A
Project Report
On
“Broadcast & Multicast Routing Algorithms”

Department of Computer Science & Engineering

NATIONAL INSTITUE OF TECHNOLOGY PATNA

University Campus, Bihar - 800005

Submitted By:

Name	Roll No.
Anish Kumar Singh	2041013
Nitesh Kumar Yadav	2041003

Course Code: CS410292

Course Title: Computing Lab 2
(Advanced Computer Networks)

Branch: M. Tech CSE – IS

TABLE OF CONTENTS

LIST OF FIGURES

AIM OF THE EXPERIMENT

Design and develop broadcast and multicast routing algorithms. Discuss the analysis of each routing algorithm in details.

1. INTRODUCTION

1.1 Routing

Routing is the process of forwarding the packets from source to the destination but the best route to send the packets is determined by the routing algorithm. In order to transfer the packets from source to the destination, the network layer must determine the best route through which packets can be transmitted. Whether the network layer provides datagram service or virtual circuit service, the main job of the network layer is directing Internet traffic efficiently by providing the best route. The routing protocol provides this job. The routing protocol is a routing algorithm that provides the best path from the source to the destination. After a data packet leaves its source, it can choose among the many different paths to reach its destination but the best path is the path that has the "least-cost path" from source to the destination.

When a device has multiple paths to reach a destination, it always selects one path by preferring it over others. This selection process is termed as Routing. Routing is done by special network devices called routers or it can be done by means of software processes. The software based routers have limited functionality and limited scope. A router is always configured with some default route. A default route tells the router where to forward a packet if there is no route found for specific destination. In case there are multiple path existing to reach the same destination, router can make decision based on the following information:

- Hop Count
- Bandwidth
- Metric
- Prefix-length
- Delay

Routes can be statically configured or dynamically learnt. One route can be configured to be preferred over others.

1.2 Classification of Routing Algorithms

The routing algorithms can be classified as follows:

- **Adaptive Algorithms:** These are the algorithms which change their routing decisions whenever network topology or traffic load changes. The changes in routing decisions are reflected in the topology as well as traffic of the network. Also known as dynamic routing, these make use of dynamic information such as current topology, load, delay, etc. to select routes. Optimization parameters are distance, number of hops and estimated transit time. Further these are classified as follows:

a) **Isolated –** In this method each, node makes its routing decisions using the information it has without seeking information from other nodes. The sending node doesn't have information about status of particular link. Disadvantage is that packet may be sent through a congested network which may result in delay. Examples: Hot potato routing, backward learning.

b) **Centralized –** In this method, a centralized node has entire information about the network and makes all the routing decisions. Advantage of this is only one node is required to keep the information of entire network and disadvantage is that if central node goes down the entire network is done. Link state algorithm is referred to as a centralized algorithm since it is aware of the cost of each link in the network.

c) **Distributed –** In this method, the node receives information from its neighbour's and then takes the decision about routing the packets. Disadvantage is that the packet may be delayed if there is change in between interval in which it receives information and sends packet. It is also known as decentralized algorithm as it computes the least-cost path between source and destination.

- **Non-Adaptive Algorithms:** These are the algorithms which do not change their routing decisions once they have been selected. This is also known as static routing as route to be taken is computed in advance and downloaded to routers when router is booted. Further these are classified as follows:

a) **Flooding –** This adapts the technique in which every incoming packet is sent on every outgoing line except from which it arrived. One problem with this is that packets may go in loop and as a result of which a node may receive duplicate packets. These problems can be overcome with the help of sequence numbers, hop count and spanning tree.

b) **Random walk –** In this method, packets are sent host by host or node by node to one of its neighbours randomly. This is highly robust method which is usually implemented by sending packets onto the link which is least queued.

1.3 Unicast routing

Most of the traffic on the internet and intranets known as unicast data or unicast traffic is sent with specified destination. Routing unicast data over the internet is called unicast routing. It is the simplest form of routing because the destination is already known. Hence the router just has to look up the routing table and forward the packet to next hop.

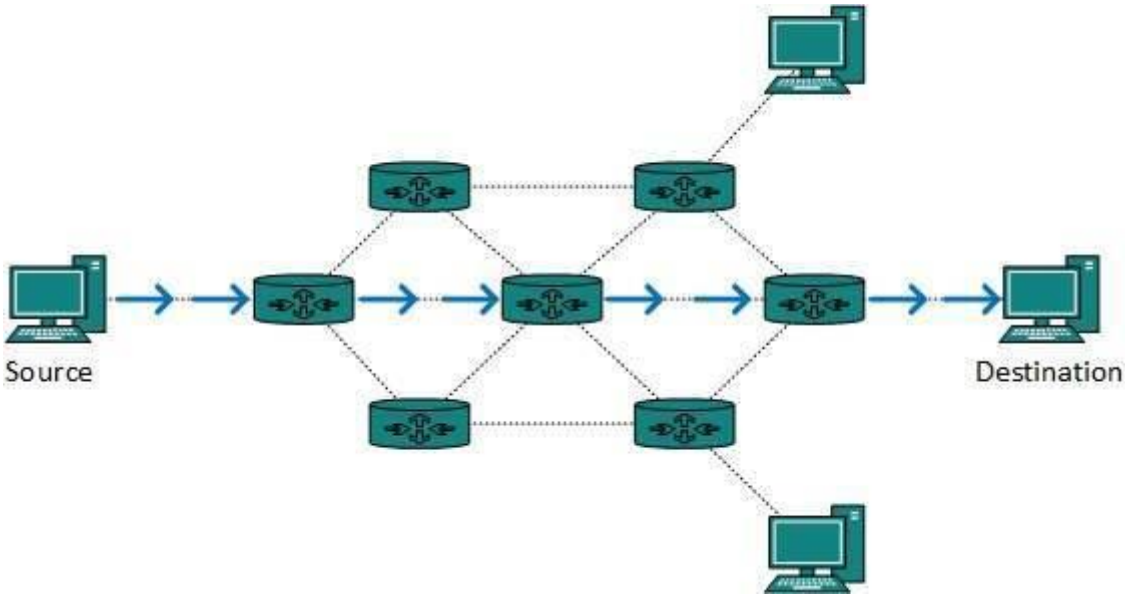

Figure 1- Example of Unicast Routing

1.4 Broadcast routing

By default, the broadcast packets are not routed and forwarded by the routers on any network. Routers create broadcast domains. But it can be configured to forward broadcasts in some special cases. A broadcast message is destined to all network devices. Broadcast routing can be done in two ways (algorithm):

- A router creates a data packet and then sends it to each host one by one. In this case, the router creates multiple copies of single data packet with different destination addresses. All packets are sent as unicast but because they are sent to all, it simulates as if router is broadcasting. This method consumes lots of bandwidth and router must destination address of each node.

- Secondly, when router receives a packet that is to be broadcasted, it simply floods those packets out of all interfaces. All routers are configured in the same way. This method is easy on router's CPU but may cause the problem of duplicate packets received from peer routers.

Reverse path forwarding is a technique, in which router knows in advance about its predecessor from where it should receive broadcast. This technique is used to detect and discard duplicates.

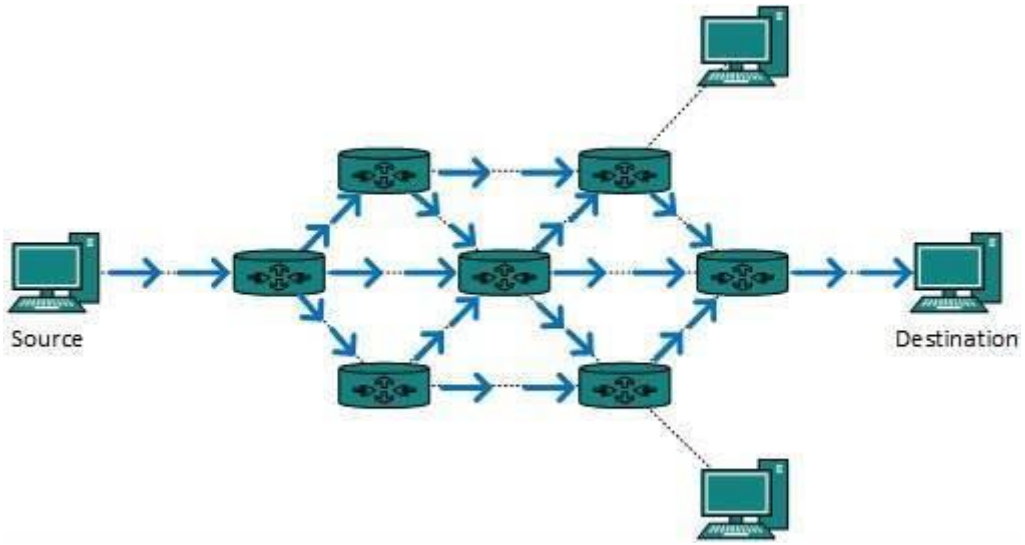

Figure 2- Example of Broadcast Routing

1.5 Multicast Routing

Multicast routing is special case of broadcast routing with significance difference and challenges. In broadcast routing, packets are sent to all nodes even if they do not want it. But in Multicast routing, the data is sent to only nodes which wants to receive the packets.

The router must know that there are nodes, which wish to receive multicast packets (or stream) then only it should forward. Multicast routing works spanning tree protocol to avoid looping. Multicast routing also uses reverse path Forwarding technique, to detect and discard duplicates and loops.

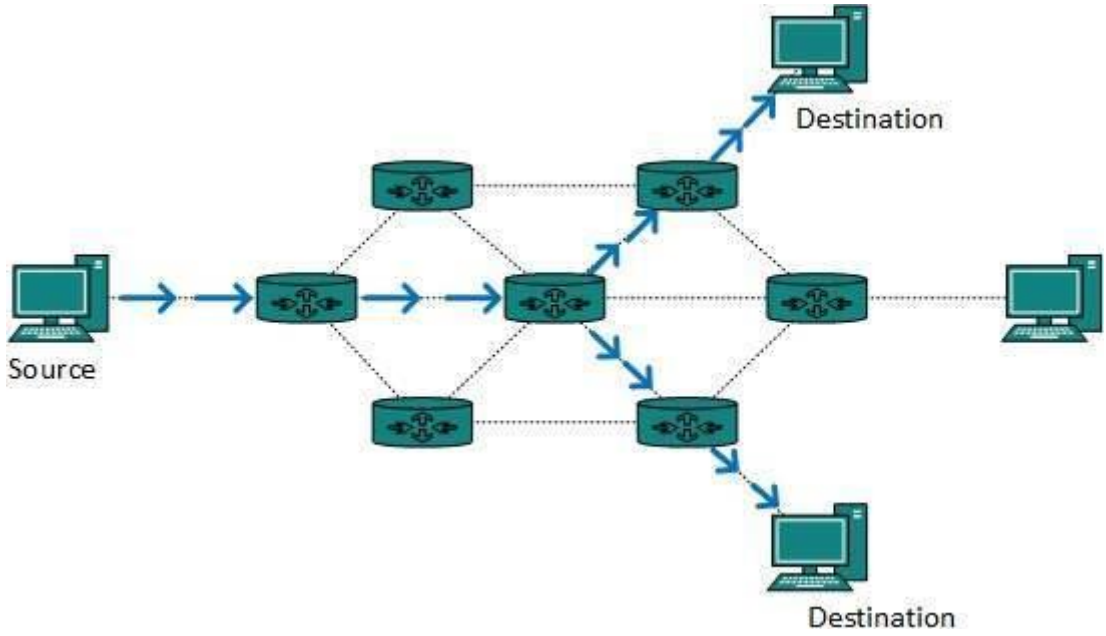

Figure 3- Example of Multicast Routing

1.6 Anycast Routing

Anycast packet forwarding is a mechanism where multiple hosts can have same logical address. When a packet destined to this logical address is received, it is sent to the host which is nearest in routing topology.

Anycast routing is done with help of DNS server. Whenever an Anycast packet is received it is enquired with DNS to where to send it. DNS provides the IP address which is the nearest IP configured on it.

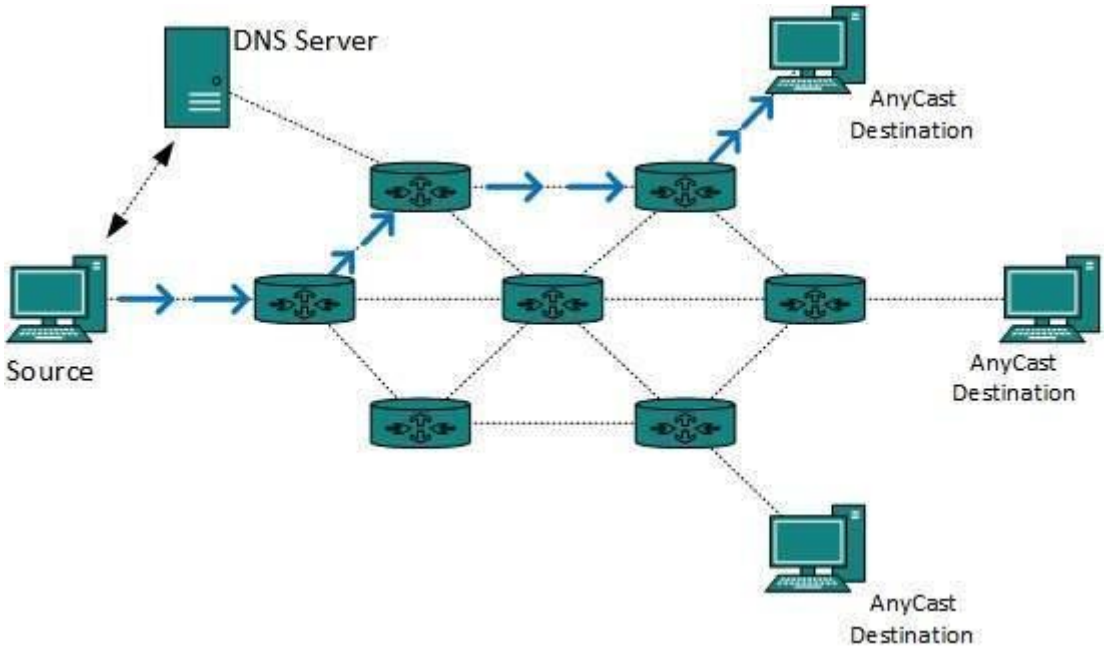

Figure 4- Example of Anycast Routing

2. SYSTEM REQUIREMENTS

Hardware Requirements:

- 2 GHz x86 processor
- 256 MB of system memory (RAM)
- 100 MB of hard-drive space
- Monitor to display output
- Keyboard/Mouse for data input

Software Requirements:

- C/C++ Compiler (cc, gcc, egcs, ...)
- MS Word(Documentation)

3. FLOWCHART / DATA FLOW DIAGRAM

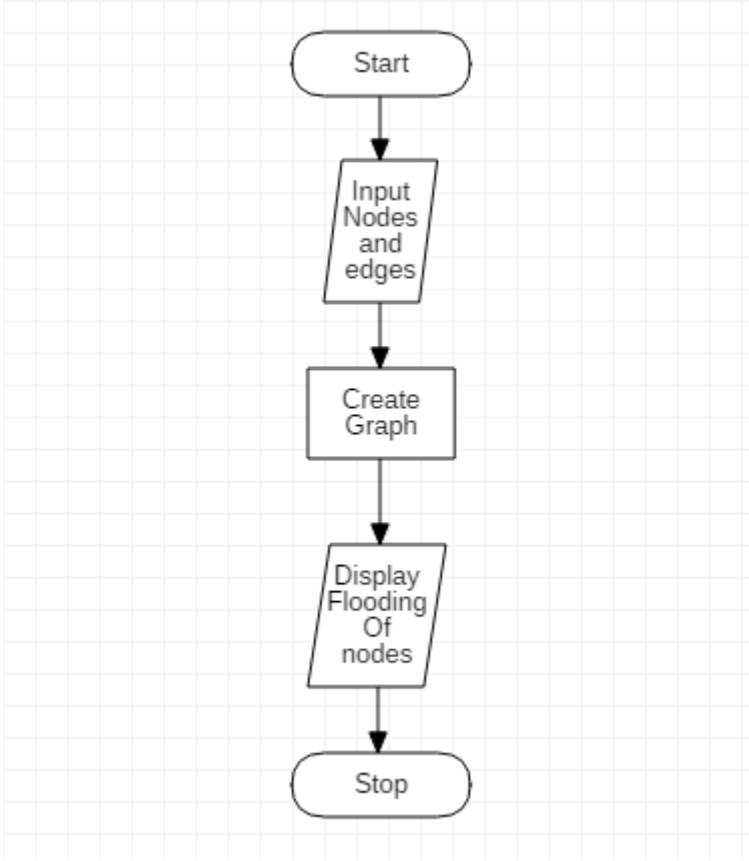

Figure 5- Flowchart of Flooding (Broadcast Routing)

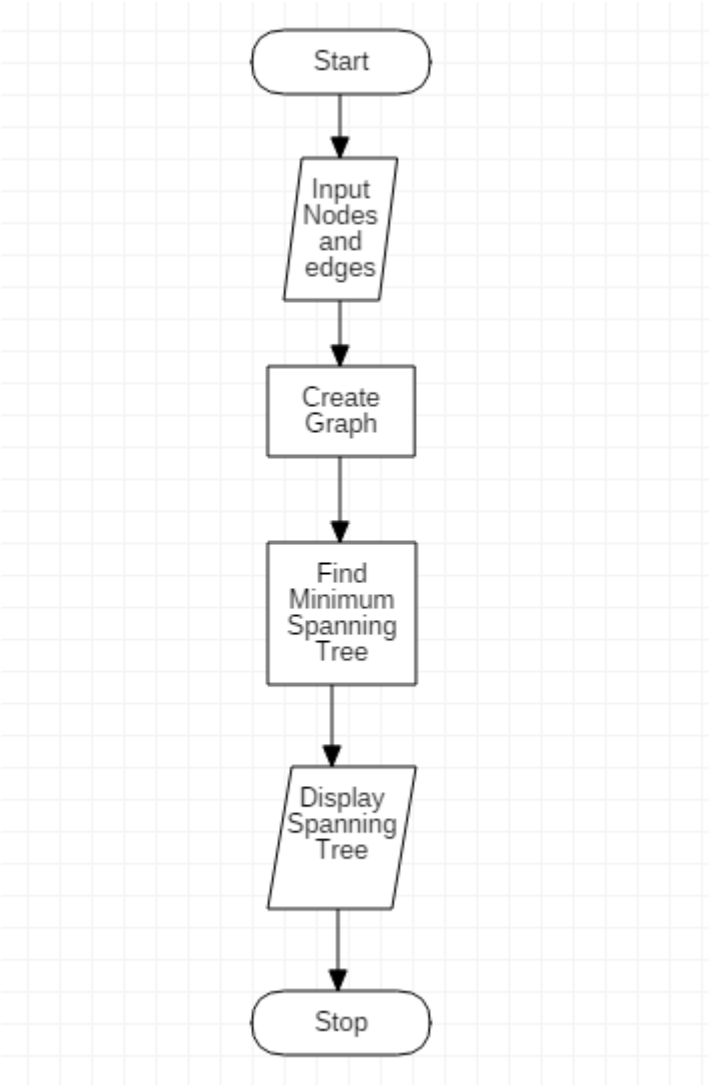

Figure 6- Flowchart of Sink /Spanning Tree (Broadcast Routing)

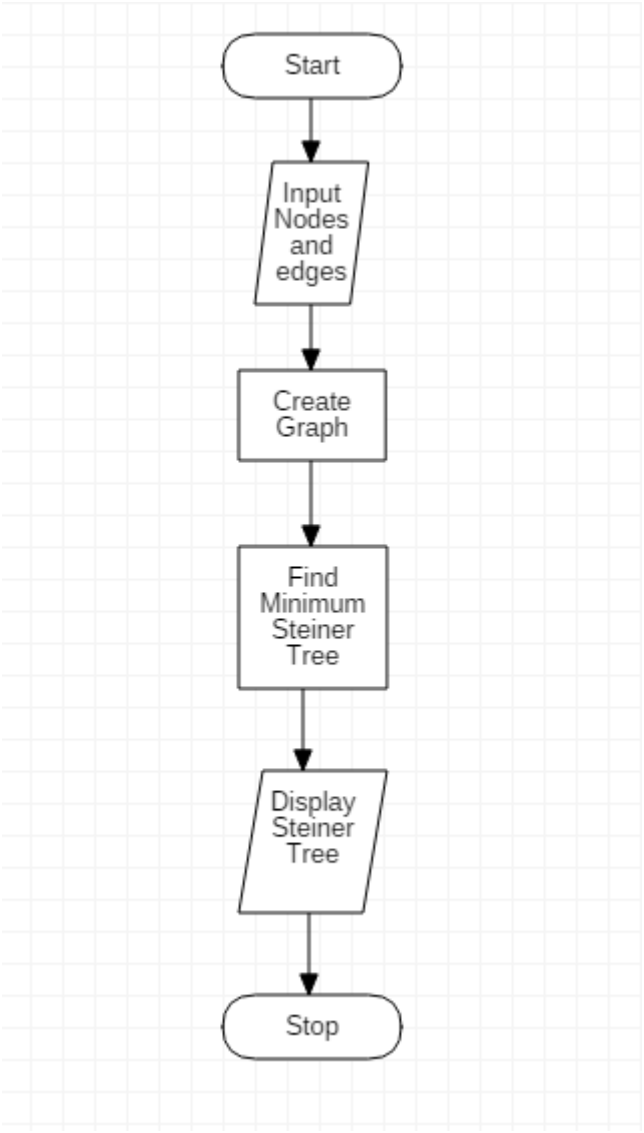

Figure 7- Flowchart of Steiner Tree (Multicast Routing)

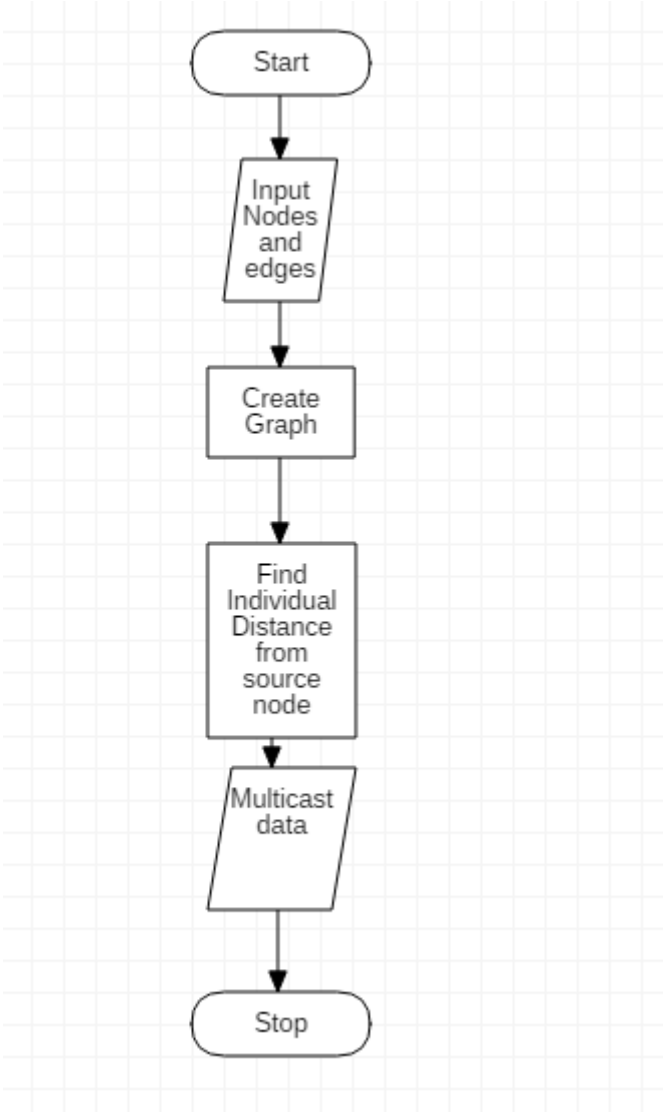

Figure 8- Flowchart of Multiple Unicast (Multicast Routing)

4. CODE

4.1 Code to implement Flooding (Broadcast Routing)

```
// Program to show flooding in broadcast algorithm from a given source vertex.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;
    public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;
```

```
    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s.
        //If a adjacent has not been visited, then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(1, 3);
    g.addEdge(3, 5);
    g.addEdge(5, 6);
    g.addEdge(3, 4);
    g.addEdge(2, 6);
    g.addEdge(4, 6);
    g.addEdge(2, 4);
    cout << "Broadcast Routing(Flooding) Example:\n";
    cout << "Starting from Source vertex '0':\n";
    g.BFS(0);

    return 0;
}
```

4.2 Code to implement Sink/ Spanning Tree (Broadcast Routing)

```
/* C Program to show Sink tree (Broadcast Routing) */

#include<stdio.h>
#include<stdlib.h>

#define MAX 10
#define TEMP 0
#define PERM 1
#define infinity 9999
#define NIL -1

//Structure to define edge
struct edge
{
	int u; // Source Node of an edge
	int v; //Destination Node of an edge
};

int n;
int adj[MAX][MAX]; // Adjacency Cost Matrix

int predecessor[MAX];
int status[MAX];
int length[MAX];

//Function to create a Graph
void create_graph();
//Function to create a Tree
void maketree(int r, struct edge tree[MAX]);
int min_temp();

// Main function
int main()
{
	int wt_tree = 0;
	int i, root;
	struct edge tree[MAX];
	printf("\nSink / Spanning Tree(Broadcast Routing) Example: ");
	create_graph();

	printf("\nEnter Source(root) vertex : ");
	scanf("%d",&root);

	maketree(root, tree);

	printf("\nEdges to be included in Spanning(Sink) tree are : \n");
```

```
	for(i=1; i<=n-1; i++)
	{
		printf("%d-> ",tree[i].u);
		printf("%d\n",tree[i].v);
		wt_tree += adj[tree[i].u][tree[i].v];
	}
	printf("\nWeight of spanning tree is : %d\n", wt_tree);

	return 0;

}
/*End of main()*/

void maketree(int r, struct edge tree[MAX])
{
	int current,i;
	int count = 0;/*number of vertices in the tree*/

	/*Initialize all vertices*/
	for(i=0; i<n; i++)
	{
		predecessor[i] = NIL;
		length[i] = infinity;
		status[i] = TEMP;
	}

	/*Make length of root vertex 0*/
	length[r] = 0;

	while(1)
	{
		/*Search for temporary vertex with minimum length
		and  make it current vertex*/
		current = min_temp();

		if(current == NIL)
		{
			if(count == n-1) /*No temporary vertex left*/
				return;
			else /*Temporary vertices left with length infinity*/
			{
				printf("\nGraph is not connected, No spanning tree possible\n");
				exit(1);
			}
		}
```

```
        /*Make the current vertex permanent*/
    status[current] = PERM;

        /*Insert the edge ( predecessor[current], current) into the tree,
        except when the current vertex is root*/
        if(current != r)
        {
      count++;
          tree[count].u = predecessor[current];
          tree[count].v = current;
        }

        for(i=0; i<n; i++)
            if(adj[current][i] > 0 && status[i] == TEMP)
                if(adj[current][i] < length[i])
                {
                    predecessor[i] = current;
                    length[i] = adj[current][i];
                }
    }

}/*End of make_tree( )*/

/*Returns the temporary vertex with minimum value of length
  Returns NIL if no temporary vertex left or
  all temporary vertices left have pathLength infinity*/
int min_temp()
{
    int i;
    int min = infinity;
    int k = -1;

    for(i=0; i<n; i++)
    {
        if(status[i] == TEMP && length[i] < min)
        {
            min = length[i];
            k = i;
        }
    }

    return k;
}
/*End of min_temp()*/
```

```
void create_graph()
{
	int i,max_edges,origin,destin,wt;

	printf("\nEnter number of Nodes : ");
	scanf("%d",&n);
	max_edges = n*(n-1)/2;

	for(i=1; i<=max_edges; i++)
	{
		printf("\nEnter edge %d(-1 -1 to quit) : ",i);
		scanf("%d %d",&origin,&destin);
		if((origin == -1) && (destin == -1))
			break;
		printf("\nEnter weight for this edge : ");
		scanf("%d",&wt);
		if( origin >= n || destin >= n || origin < 0 || destin < 0)
		{
			printf("\nInvalid edge!\n");
			i--;
		}
		else
		{
			adj[origin][destin] = wt;
			adj[destin][origin] = wt;
		}
	}
}
```

4.3 Code to implement Source Based Tree/Steiner Tree (Multicast Routing)

```
/*
 * Steiner Tree in Graph.
 * We need to form an MST "T" that contains all the given terminals (vertices).
 * The MST can include remaining vertices also.
 * Dijkstra's Shortest Path Algorithm is used here.
 * Initially T is empty.
 */

#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define V 7     /* Number of Vertices in Graph */
#define K 4     /* Number of Terminal Vertices in MST T */
```

```
int dist[V];
int parent[V];

int T[V] = {0};      /* Set of Vertices in MST T */
int y = 0;           /* Number of Vertices in MST T */

int a[V] = {0};      /* Stores Set of Vertices in the desired Path
                     from source to destination vertex excluding the latter*/

int d = 0;           /* Number of Vertices in the desired Path
                     from source to destination vertex excluding the latter*/


/*
 * A utility function to find the vertex with minimum distance value, from
 * the set of vertices not yet included in shortest path tree
 */
int minDistance(int dist[], bool sptSet[])
{
   /*
    * Initialize min value
    */
   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 find_parent(int parent[], int j)
{
   /*
    * Base Case : If j is source
    */
   if (parent[j] == - 1)
                  return;

   find_parent(parent, parent[j]);
   a[d] = j;
   d++;
}

/*
 * Function to print shortest path from source to j using parent array
 */
```

```
void printPath(int parent[], int j)
{
   if (parent[j] == - 1)  {
                return;
   }
   printPath(parent, parent[j]);
   printf("%d ", j);
}

/*
 * A utility function to print the constructed distance array
 */
int printSolution(int dist[], int n,int x)
{
   int src = x;
   printf("Vertex \tDistance from Source\t Path");
        for (int i = 1; i < V; i++)
        {
                printf("\n%d -> %d \t\t %d\t\t%d ",
                                src, i, dist[i], src);
                printPath(parent, i);
   }
   printf("\n");
}

/*
 * Function that implements Dijkstra's single source shortest path algorithm
 * for a graph represented using adjacency matrix representation
 */
void dijkstra(int graph[V][V], int src)
{
   // The output array. dist[i] will hold the shortest
   // distance from src to i

   //int dist[V];

   bool sptSet[V]; // sptSet[i] will be true if vertex i is included in
   // shortest  path tree or shortest distance from src to i is finalized

   // Initialize all distances as INFINITE and stpSet[] as false
   for (int i = 0; i < V; i++)
                dist[i] = INT_MAX, sptSet[i] = false, parent[i] = -1;

   // Distance of source vertex from itself is always 0
   dist[src] = 0;

   // Find shortest path for all vertices
```

```
    for (int count = 0; count < V; count++)
    {
                // Pick the minimum distance vertex from the set of vertices not
                // yet processed. u is always equal to src in the first iteration.
                int u = minDistance(dist, sptSet);

                // Mark the picked vertex as processed
                sptSet[u] = true;

                // Update dist value of the adjacent vertices of the picked vertex.
                for (int v = 0; v <= V; v++)

                   // Update dist[v] only if is not in sptSet, there is an edge from
                   // u to v, and total weight of path from src to v through u is
                   // smaller than current value of dist[v]
                   if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                           && dist[u]+graph[u][v] < dist[v]) {

                              parent[v] = u;
                              dist[v] = dist[u] + graph[u][v];
                   }
    }
}


int main()
{
    int i = 0, j = 0;

    int terminal_vertices[K] = {1,3,5,6};
    int remaining_vertices[K] = {0};
    int source_vertex = 0;
    int is_processed[V] = {0};
    int is_added[V][V] = {0};
    int is_included[V][V] = {0};
    int total_cost = 0, count = 0, cost = 0, x = 0;

    int graph[V][V] = {
                              {0,0,0,0,0,0,0},
                              {0,0,7,0,0,0,6},
                              {0,7,0,1,0,0,5},
                              {0,0,1,0,1,3,0},
                              {0,0,0,1,0,1,4},
                              {0,0,0,3,1,0,10},
                              {0,6,5,0,4,10,0}
                         };
    printf("Multicast Routing Example (Steiner Tree):\n");
```

```
/*
 * Step 1 of the algorithm
 * First Terminal Vertex is added to T
 * "Start with a subtree T consisting of
 * one given terminal vertex"
 */
// printf("Terminal Vertex \"%d\" is added to T\n",terminal_vertices[0]);
dijkstra(graph,terminal_vertices[0]);
//printSolution(dist, V, terminal_vertices[0]);
is_processed[terminal_vertices[0]] = 1;
T[0] = terminal_vertices[0];
y++;

/*
 * Step 2 of the algorithm starts here
 * "While T does not span all terminals"
 */
count = 1;
while(count<K)
{
      /*
       * Step 2 a) of the algorithm
       * "Select a terminal x not in T that is closest
     * to a vertex in T"
       */
      x = 0;                          /* x --> Next Terminal Vertex */
      int min = INT_MAX;

      for(i=1;i<K;i++)
      {
            if((min>dist[terminal_vertices[i]]) &&
              (dist[terminal_vertices[i]] != 0) &&
              ((is_processed[terminal_vertices[i]]) == 0))
            {
                  min = dist[terminal_vertices[i]];
                  x = terminal_vertices[i];
            }
      }
      //printf("Next Terminal Vertex to be added to T is : \"%d\" \n",x);
      /*printf("T ---> ");
      for(i=0;i<y;i++)
      {
            printf("%d  ",T[i]);
      }*/

      /*
       * Step 2 b) of the algorithm starts here
       * "Finding Vertex in T which is closest to
```

```
 * Next Terminal Vertex to be added"
 */
//printf("\nStarted Processing T\n");
int min_cost = INT_MAX;
for(i=0;i<y;i++)
{
	cost = 0;
	//printf("%d\n",T[i]);
	dijkstra(graph,T[i]);
	//printSolution(dist, V, T[i]);
	d = 0;
	find_parent(parent, x);
	/*printf("a ---> ");
	for(j=0;j<d;j++)
	{
		printf("%d  ",a[j]);
	}
	printf("\n");*/

	for(j=0;j<d;j++)
	{
		if(j == 0)
		{
			if(is_added[T[i]][a[0]] == 0)
			{
				is_added[T[i]][a[0]] = 1;
				//printf("Edge %d -> %d is considered\n",T[i],a[0]);
				cost += graph[T[i]][a[0]];
			}
		}
		else
		{
			if(is_added[a[j-1]][a[j]] == 0)
			{
				is_added[a[j-1]][a[j]] = 1;
				//printf("Edge %d -> %d is considered\n",a[j-1],a[j]);
				cost += graph[a[j-1]][a[j]];
			}
		}
	}
	if(cost < min_cost)
	{
		min_cost = cost;
		source_vertex = T[i];
	}
	/*printf("Source Vertex : %d\n",T[i]);
	printf("Cost : %d\n",cost);*/
```

```
		for(int l=0;l<V;l++)
		{
			for(int m=0;m<V;m++)
			{
				is_added[l][m] = is_included[l][m];
			}
		}
	}
	/*printf("Ended Processing T\n");
	printf("Closest Source Vertex : %d\n",source_vertex);
	printf("Minimum Cost : %d\n",cost);*/

	/*
	 * x is connected with "source_vertex" in T
	 * "Adding to T the shortest path that connects x with T"
	 * Total Cost is calculated here
	 */
	dijkstra(graph,source_vertex);
	//printSolution(dist, V, source_vertex);

	d = 0;
	find_parent(parent, x);
	/*printf("a ---> ");
	for(j=0;j<d;j++)
	{
		printf("%d  ",a[j]);
	}
	printf("\n");*/
	for(j=0;j<d;j++)
	{
		if(j==0)
		{
			if(is_included[source_vertex][a[0]] == 0)
			{
				is_included[source_vertex][a[0]] = 1;
				total_cost += graph[source_vertex][a[0]];
				//printf("Edge %d\t%d is added\n",source_vertex,a[0]);

				if(is_processed[a[0]] == 0)
				{
					is_processed[a[0]] = 1;
					T[y] = a[0];
					y++;
				}
			}
		}
		else
		{
```

```
                        if(is_included[a[j-1]][a[j]] == 0)
                        {
                                is_included[a[j-1]][a[j]] = 1;
                                total_cost += graph[a[j-1]][a[j]];
                                //printf("Edge %d\t%d is added\n",a[j-1],a[j]);
                                if(is_processed[a[j-1]] == 0)
                                {
                                        is_processed[a[j-1]] = 1;
                                        T[y] = a[j-1];
                                        y++;
                                }
                                if(is_processed[a[j]] == 0)
                                {
                                        is_processed[a[j]] = 1;
                                        T[y] = a[j];
                                        y++;
                                }
                        }
                }
        }
        count++;
   }

   for(i=0;i<y;i++)
   {
          printf("Vertex %d is present in T\n",T[i]);
   }

   for(i=0;i<V;i++)
   {
        for(j=0;j<V;j++)
        {
                if(is_included[i][j] == 1)
                {
                        printf("Edge %d <-> %d is in T\n",i,j);
                }
        }
   }
   printf("Total Cost : %d\n",total_cost);
   return 0;
}
```

4.4 Code to implement Multiple Unicast

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

struct Edge
{
   /* This structure is equal to an edge. Edge contains two end points. These edges are
directed edges so they contain source and destination and some weight. These 3 are
elements in this structure*/
   int source, destination, weight;
};

// a structure to represent a connected, directed and weighted graph
struct Graph
{
   int V, E;
         // V is number of vertices and E is number of edges

   struct Edge* edge;
         // This structure contain another structure which we already created edge.
};

struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
         //Allocating space to structure graph

   graph->V = V;   //assigning values to structure elements that taken form user.

   graph->E = E;

   graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
         //Creating "Edge" type structures inside "Graph" structure, the number of edge
type structures are equal to number of edges

   return graph;
}

void FinalSolution(int dist[], int n)
{
         // This function prints the final solution
   printf("\nVertex\tDistance from Source Vertex\n");
   int i;

   for (i = 0; i < n; ++i){
                  printf("%d \t\t %d\n", i, dist[i]);
```

```
    int i;

    for (i = 0; i < n; ++i){
                printf("%d \t\t %d\n", i, dist[i]);
        }
}

void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;

    int E = graph->E;

    int StoreDistance[V];

    int i,j;

    // This is initial step that we know , we initialize all distance to infinity except source.
        // We assign source distance as 0(zero)

    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    //The shortest path of graph that contain V vertices, never contain "V-1" edges. So we
do here "V-1" relaxations
    for (i = 1; i <= V-1; i++)
    {
        for (j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;

            int v = graph->edge[j].destination;

            int weight = graph->edge[j].weight;

            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }

    // Actually upto now shortest path found. But BellmanFord checks for negative edge
cycle. In this step we check for that
    // shortest distances if graph doesn't contain negative weight cycle.

    // If we get a shorter path, then there is a negative edge cycle.
```

```
    for (i = 0; i < E; i++)
    {
        int u = graph->edge[i].source;

        int v = graph->edge[i].destination;

        int weight = graph->edge[i].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])

            printf("This graph contains negative edge cycle\n");
    }

    FinalSolution(StoreDistance, V);
    return;
}

int main()
{
    int V,E,S;  //V = no.of Vertices, E = no.of Edges, S is source vertex

            printf("Enter number of vertices in graph\n");
    scanf("%d",&V);

            printf("Enter number of edges in graph\n");
    scanf("%d",&E);

            printf("Enter your source vertex number\n");
            scanf("%d",&S);

    struct Graph* graph = createGraph(V, E);    //calling the function to allocate space to
these many vertices and edges

    int i;
    for(i=0;i<E;i++){
        printf("\nEnter edge %d properties Source, destination, weight respectively\n",i+1);
        scanf("%d",&graph->edge[i].source);
        scanf("%d",&graph->edge[i].destination);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);
            //passing created graph and source vertex to BellmanFord Algorithm function

    return 0;
}
```

5. OUTPUT

5.1 Output of Flooding (Broadcast Routing)

```
Broadcast Routing(Flooding) Example:
Starting from Source vertex '0':
0 1 2 3 6 4 5
```

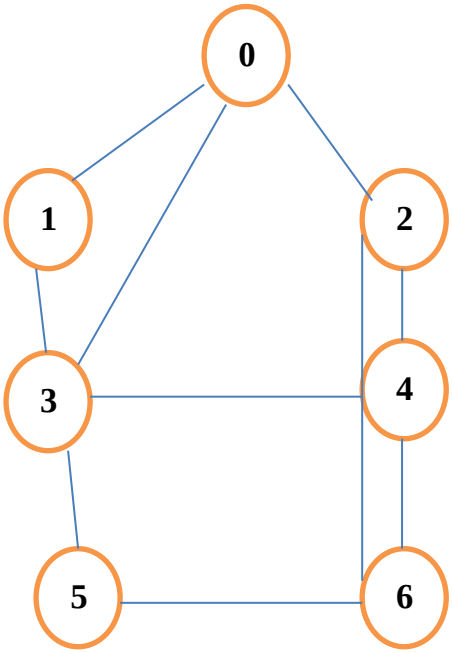

Figure 9- Original Network

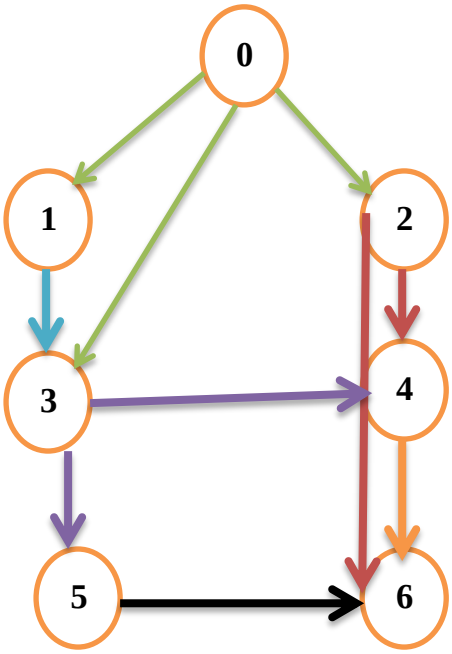

Figure 10- Flooding in network(Broadcast Routing)

5.2 Output of Sink/ Spanning Tree (Broadcast Routing)

```
Sink / Spanning Tree(Broadcast Routing) Example:
Enter number of Nodes : 6

Enter edge 1(-1 -1 to quit) : 0 1

Enter weight for this edge : 1

Enter edge 2(-1 -1 to quit) : 0 2

Enter weight for this edge : 9

Enter edge 3(-1 -1 to quit) : 0 5

Enter weight for this edge : 14

Enter edge 4(-1 -1 to quit) : 1 2

Enter weight for this edge : 10

Enter edge 5(-1 -1 to quit) : 1 3

Enter weight for this edge : 15
Enter edge 6(-1 -1 to quit) : 2 3

Enter weight for this edge : 11

Enter edge 7(-1 -1 to quit) : 2 5

Enter weight for this edge : 2

Enter edge 8(-1 -1 to quit) : 3 4

Enter weight for this edge : 6

Enter edge 9(-1 -1 to quit) : 4 5

Enter weight for this edge : 9

Enter edge 10(-1 -1 to quit) : -1 -1

Enter Source(root) vertex : 0
```

```
Edges to be included in Spanning(Sink) tree are :
0-> 1
0-> 2
2-> 5
5-> 4
4-> 3

Weight of spanning tree is : 27
```

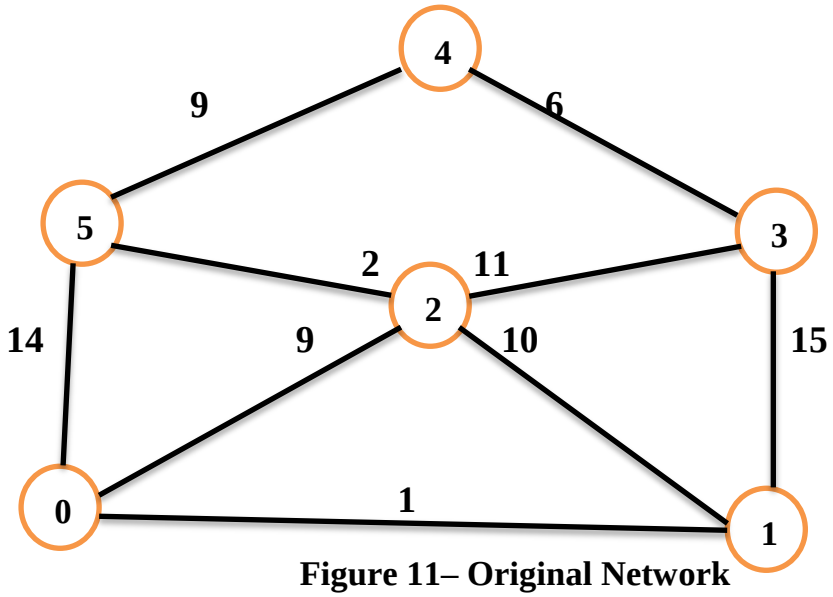

Figure 11– Original Network

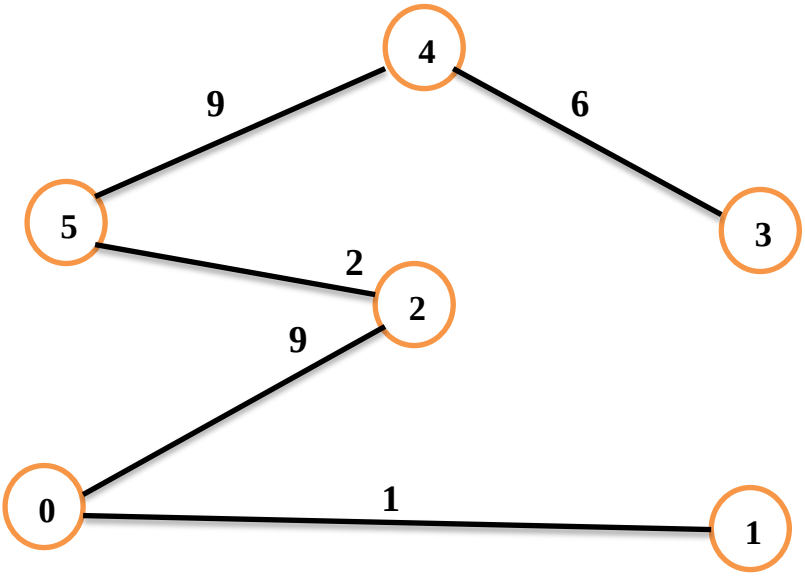

Figure 12– Minimum Spanning Tree

5.3 Output of Source Based Tree/Steiner Tree (Multicast Routing)

```
Multicast Routing Example (Steiner Tree):
Vertex 1 is present in T
Vertex 6 is present in T
Vertex 4 is present in T
Vertex 3 is present in T
Vertex 5 is present in T
Edge 1 <-> 6 is in T
Edge 4 <-> 3 is in T
Edge 4 <-> 5 is in T
Edge 6 <-> 4 is in T
Total Cost : 12
```

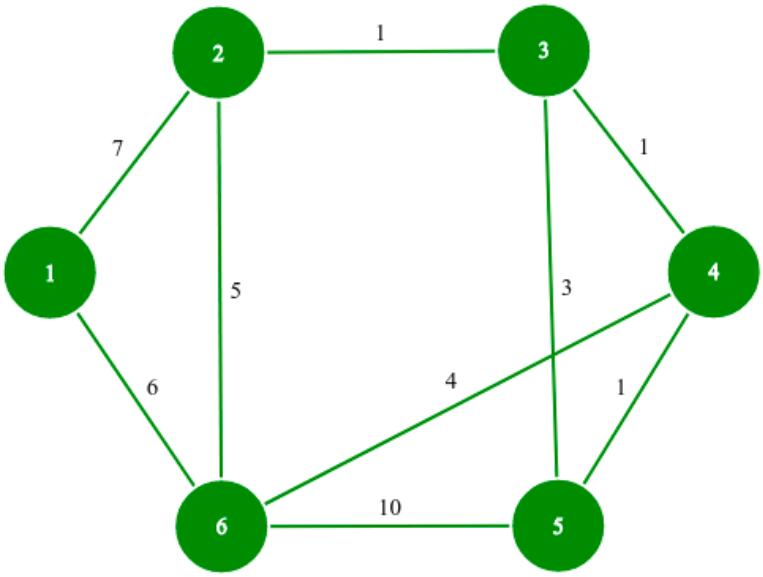

Figure 13 – Original Network

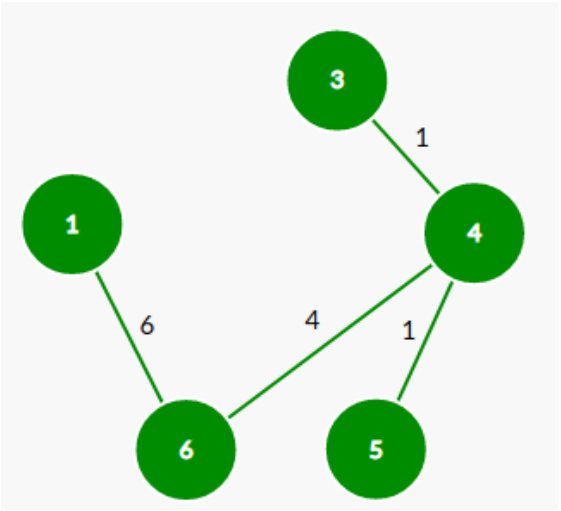

Figure 14- Steiner Tree (Source Based Multicast Routing)

5.4 Output of Multiple Unicast

```
Enter number of vertices in graph
5
Enter number of edges in graph
7
Enter your source vertex number
0

Enter edge 1 properties Source, destination, weight respectively
0 1 6

Enter edge 2 properties Source, destination, weight respectively
0 2 5

Enter edge 3 properties Source, destination, weight respectively
2 1 -2

Enter edge 4 properties Source, destination, weight respectively
1 3 -1

Enter edge 5 properties Source, destination, weight respectively
2 3 4

Enter edge 6 properties Source, destination, weight respectively
2 4 3

Enter edge 7 properties Source, destination, weight respectively
3 4 3

Vertex  Distance from Source Vertex
0               0
1               3
2               5
3               2
4               5
```

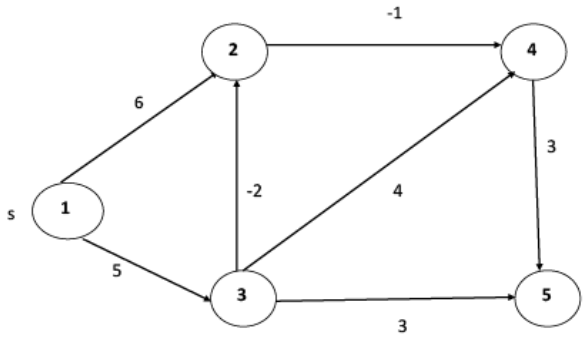

Figure 15- Original Network

6. OBSERVATIONS

6.1 Broadcast Routing

In broadcast routing, the network layer provides a service of delivering a packet sent from a source node to all other nodes in the network.

- **Flooding:** The most obvious technique for achieving broadcast is a flooding. The source node sends a copy of the packet to all of its neighbours. When a node receives a broadcast packet, it duplicates the packet and forwards it to all of its neighbours (except the neighbour from which it received the packet). This scheme eventually delivers a copy of the broadcast packet to all nodes if they are connected.

  Disadvantages:

  - If the graph has cycles, then one or more copies of each broadcast packet will cycle indefinitely.
  - When a node is connected to more than two other nodes, then it could result in broadcast storm (resulting from the endless multiplication of broadcast packets). To avoid Broadcast storm controlled flooding can be done.

- **Reverse path forwarding (RPF) / reverse path broadcast (RPB):** When a router receives a broadcast packet with a given source address, it transmits the packet on its entire outgoing links (except the one on which it was received). The router keeps the packet only if the packet arrived on the link that is on its own shortest unicast path back to the source. Otherwise, the router simply discards the incoming packet. RPF does not use unicast routing to actually deliver a packet to a destination, nor does it require that a router know the complete shortest path from itself to the source. RPF need only know the next neighbour on its unicast shortest path to the sender. The principal advantage of reverse path forwarding is that it is efficient while being easy to implement.

- **Spanning Tree Broadcast:** While controlled flooding (sequence number) and RPF avoid broadcast storms, they do not completely avoid the transmission of redundant broadcast packets. The solution of this problem is **spanning tree (**a tree that contains each and every node in a graph). So, first a spanning tree is constructed and when a source node wants to send a broadcast packet, it sends the packet out on all of the incident links that belong to the spanning tree. Not only does spanning tree eliminate redundant broadcast packets, but once in place, the spanning tree can be used by any node to begin a broadcast. In this algorithm, a node need not be aware of the entire tree; it simply needs to know

which of its neighbours in *G* are spanning-tree neighbours. The main complexity associated with the spanning-tree based broadcast approach is the **creation and maintenance of the spanning tree**.

6.2 Multicast Routing

In multicast routing, a single source node can send a copy of a packet to a subset of the other network nodes. The goal of multicast routing is the need to create routing trees to optimally route the packets from a source to the destinations belonging to the multicast group.

- **Source-Based Tree Approach:** In this each router needs to create a separate tree for each source-group combination. In each tree, the corresponding source is the root, the members of the group are the leaves, and the router itself is somewhere on the tree.

- **Group-Shared Tree Approach:** In this we designate a router to act as the dummy source for each group. The designated router (called as core router) acts as the representative for the group. Any source that has a packet to send to a member of that group, first, sends it to the core router (unicast communication) and then the core router is responsible for multicasting.

6.3 Multicasting Vs Multiple Unicasting

Multicasting

- It starts with a single packet from source that is duplicated by the routers.
- The destination address in each packet is the same for all duplicates.
- Only a single copy of the packet travels between any two routers.
- IP Multicast uses UDP for communication, therefore it is unreliable.

Multiple Unicasting

- In this several packets start from the source.
- If there are three destinations, the source sends three packets, each with a different unicast destination address.
- Note that there may be multiple copies traveling between two routers.

7. CONCLUSION

This mini project details the great potential that networking has. I was able to implement different broadcast and multicast routing protocols. The routing protocol is a routing algorithm that provides the best path from the source to the destination. In unicast routing, there is one source and one destination node i.e. point-to-point communication. The relationship between the source and the destination network is one to one. Each router in the path tries to forward the packet to one and only one of its interfaces. In broadcast routing, the network layer provides a service of delivering a packet sent from a source node to all other nodes in the network. In multicast routing, a single source node can send a copy of a packet to a subset of the other network nodes. Multicast routing is special case of broadcast routing with significance difference and challenges. In broadcast routing, packets are sent to all nodes even if they do not want it. But in Multicast routing, the data is sent to only nodes which wants to receive the packets.

This study helps me to realize the advantages and disadvantages of different routing protocols. Flooding generates duplicate packets and can cause Broadcast storm. The solution to this problem is Spanning Tree i.e. a tree which connects all the nodes and when the cost of traversal is minimum then it is called Minimum Spanning Tree. But there is a complexity associated with the spanning-tree based broadcast approach that is the creation and maintenance of the spanning tree. Multicast routing can be implemented by using Steiner Tree approach.

Thus I was able to execute different broadcast and multicast routing algorithms.

8. LEARNING OUTCOME

Following are the learning outcomes of mine by this mini project that is “Design and develop Broadcast and Multicast routing algorithms”:

- After this experiment, I am familiar with different types of broadcast and multicast routing algorithms.
- I am also able to write program for these algorithms.
- I am able to solve numerical related to these algorithms.
- After this experiment, I am able to classify broadcast and multicast routing algorithms.
- I am able to identify broadcast and multicast routing algorithms.
- After this experiment, I can apply my learnings when I want to develop a routing protocol or a network.
- Having a solid understanding of routing algorithms can help me design and built a network that is secure.

9. REFERENCES

[1] Tanenbaum S. Andrew and David J. Wetherall, Computer Networks (Vol. 5), Pearson Education Inc., 2011.

[2] Forouzan A. Behrouz, Data Communications And Networking (Vol.4), McGraw-HilI Forouzan networking series, 2007.

[3] S.E. Deering and D.R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," ACM Transactions on Computer Systems, vol. 8, no. 2, May 1990, pp. 85-110.

[4] S. L. Hakimi, "Steiner's problem in graphs and its implications", Networks, vol. 1, pp. 113-133, 1971.

[5] B. K. Kadaba and J. M. Jaffe, "Routing to multiple destinations in computer networks", IEEE Trans. Commun., vol. COM-31, no. 3, pp. 343-351, Mar. 1983.

[6] J. B. Kruksal, "On the Shortest Spanning Subtree of A Graph and the Traveling Salesman Problem", Proc. Amer. Math. Soc., vol. 7, pp. 48-50, 1956.

[7] Y. K. Dalal and R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets", Commun. ACM, vol. 21, pp. 1040-1043, Dec. 1973.

[8] T. Ballardie, Core Based Trees (CBT version 2) Multicast Routing, Sept. 1997.

[9] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs", Mathematica Japonica, vol. 24, no. 6, pp. 573-577, 1980.

[10] P. Winter, "Steiner Problem in Networks: A Survey", *Networks*, vol. 17, pp. 129-167, 1987.

.