Experiment 6

GRAPH PRACTICE PROBLEMS

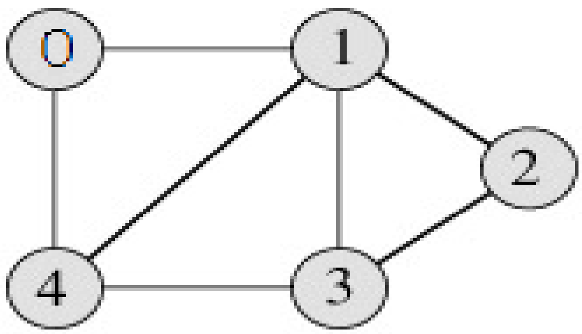
-Ruchir Bhaskar

16BCE0190

Q1) Use adjaceny matrix and adjacency list representation to express a computer networks with maximum degree of 3 for each node. Consider number of nodes in the networks as 5 interconnected by links.

FOR ADJACENCY MATRIX-

graph used -



Adjacency Matrix Representation of the above graph-

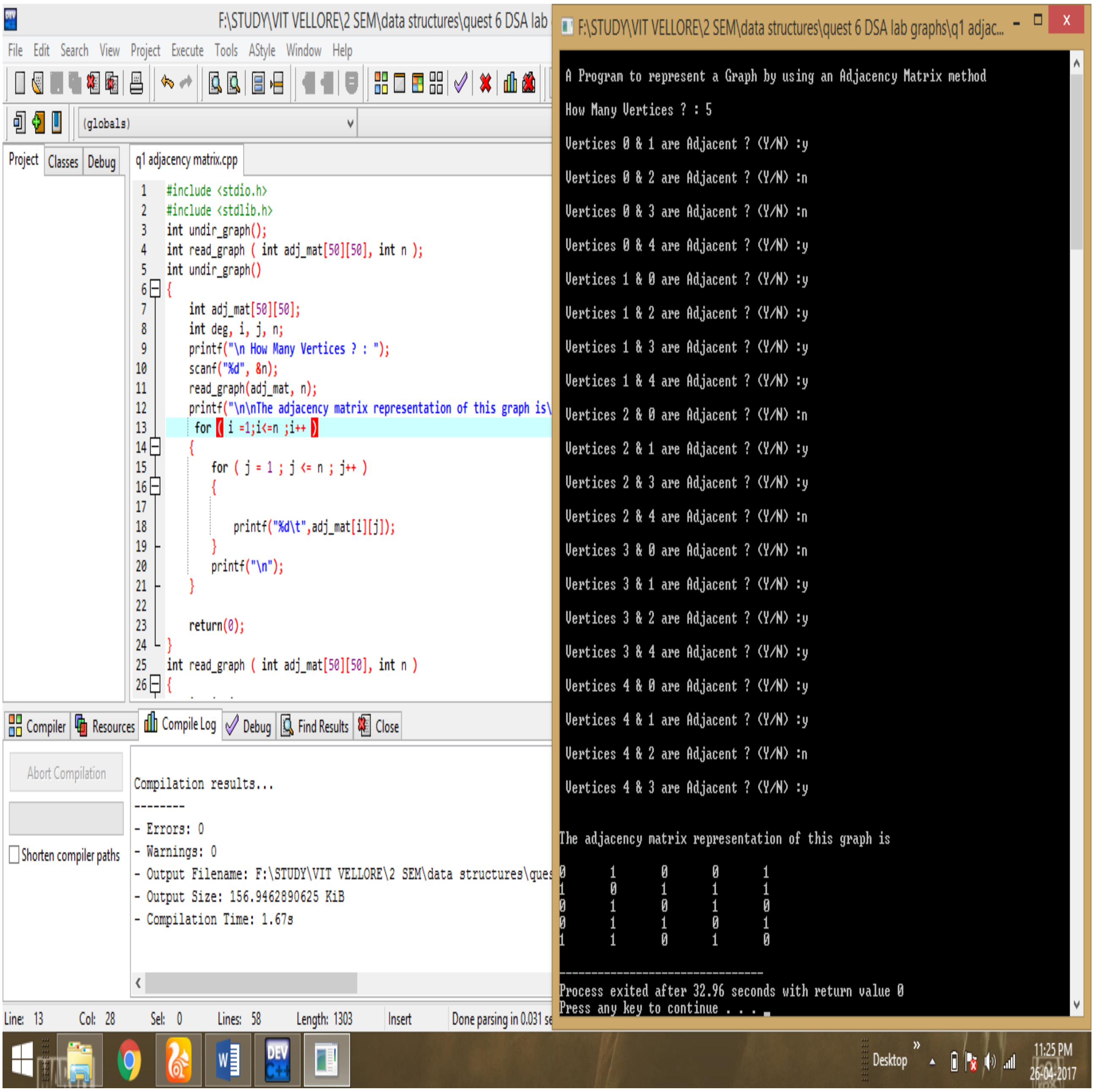
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

C/C++ Code

```
#include <stdio.h>
#include <stdlib.h>
int undir_graph();
int read_graph ( int adj_mat[50][50], int n );
int undir_graph()
{
    int adj_mat[50][50];
    int deg, i, j, n;
    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);
    read_graph(adj_mat, n);
    printf("\n\nThe adjacency matrix representation of this graph is\n\n");
            for ( i =1;i<=n ;i++ )
    {
        for ( j = 1 ; j <= n ; j++ )
        {

             printf("%d\t",adj_mat[i][j]);
        }
        printf("\n");
             }

    return(0);
}
int read_graph ( int adj_mat[50][50], int n )
{
    int i, j;
    char reply;
    for ( i =1;i<=n ;i++ )
    {
        for ( j = 1 ; j <= n ; j++ )
```

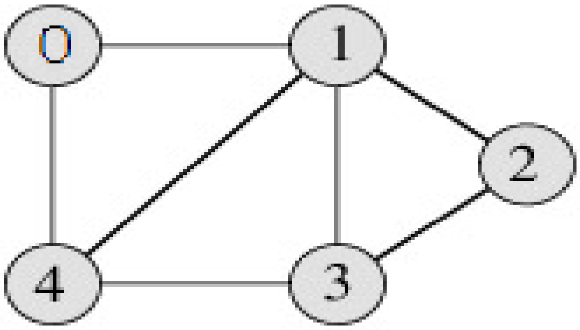
```
        {
            if ( i == j )
            {
                adj_mat[i][j] = 0;
                          continue;
            }
            printf("\n Vertices %d & %d are Adjacent ? (Y/N) :",i-1,j-1);
            fflush(stdin);
                                        scanf("%c", &reply);
            fflush(stdin);
            if ( reply == 'y' || reply == 'Y' )
                adj_mat[i][j] = 1;
            else
                adj_mat[i][j] = 0;
             }
    }
    return(0);
}

int main()
{
    printf("\n A Program to represent a Graph by using an ");
             printf("Adjacency Matrix method \n ");
             undir_graph();
return(0);
}
```

OUTPUT SCREENSHOT-

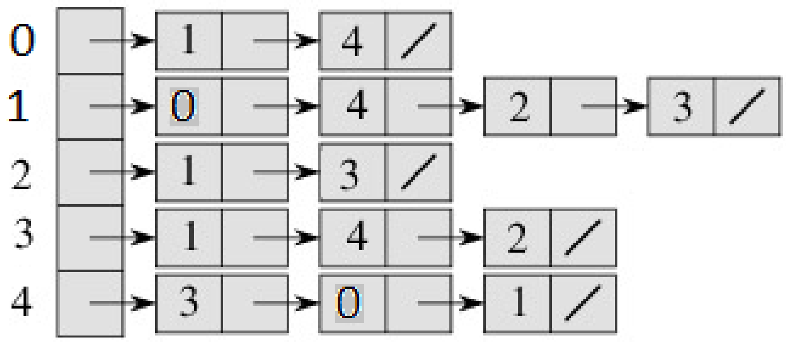


FOR ADJACENCY LIST-

graph used -



Adjacency List Representation of the above Graph-



C/C++ Code-

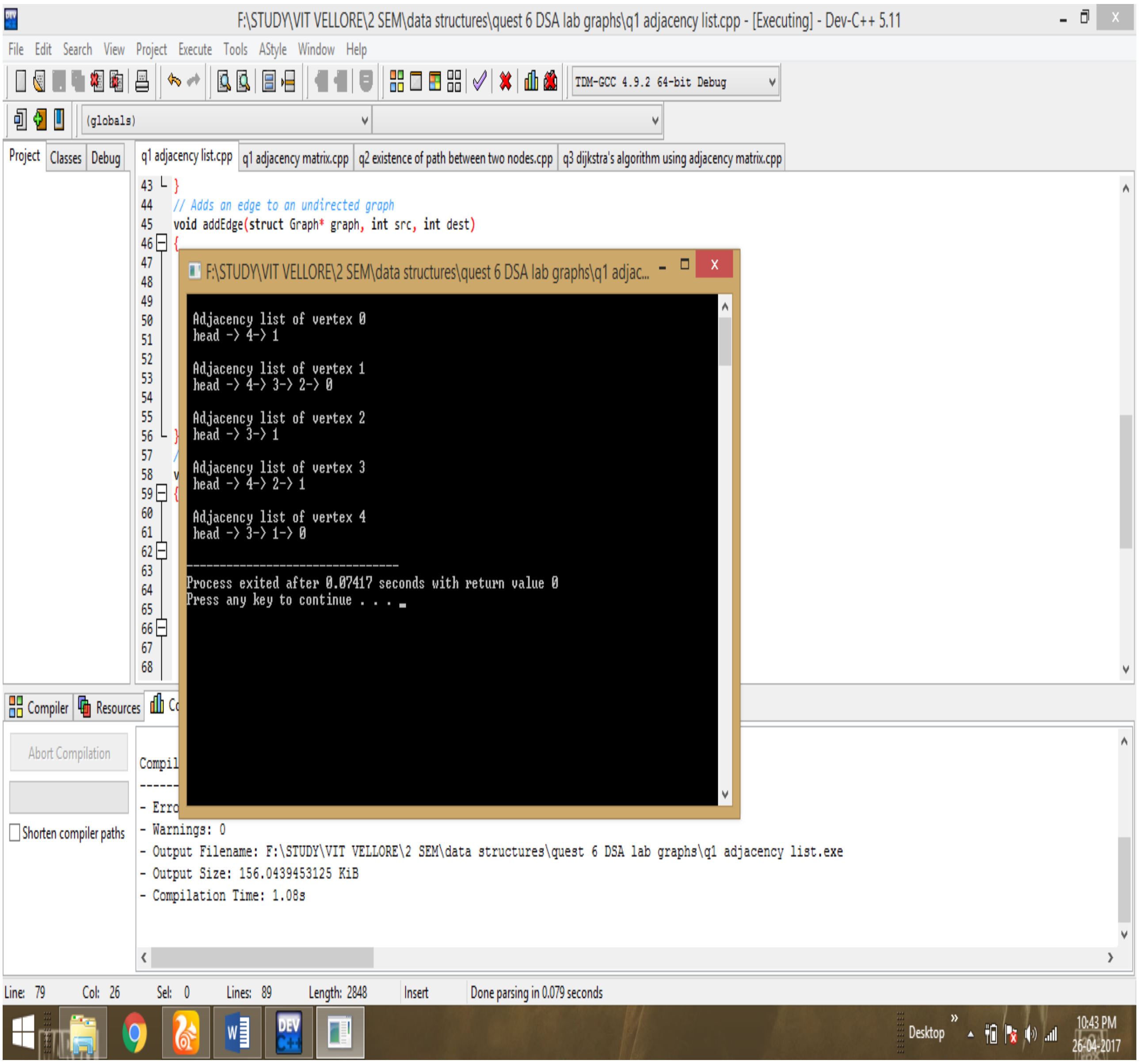
```
// A C Program to demonstrate adjacency list representation of graphs
#include <stdio.h>
#include <stdlib.h>
// A structure to represent an adjacency list node
struct AdjListNode
{
   int dest;
   struct AdjListNode* next;
};
// A structure to represent an adjacency list
```

```
struct AdjList
{
    struct AdjListNode *head;  // pointer to head node of list
};
// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};
// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
    struct AdjListNode* newNode =
            (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    // Create an array of adjacency lists.  Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));
     // Initialize each adjacency list as empty by making head as NULL
    int i;
    for (i = 0; i < V; ++i)
```

```
        graph->array[i].head = NULL;
    return graph;
}
// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest.  A new node is added to the adjacency
    // list of src.  The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}
// A utility function to print the adjacenncy list representation of graph
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
```

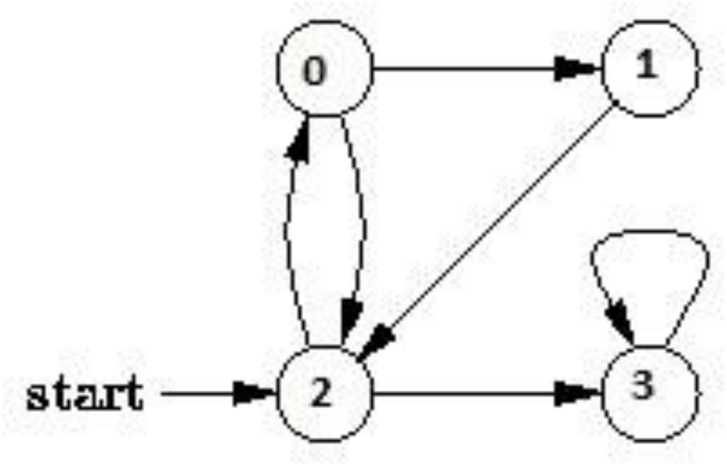
```
    }
}
// Driver program to test above functions
int main()
{
    // create the graph given in above fugure
    int V = 5;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);
    // print the adjacency list representation of the above graph
    printGraph(graph);
    return 0;
}
```

OUTPUT SCREENSHOT-



Q2) Write a function to find existence of a path between any two given nodes in the network.

input graph-



C- Code

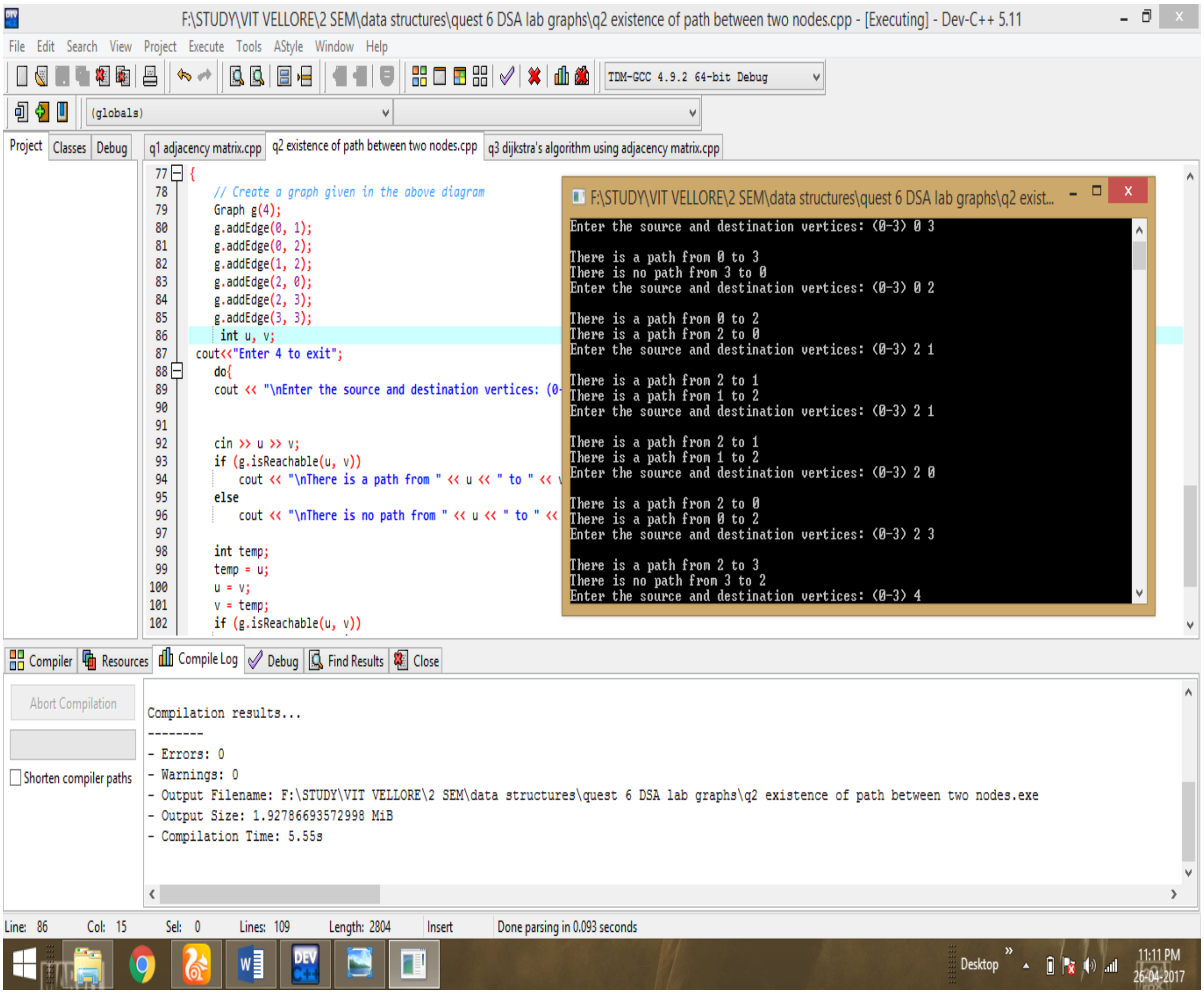
```
#include <iostream>
#include <list>
using namespace std;
// This class represents a directed graph using adjacency list representation
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
  public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    bool isReachable(int s, int d); // returns true if there is a path from s to d
};
```

```
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.
}

// A BFS based function to check whether d is reachable from s.
bool Graph::isReachable(int s, int d)
{
    // Base case
    if (s == d)
        return true;

    // 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);

    // it 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();
        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 this adjacent node is the destination node, then return true
            if (*i == d)
                return true;

            // Else, continue to do BFS
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }

    return false;
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
```

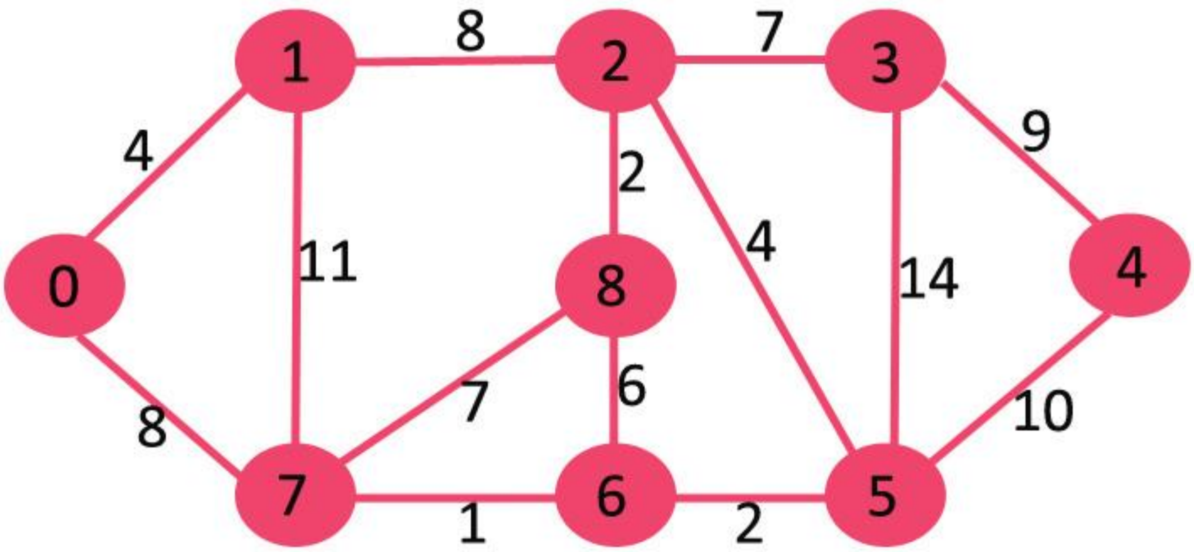
```
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
int u, v;
 cout<<"Enter 4 to exit";
            do{
            cout << "\nEnter the source and destination vertices: (0-3) ";
            cin >> u >> v;
    if (g.isReachable(u, v))
        cout << "\nThere is a path from " << u << " to " << v;
    else
        cout << "\nThere is no path from " << u << " to " << v;

    int temp;
    temp = u;
    u = v;
    v = temp;
    if (g.isReachable(u, v))
        cout << "\nThere is a path from " << u << " to " << v;
    else
        cout << "\nThere is no path from " << u << " to " << v;

 }while(u!=4);
    return 0;
}
```

OUTPUT SCREENSHOT-



Q3) Write a program to find shortest path between source node s to the rest of n- 1 nodes using Dijkstra's shortest path algorithm. To represent the n node network with atmost m links, use adjacency matrix representation.

input graph-



C- Code

```
// A C / C++ program for Dijkstra's single source shortest
// path algorithm. The program is for adjacency matrix
// representation of the graph.
#include <stdio.h>
#include <limits.h>
// Number of vertices in the graph
#define V 9
// 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;
}

// Function to print shortest path from source to j
// using parent array
void printPath(int parent[], int j)
{
    // Base Case : If j is source
    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 parent[])
{
    int src = 0;
    printf("Vertex\t\tDistance\tPath\n");
    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);
```

```
    }
}

// Funtion 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)
{
    int dist[V];  // The output array. dist[i] will hold
                  // the shortest distance from src to i

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

    // Parent array to store shortest path tree
    int parent[V];

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

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

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
```

```
        // Pick the minimum distance vertex from the set of
        // vertices not yet processed. u is always equal to src
        // in 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] + graph[u][v] < dist[v])
            {
                parent[v]  = u;
                dist[v] = dist[u] + graph[u][v];
            }
    }

    // print the constructed distance array
    printSolution(dist, V, parent);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
```

```
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                       {4, 0, 8, 0, 0, 0, 0, 11, 0},
                       {0, 8, 0, 7, 0, 4, 0, 0, 2},
                       {0, 0, 7, 0, 9, 14, 0, 0, 0},
                       {0, 0, 0, 9, 0, 10, 0, 0, 0},
                       {0, 0, 4, 0, 10, 0, 2, 0, 0},
                       {0, 0, 0, 14, 0, 2, 0, 1, 6},
                       {8, 11, 0, 0, 0, 0, 1, 0, 7},
                       {0, 0, 2, 0, 0, 0, 6, 7, 0}
                      };

    dijkstra(graph, 0);

    return 0;
}
```

OUTPUT SCREENSHOT-