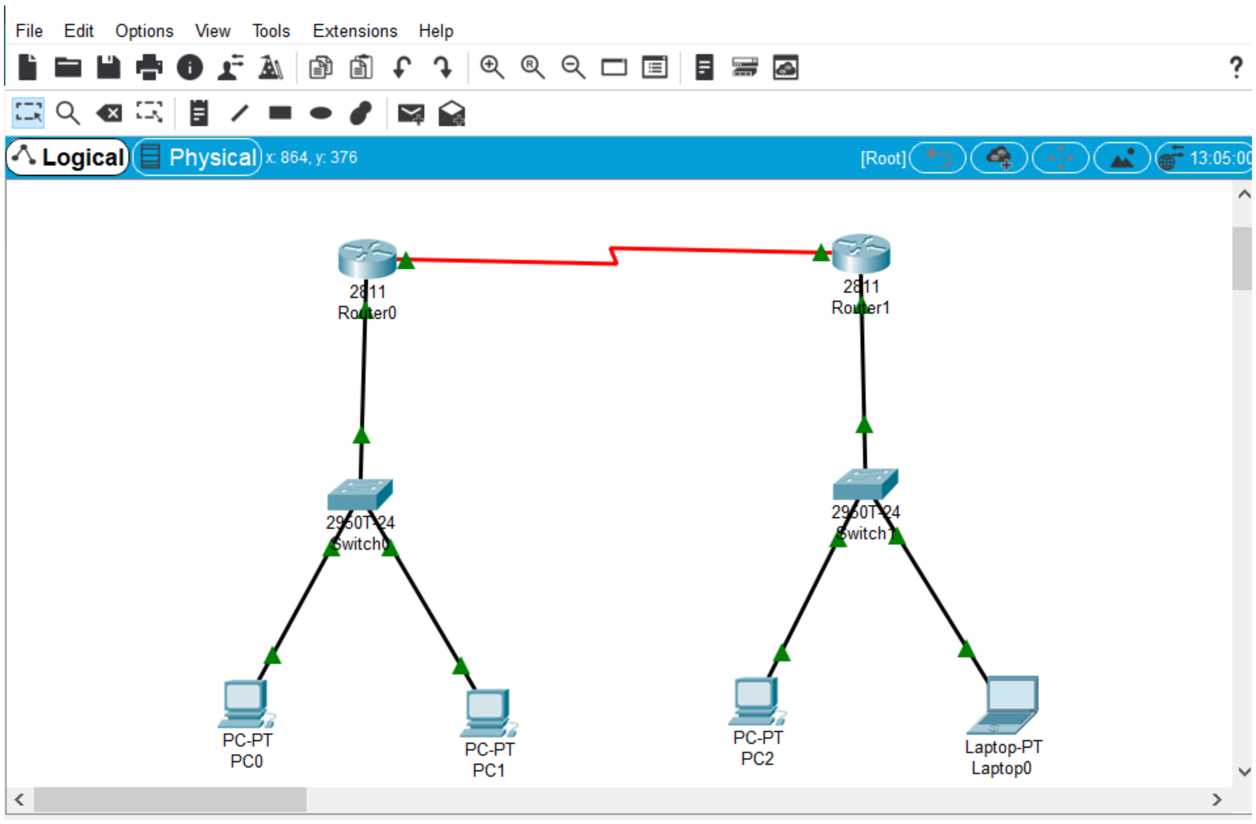
In packet switching networks, routing is the higher-level decision making that directs network packets from their source toward their destination through intermediate network nodes by specific packet forwarding mechanisms. Packet forwarding is the transit of network packets from one network interface to another. Intermediate nodes are typically network hardware devices such as routers, gateways, firewalls, or switches. General-purpose computers also forward packets and perform routing, although they have no specially optimized hardware for the task.

The routing process usually directs forwarding on the basis of routing tables. Routing tables maintain a record of the routes to various network destinations. Routing tables may be specified by an administrator, learned by observing network traffic or built with the assistance of routing protocols.

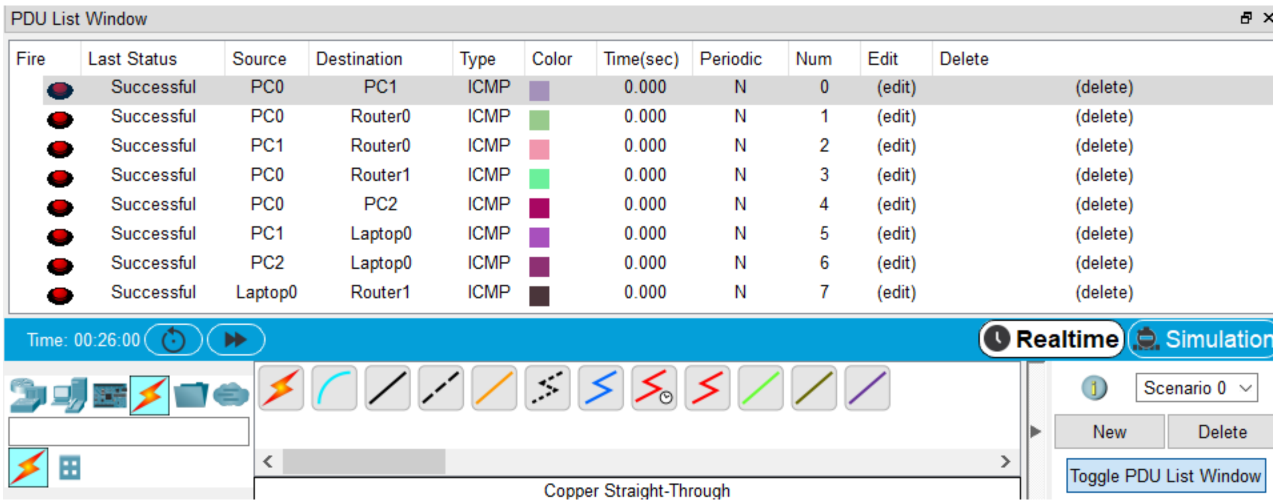
<u>Procedure:</u>

1) Create and configure two different networks using switches, routers and computers
2) Assign IP addresses to all the devices
3) Connect the two routers using a Serial DTE cable
4) Configure the routers using host commands to create a path between the two routers (hence the two networks)
5) Test the routed network by sending messages across different devices in different networks

Network Diagram:



Output:



Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
●	Successful	PC0	PC1	ICMP	🟪	0.000	N	0	(edit)	(delete)
●	Successful	PC0	Router0	ICMP	🟩	0.000	N	1	(edit)	(delete)
●	Successful	PC1	Router0	ICMP	🟥	0.000	N	2	(edit)	(delete)
●	Successful	PC0	Router1	ICMP	🟩	0.000	N	3	(edit)	(delete)
●	Successful	PC0	PC2	ICMP	🟥	0.000	N	4	(edit)	(delete)
●	Successful	PC1	Laptop0	ICMP	🟪	0.000	N	5	(edit)	(delete)
●	Successful	PC2	Laptop0	ICMP	🟪	0.000	N	6	(edit)	(delete)
●	Successful	Laptop0	Router1	ICMP	🟫	0.000	N	7	(edit)	(delete)

<u>**Conclusion:**</u>

We saw how to connect two networks using the concepts of routers and routing, also observed the successful connections by send messages across networks

EXPERIMENT 5

Aim:

Connecting two networks using router using packet tracer.

Requirements:

Cisco Packet Tracer 7.3.1

Theory:

A virtual LAN (VLAN) is any broadcast domain that is partitioned and isolated in a computer network at the data link layer (OSI layer 2).[1][2] LAN is the abbreviation for local area network and in this context virtual refers to a physical object recreated and altered by additional logic. VLANs work by applying tags to network frames and handling these tags in networking systems – creating the appearance and functionality of network traffic that is physically on a single network but acts as if it is split between separate networks. In this way, VLANs can keep network applications separate despite being connected to the same physical network, and without requiring multiple sets of cabling and networking devices to be deployed.

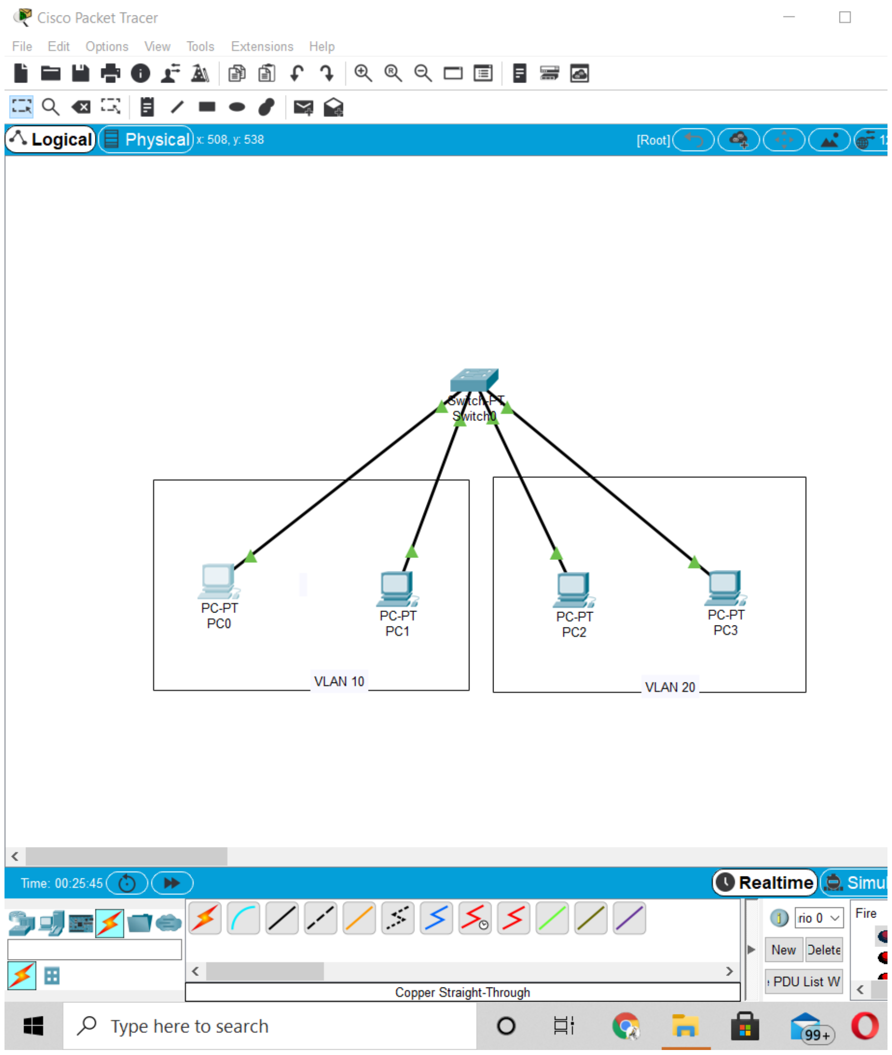
VLANs allow network administrators to group hosts together even if the hosts are not directly connected to the same network switch. Because VLAN membership can be configured through software, this can greatly simplify network design and deployment. Without VLANs, grouping hosts according to their resource needs the labour of relocating nodes or rewiring data links. VLANs allow devices that must be kept separate to share the cabling of a physical network and yet be prevented from directly interacting with one another.

To subdivide a network into VLANs, one configures network equipment. Simpler equipment might partition only each physical port (if even that), in which case each VLAN runs over a dedicated network cable. More sophisticated devices can mark frames through VLAN tagging, so that a single interconnect (trunk) may be used to transport data for multiple VLANs. Since VLANs share bandwidth, a VLAN trunk can use link aggregation, quality-of-service prioritization, or both to route data efficiently.

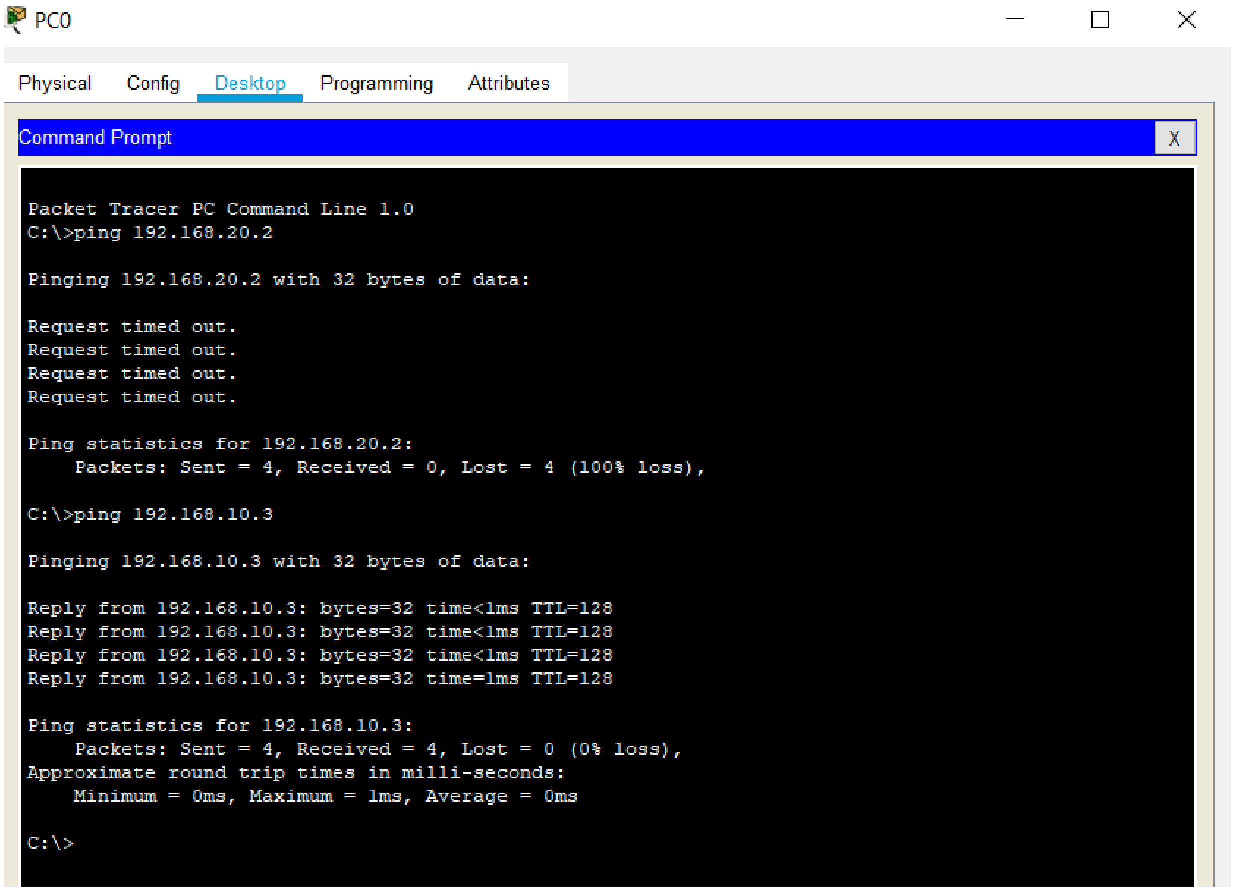
Procedure:

1) Set up and configure a network consisting of a switch and computers and connect them using fast ethernet cables.
2) Assign IP addresses to each device.
3) In the switch CLI create two VLANs using the vlan command followed by the vlan number
4) Using the interface and switchport access commands assign each device to a VLAN of your choice
5) Test the VLANs by sending messages across different devices and different VLANs

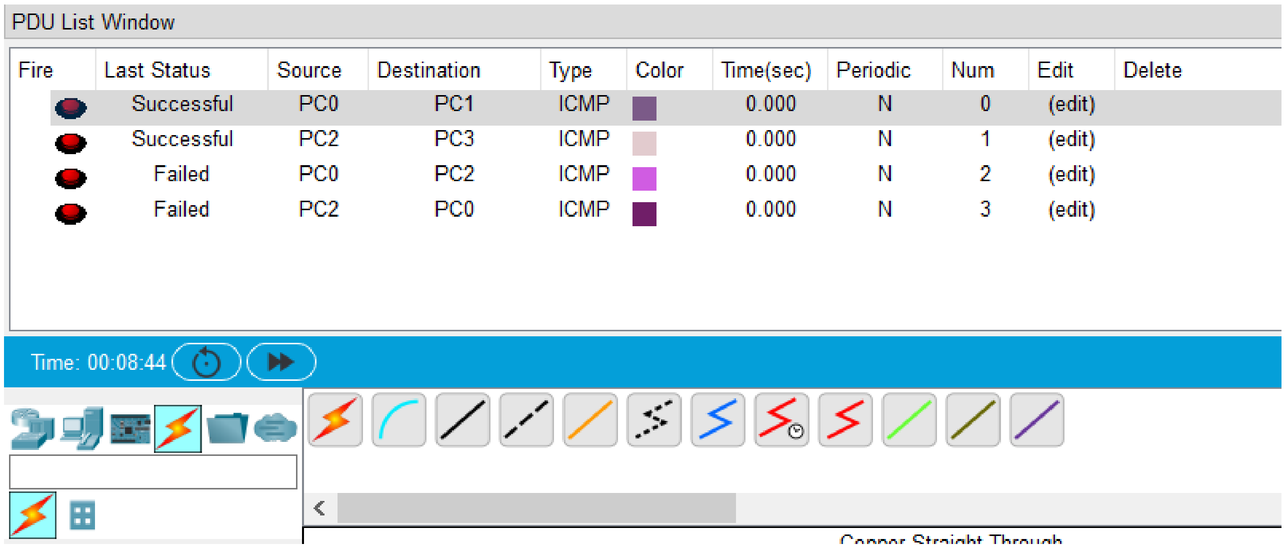
Network Diagram:



Output:



```
Packet Tracer PC Command Line 1.0
C:\>ping 192.168.20.2

Pinging 192.168.20.2 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.20.2:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\>ping 192.168.10.3

Pinging 192.168.10.3 with 32 bytes of data:

Reply from 192.168.10.3: bytes=32 time<1ms TTL=128
Reply from 192.168.10.3: bytes=32 time<1ms TTL=128
Reply from 192.168.10.3: bytes=32 time<1ms TTL=128
Reply from 192.168.10.3: bytes=32 time=1ms TTL=128

Ping statistics for 192.168.10.3:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms

C:\>
```



Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
●	Successful	PC0	PC1	ICMP	🟪	0.000	N	0	(edit)	
●	Successful	PC2	PC3	ICMP	🟪	0.000	N	1	(edit)	
●	Failed	PC0	PC2	ICMP	🟪	0.000	N	2	(edit)	
●	Failed	PC2	PC0	ICMP	🟪	0.000	N	3	(edit)	

Conclusion:

We saw how to create and configure VLANs using Cisco Packet Tracer, also observed how VLANs work, we discovered that communication can take place only between members of a VLAN and not across VLANs.

EXPERIMENT 6

Aim:

To study the Dijkstra's algorithm and used it to find the minimum distance from all nodes to the source node in a graph.

Requirements:

C++

Theory:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithm is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's. The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the generic Dijkstra shortest-path algorithm.

Algorithm:

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

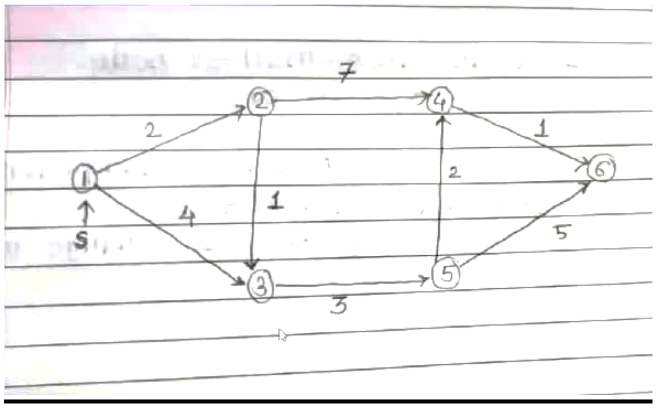
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* doesn't include all vertices
   i. Pick a vertex u which is not there in *sptSet* and has minimum distance value.
   ii. Include u to *sptSet*.
   iii. Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Code:

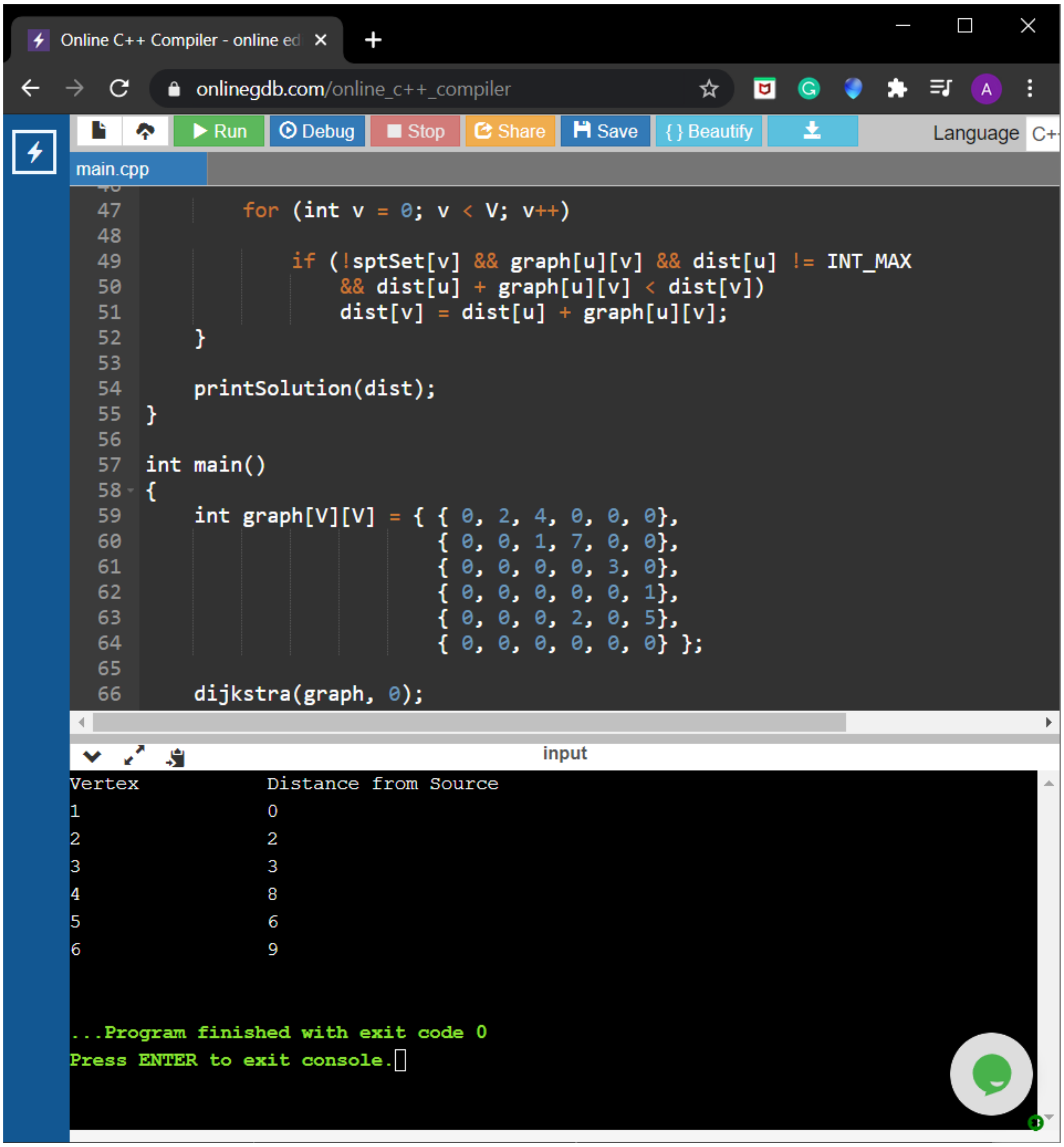
```
#include <limits.h>
#include <stdio.h>
#define V 6
int minDistance(int dist[], bool sptSet[])
{
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;

	return min_index;
}
void printSolution(int dist[])
{
	printf("Vertex \t\t Distance from Source\n");
	for (int i = 0; i < V; i++)
		printf("%d \t\t %d\n", i+1, dist[i]);
}
void dijkstra(int graph[V][V], int src)
{
	int dist[V];

	bool sptSet[V];

	for (int i = 0; i < V; i++)
```

```
            dist[i] = INT_MAX, sptSet[i] = false;

        dist[src] = 0;

        for (int count = 0; count < V - 1; count++) {
            int u = minDistance(dist, sptSet);

            sptSet[u] = true;

            for (int v = 0; v < V; v++)

                if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                    && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        printSolution(dist);
}
int main()
{
        int graph[V][V] = { { 0, 2, 4, 0, 0, 0},
                            { 0, 0, 1, 7, 0, 0},
                            { 0, 0, 0, 0, 3, 0},
                            { 0, 0, 0, 0, 0, 1},
                            { 0, 0, 0, 2, 0, 5},
                            { 0, 0, 0, 0, 0, 0} };

        dijkstra(graph, 0);
        return 0;
}
```

<u>Given Graph:</u>



Output:


Online C++ Compiler - online ed
onlinegdb.com/online_c++_compiler
Run
Debug
Stop
Share
Save
{ } Beautify
Language
main.cpp
input

```
        for (int v = 0; v < V; v++)

            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    printSolution(dist);
}

int main()
{
    int graph[V][V] = { { 0, 2, 4, 0, 0, 0},
                        { 0, 0, 1, 7, 0, 0},
                        { 0, 0, 0, 0, 3, 0},
                        { 0, 0, 0, 0, 0, 1},
                        { 0, 0, 0, 2, 0, 5},
                        { 0, 0, 0, 0, 0, 0} };

    dijkstra(graph, 0);
```

```
Vertex          Distance from Source
1               0
2               2
3               3
4               8
5               6
6               9


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

We studied the Dijkstra's algorithm and used it to find the minimum distance from all nodes to the source.