Project Report on

ALGEBRAIC REPRESENTATION OF GRAPH

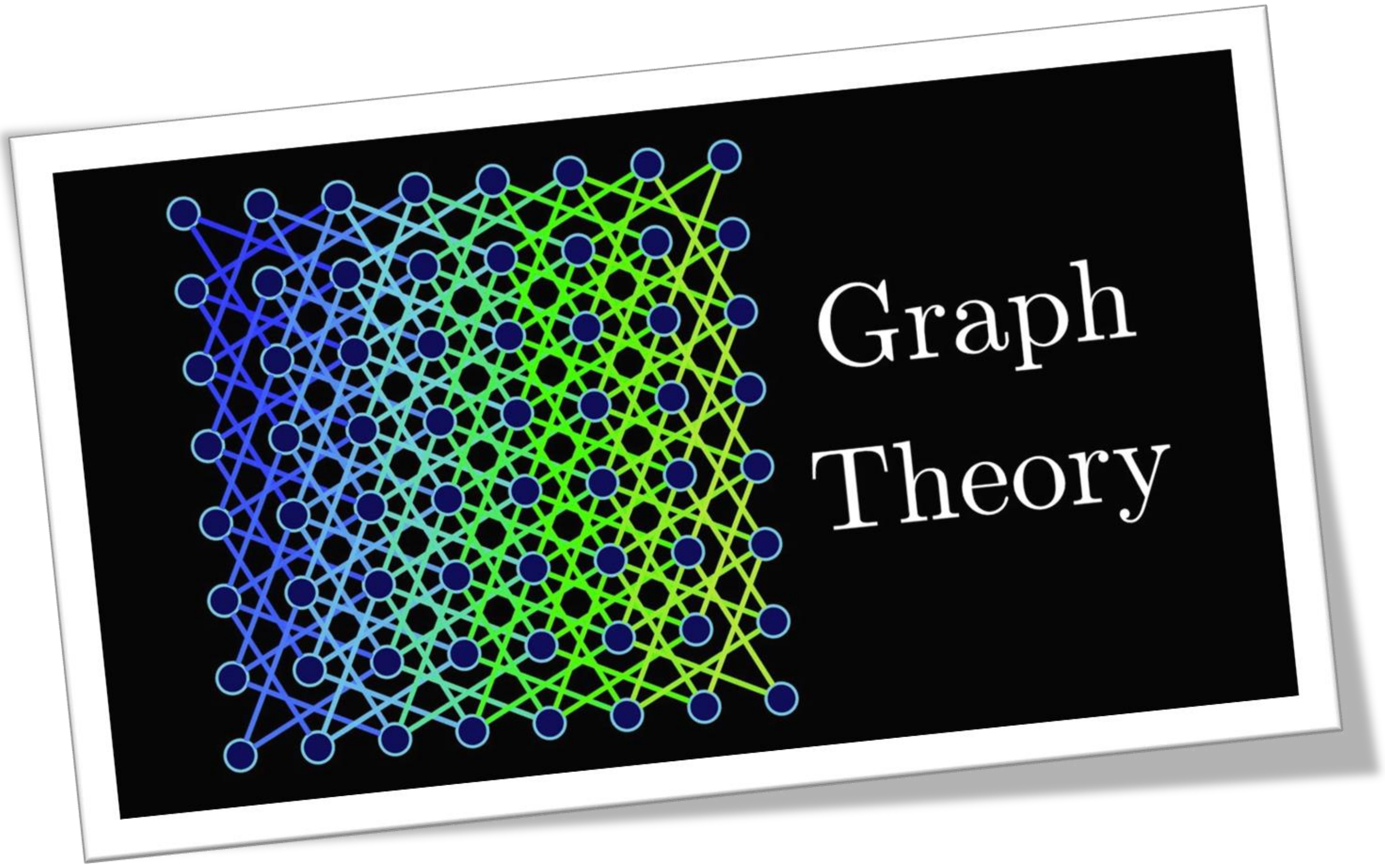
Submitted by:

Aneesh Panchal - 2K20/MC/21
Ayushi sagar - 2K20/MC/35

Submitted to:

Ms. Priyanka Goel

Department of Applied Mathematics
Delhi Technological University



Certificate

I hereby certify that the project dissertation titled “Algebraic Representation of Graph” which is submitted by Aneesh Panchal (2K20/MC/21) and Ayushi Sagar (2K20/MC/35) of Mathematics and Computing Department, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Bachelor of Technology, is a record of the project work carried out by the students. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this university or elsewhere.

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

Acknowledgement

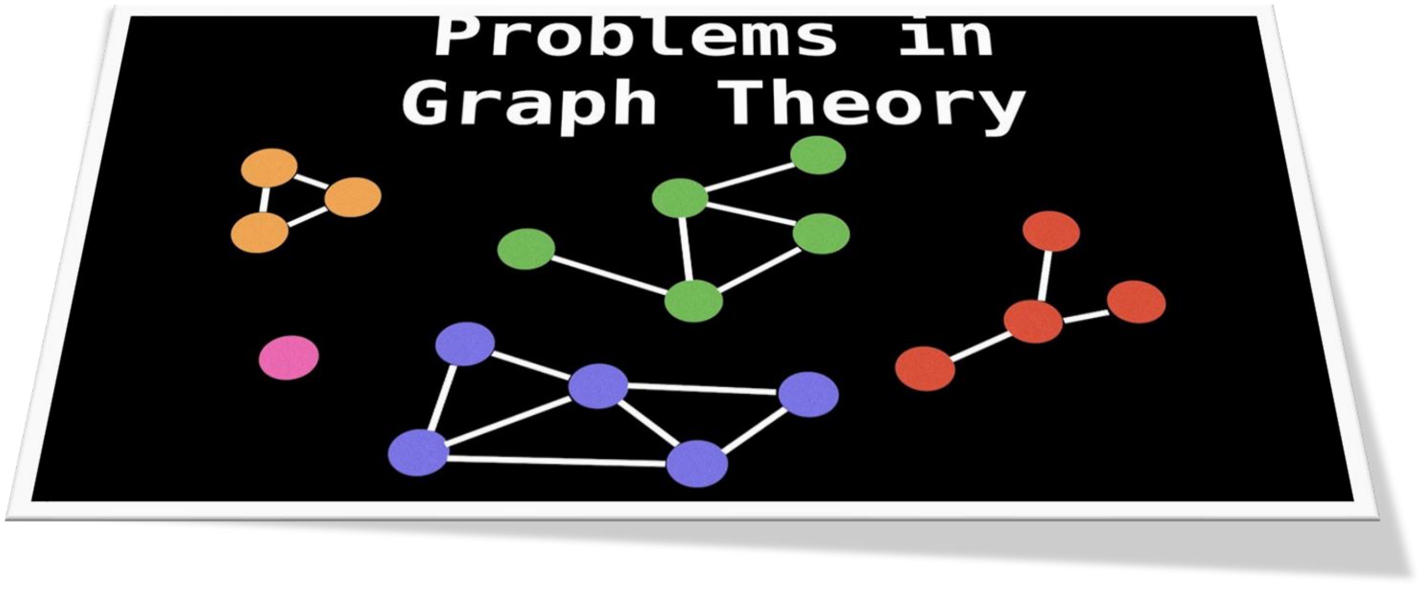
We, Aneesh Panchal(2K20/MC/21) and Ayushi Sagar (2K20/MC/35) would like to express our special thanks of gratitude to Ms. Priyanka Goel, Mathematics and Computing Department, Delhi Technological University for his able guidance and support in completing our project report.

We came to know about many new things, we are really thankful to him. We would also like to thank our classmates who have also helped whenever we were stuck at some point.

Thanking You

Aneesh Panchal (2K20/MC/21)
Ayushi Sagar (2K20/MC/35)

Index



Abstract

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

Role of Graphs :

- Extremely important in computer science and mathematics
- Numerous important applications
- Modeling the concept of binary relation

As we can understand graph through their pictorial version but for the machines, we need to provide some method to machines so that it can understand the graphs.

So one of the method is through matrices which we discuss below in detail. The Adjaceny matrix is used to let the machines understand that whatever we are providing is a graph or not.

So our project contains some theoretical knowledge of graphs and a code to understand how machines understand these.

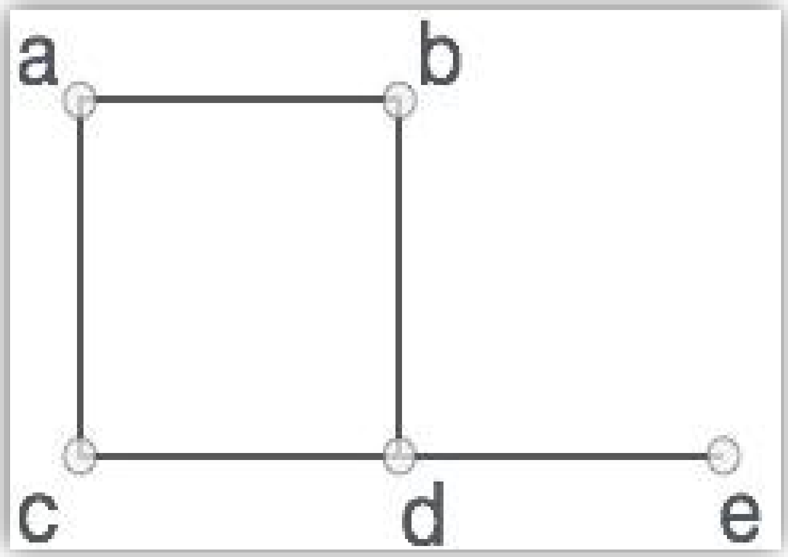
Introduction:

A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related".
It is a mathematical structure consisting if two finite sets V and E. The elements of V are called vertices or nodes and the elements of E are called edges. Each edge is associated with a set consisting of either one or two vertices called its endpoints.
The correspondence from edges to endpoints is called edge-endpoint function.

A graph G consists of two sets V and E where E is some subset of (V 2). The set V is called the vertex set of G and E is called the edge set of G. In this case we write G = (V, E).
A graph (denoted as G=(V,E)) consists of a non-empty set of vertices or nodes V and a set of edges E.
Let G = (V, E) be a graph

- **Vertices** - The elements of V are called the vertices of G
- **Edges** - The elements of E are called the edges of G.

In the graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}

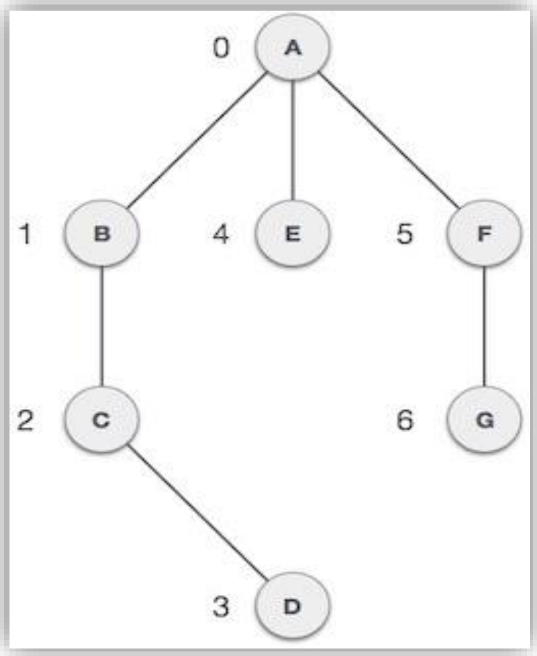


Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

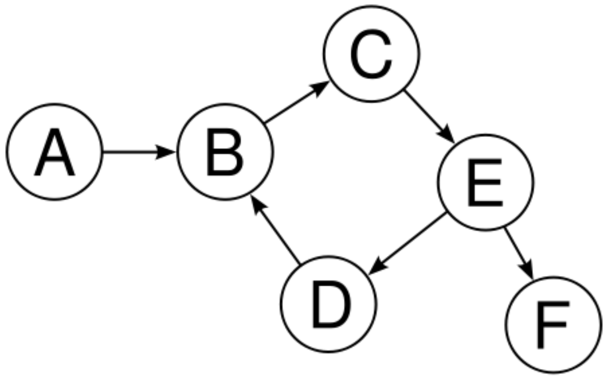
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.



❖ **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

DIRECTED GRAPH (DIGRAPH):

Directed graph (digraph) is an ordered pair: G = (V, E), where: V is the vertex set E is the edge set (or arc set) each edge e = (v,w) in E is an ordered pair of vertices from V, called the tail and head end of the edge e, respectively.

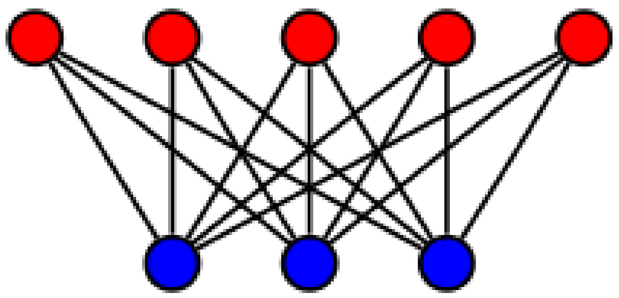


PICTURE OF A GRAPH:

A given graph can be depicted on a plane (or other 2-dimensional surface) in multiple ways. A picture is only a visual form of representation of a graph. It is necessary to distinguish between an abstract (mathematical) concept of a graph and its picture (visual representation).

BIPARTITE GRAPH (BIGRAPH):

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph.

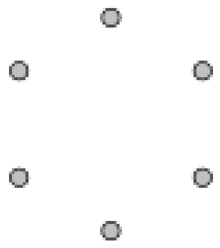


REGULAR GRAPH:

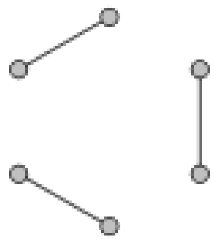
In graph theory, a regular graph is a graph where each vertex has the same number of neighbours i.e. every vertex has the same degree or valency.
A regular directed graph must also satisfy the stronger condition that the indegree and outdegree of each vertex are equal to each other.
A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k. A regular graph contains an even number of vertices with odd degree.

EXAMPLES OF REGULAR GRAPHS:

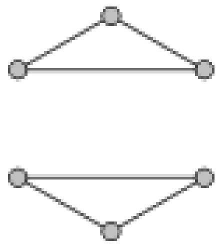
0 – Regular Graph



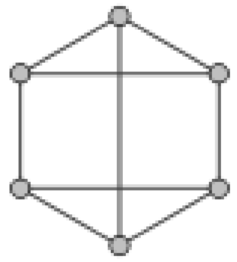
1 – Regular Graph



2 – Regular Graph



3 – Regular Graph



MATRIX REPRESENTATION OF GRAPH:

A graph can be represented inside a computer by using adjacency matrices.

ADJACENCY MATRIX:

In graph theory, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

Adjacency Matrix is a 2D array of size V x V,
where V is the number of vertices in a graph.

Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric.

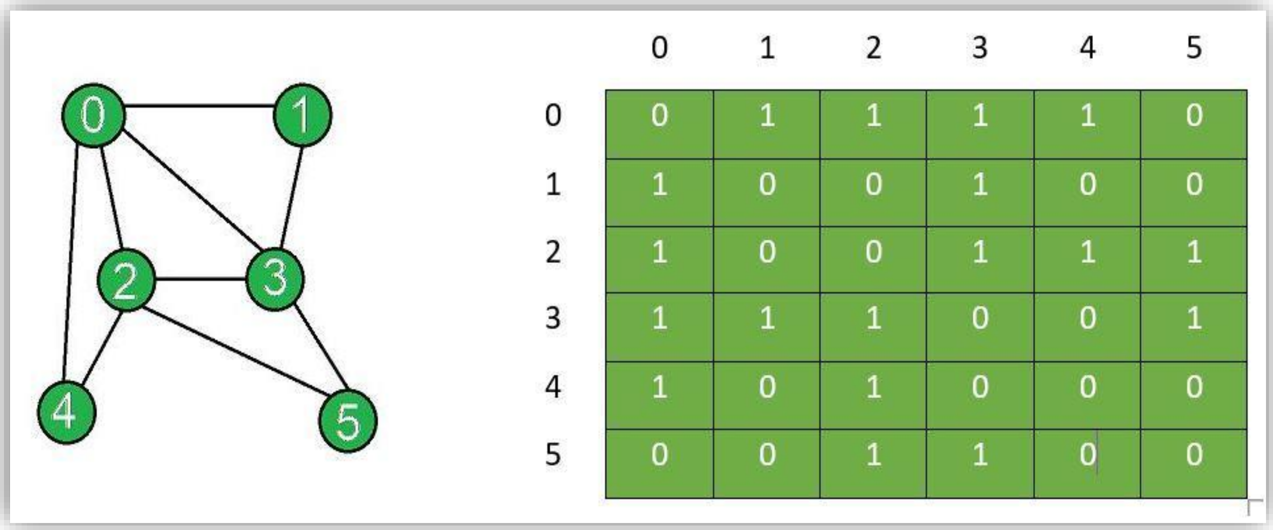
Adjacency Matrix is also used to represent weighted graphs.

If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

For a graph G = (V, E), having n vertices its adjacency matrix is a square matrix A having n rows and columns indexed by the vertices so that

A[i, j] = 1 ⇔ vertices i, j are adjacent,

else A[i, j] = 0. (in case of self-loop (i, i), A[i, i] = 2)



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

It is a matrix A[n][n] where n is number of vertices and

A[i][j] = 1 if i & j are adjacent

= 0 otherwise

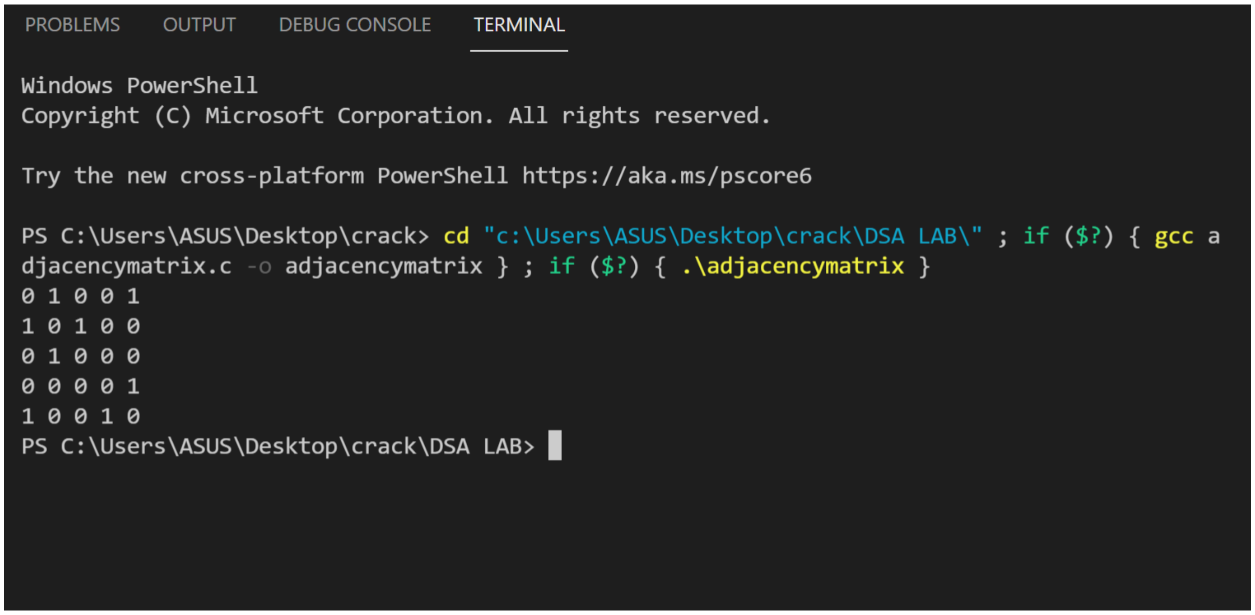
Space complexity O(N^2)

Program to implement Adjacency Matrix of a given Graph:

```
// MC201
// 2K20MC21 & 2K20MC35

#include <stdio.h>

// N vertices and M Edges
int N, M;

// Function to create Adjacency Matrix
void createAdjMatrix(int Adj[][N + 1],int arr[][2]) {
    // Initialise all value to this
    // Adjacency list to zero
    for (int i = 0; i < N + 1; i++) {
        for (int j = 0; j < N + 1; j++) {
            Adj[i][j] = 0;
        }
    }
```

```c
    // Traverse the array of Edges
    for (int i = 0; i < M; i++) {

        // Find X and Y of Edges
        int x = arr[i][0];
        int y = arr[i][1];

        // Update value to 1
        Adj[x][y] = 1;
        Adj[y][x] = 1;
    }
}

// Function to print the created
// Adjacency Matrix
void printAdjMatrix(int Adj[][N + 1])
{
    // Traverse the Adj[][]
    for (int i = 1; i < N + 1; i++) {
        for (int j = 1; j < N + 1; j++) {

            // Print the value at Adj[i][j]
            printf("%d ", Adj[i][j]);
        }
        printf("\n");
    }
}

// Driver Code
int main()
{
   // Number of vertices
   N = 5;
   // Given Edges
    int arr[][2]
        = { { 1, 2 }, { 2, 3 },
            { 4, 5 }, { 1, 5 } };
    // Number of Edges
    M = sizeof(arr) / sizeof(arr[0]);

    // For Adjacency Matrix
    int Adj[N + 1][N + 1];

    // Function call to create
    // Adjacency Matrix
    createAdjMatrix(Adj, arr);

    // Print Adjacency Matrix
    printAdjMatrix(Adj);

    return 0;
}
```

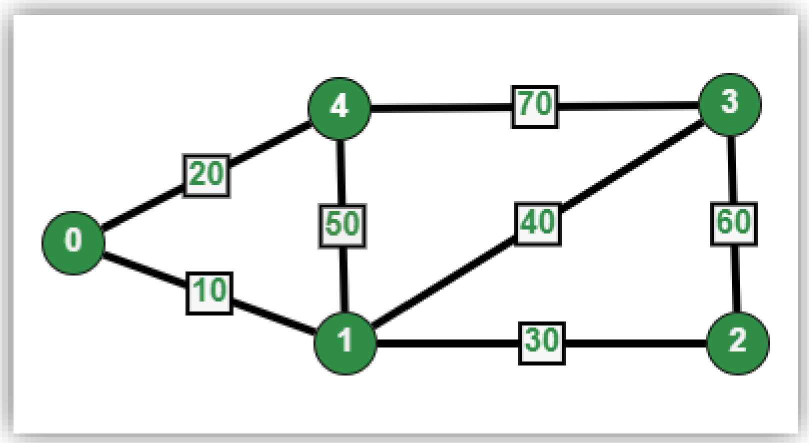
Output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\ASUS\Desktop\crack> cd "c:\Users\ASUS\Desktop\crack\DSA LAB\" ; if ($?) { gcc a
djacencymatrix.c -o adjacencymatrix } ; if ($?) { .\adjacencymatrix }
0 1 0 0 1
1 0 1 0 0
0 1 0 0 0
0 0 0 0 1
1 0 0 1 0
PS C:\Users\ASUS\Desktop\crack\DSA LAB> 
```

WEIGHTED GRAPH:

A weighted graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labelled graph in which the labels are numbers (which are usually taken to be positive)



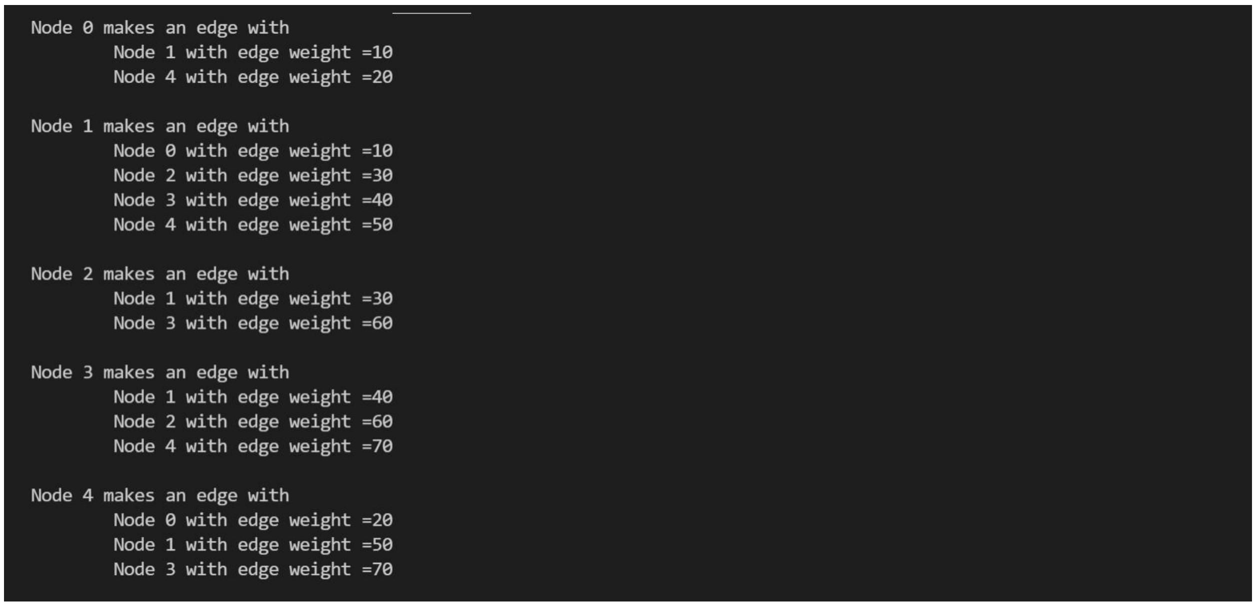
Weighted graphs are used for applications where we need to take into account some cost or measurement between vertices of the graph. For example, the weights can represent the time it costs to travel from one location to another. Or, they can represent a measurement, such as the distance between the locations.

Program to represent Unidirected Weighted Graph:

```
/*************************************************************************
MC201
ANEESH - 2K20MC21
AYUSHI - 2K20MC35

C++ program to represent undirected and weighted graph using STL. The
program basically prints adjacency list representation of graph.
*************************************************************************/

#include <bits/stdc++.h>
using namespace std;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}
```

```
// Print adjacency list representation ot graph
void printGraph(vector<pair<int,int> > adj[], int V)
{
    int v, w;
    for (int u = 0; u < V; u++)
    {
        cout << "Node " << u << " makes an edge with \n";
        for (auto it = adj[u].begin(); it!=adj[u].end(); it++)
        {
            v = it->first;
            w = it->second;
            cout << "\tNode " << v << " with edge weight ="
                << w << "\n";
        }
        cout << "\n";
    }
}

// Driver code
int main()
{
    int V = 5;
    vector<pair<int, int> > adj[V];
    addEdge(adj, 0, 1, 10);
    addEdge(adj, 0, 4, 20);
    addEdge(adj, 1, 2, 30);
    addEdge(adj, 1, 3, 40);
    addEdge(adj, 1, 4, 50);
    addEdge(adj, 2, 3, 60);
    addEdge(adj, 3, 4, 70);
    printGraph(adj, V);
    return 0;
}
```

Output:



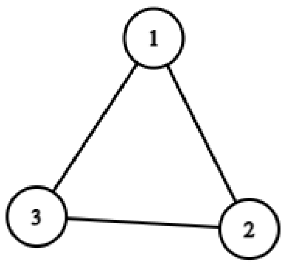
```
Node 0 makes an edge with
        Node 1 with edge weight =10
        Node 4 with edge weight =20

Node 1 makes an edge with
        Node 0 with edge weight =10
        Node 2 with edge weight =30
        Node 3 with edge weight =40
        Node 4 with edge weight =50

Node 2 makes an edge with
        Node 1 with edge weight =30
        Node 3 with edge weight =60

Node 3 makes an edge with
        Node 1 with edge weight =40
        Node 2 with edge weight =60
        Node 4 with edge weight =70

Node 4 makes an edge with
        Node 0 with edge weight =20
        Node 1 with edge weight =50
        Node 3 with edge weight =70
```

SPECTRUM OF GRAPH:

The adjacency matrix of an undirected simple graph is symmetric, and therefore has a complete set of real eigenvalues and an orthogonal eigenvector basis.
The set of eigenvalues of a graph is the Spectrum of the graph.
It is common to denote the eigenvalues by,
$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$

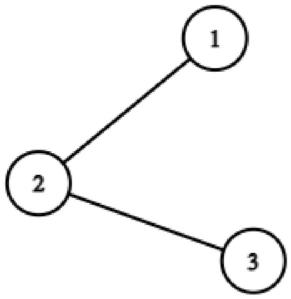
Property I:

The greatest eigenvalue λ_1 is bounded above by the maximum degree.



$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Maximum degree of graph is 2.
$\lambda_1, \lambda_2, \lambda_3 = 2, -1, -1$
$\lambda_1 = 2 \leq 2$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

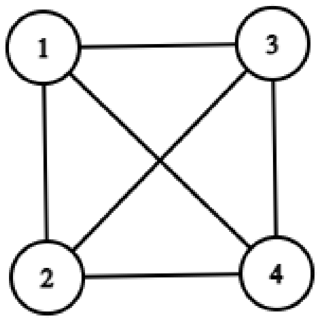
Maximum degree of graph is 2.

$\lambda_1, \lambda_2, \lambda_3 = \sqrt{2}, -\sqrt{2}, 0$

$\lambda_1 = \sqrt{2} \leq 2$

Property II:

For d-regular graphs, d is the first eigenvalue of A, λ_i. The multiplicity of this eigenvalue is the number of connected components of G.



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

d3 regular graph:

It is regular graph because all nodes have degree 3.

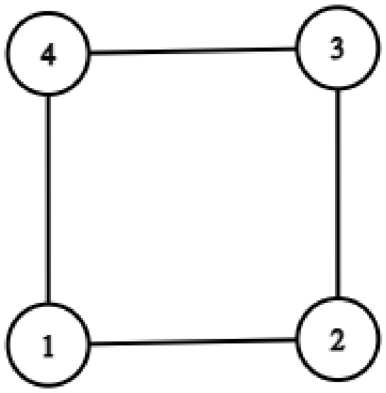
Eigen Values:

$\lambda_1, \lambda_2, \lambda_3, \lambda_4 = 3, -1, -1, -1$

$\lambda_1 = 3$

Multiplicity, m = 1

As it is clearly seen that there is one connected component i.e. whole graph.



$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

d2 regular graph:

It is regular graph because all nodes have degree 2.

Eigen Values:

$\lambda_1, \lambda_2, \lambda_3, \lambda_4 = 2, -2, 0, 0$

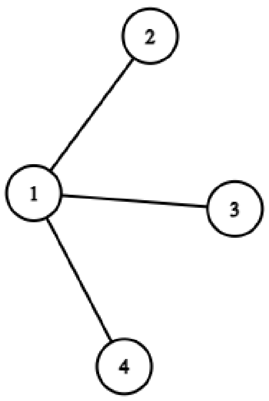
$\lambda_1 = 2$

Multiplicity, m = 1

As it is clearly seen that there is one connected component i.e. whole graph.

Property III:

It can be shown that for each eigenvalue λ_i , its opposite $\lambda_i = -\lambda_{n+1-i}$ is also an eigenvalue of A if G is a bipartite graph.



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Bipartite graph:

It is a bipartite graph because every edge connects the node of X = {1} with the node of Y = {2,3,4}

Eigen Values:

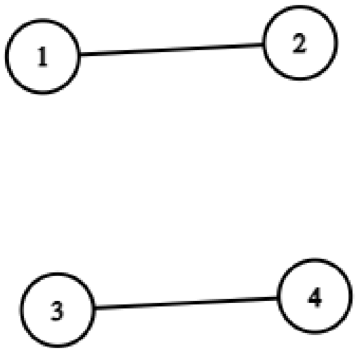
$\lambda_1, \lambda_2, \lambda_3, \lambda_4 = \sqrt{3}, 0, 0, -\sqrt{3}$

$\lambda_1 = \sqrt{3} = -(-\sqrt{3}) = -\lambda_4 = -\lambda_{4-1+1}$

$\lambda_2 = 0 = -(0) = -\lambda_3 = -\lambda_{4-2+1}$

Hence, $\lambda_i = -\lambda_{n+1-i}$

Regular Bipartite Graph:



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

d1 regular graph:

It is regular graph because all nodes have degree 1.

Bipartite graph:

It is a bipartite graph because every edge connects the node of X = {1,3} with the node of Y = {2,4}

Eigen Values:

$\lambda_1, \lambda_2, \lambda_3, \lambda_4 = 1, 1, -1, -1$

$\lambda_1 = 1$

Multiplicity, m = 2
As it is clearly seen that there are 2 connected components i.e. (1,2) and (3,4).

$\lambda_1 = 1 = -(-1) = \; -\lambda_4 = -\lambda_{4-1+1}$
$\lambda_2 = 1 = -(-1) = \; -\lambda_3 = -\lambda_{4-2+1}$

Hence, $\lambda_i = -\lambda_{n+1-i}$

Matrix powers:

If A is the adjacency matrix of the directed or undirected graph G, then the matrix A^n (i.e., the matrix product of n copies of A) has an interesting interpretation:

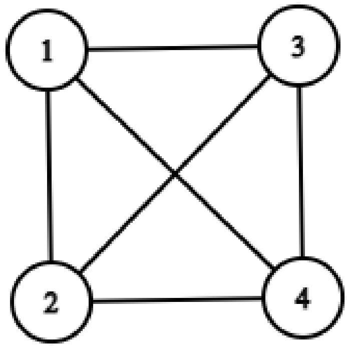
The element (i, j) gives the number of (directed or undirected) walks of length n from vertex i to vertex j.

For degree 3 i.e. for n = 3

A*A*A = $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$ **=** $\begin{bmatrix} 6 & 7 & 7 & 7 \\ 7 & 6 & 7 & 7 \\ 7 & 7 & 6 & 7 \\ 7 & 7 & 7 & 6 \end{bmatrix}$

A*A = $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$ **=** $\begin{bmatrix} 3 & 2 & 2 & 2 \\ 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 2 \\ 2 & 2 & 2 & 3 \end{bmatrix}$

Matrix Powers are used to find out the **Geometry of the graph** without making it.



Applications of Matrix Powers:

❖ **Find out number of lines in a graph:**

To find out number of lines in a graph first we have to find out number of walk length for which i != j in matrix of order 2:

Walks, w = (2+2+2+2+2+2)*2 = 24

Number of lines, n =$\frac{w}{2*2} = \frac{24}{4} = 6$ lines

As we can see that total number of lines are 6 only,
Lines = {(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)}

❖ **Find out number of triangles in a graph:**

To find out number of triangles in a graph first we have to find out number of walk length for which i = j in matrix of order 3:

Walks, w = (6+6+6+6) = 24

Number of lines, n =$\frac{w}{3*2} = \frac{24}{6} = 4$ triangles

As we can see that total number of triangles are 4 only,
Triangles = {(1,2,4), (1,3,4), (2,4,3), (1,2,3)}

Dijkstra's Algorithm:

1. Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.

2. The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
3. Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
4. The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

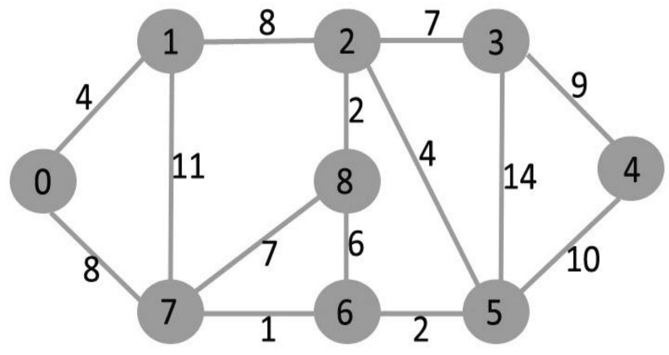
Program for shortest path using Dijkstra's Algorithm:

```
/*************************************************************************
MC201
ANEESH - 2K20MC21
AYUSHI - 2K20MC35

A C++ program for Dijkstra's single source shortest path algorithm.
*************************************************************************/
#include <iostream>
using namespace std;
#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;
}
```

```
// A utility function to print the constructed distance array
void printSolution(int dist[])
{
    cout <<"Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t"<<dist[i]<< endl;
}

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

    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;

    // 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 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])
                dist[v] = dist[u] + graph[u][v];
    }
    // print the constructed distance array
    printSolution(dist);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph */
    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, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 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:

```
Vertex     Distance from Source
0               0
1               4
2               12
3               19
4               21
5               11
6               9
7               8
8               14
```



Real life application of Graphs:

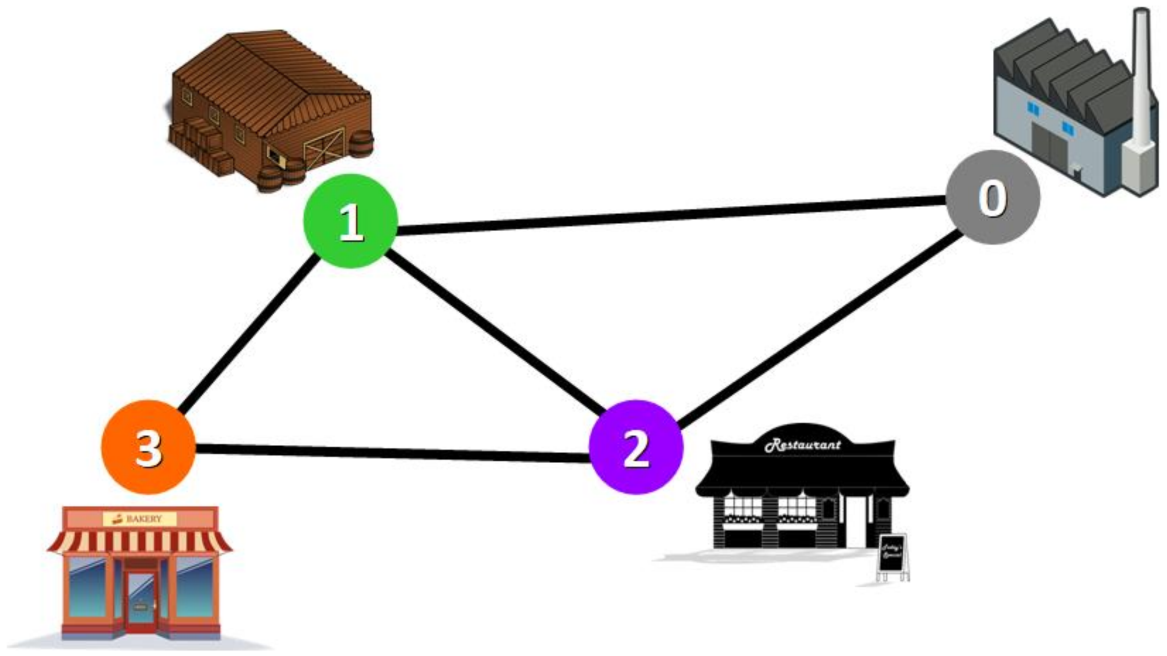
Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks.

For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale.

The Internet, for example, is a vast, virtual graph. Every vertex is an individual webpage, and every edge means that there is a hyperlink between two pages.

Some websites, like Wikipedia or Facebook, have lots of incoming links, while many smaller websites may have very few incoming links. This is the underlying concept which Google uses to sort search results.

- Social networking Facebook
- Google maps platform
- Flight networks
- GPS navigations systems



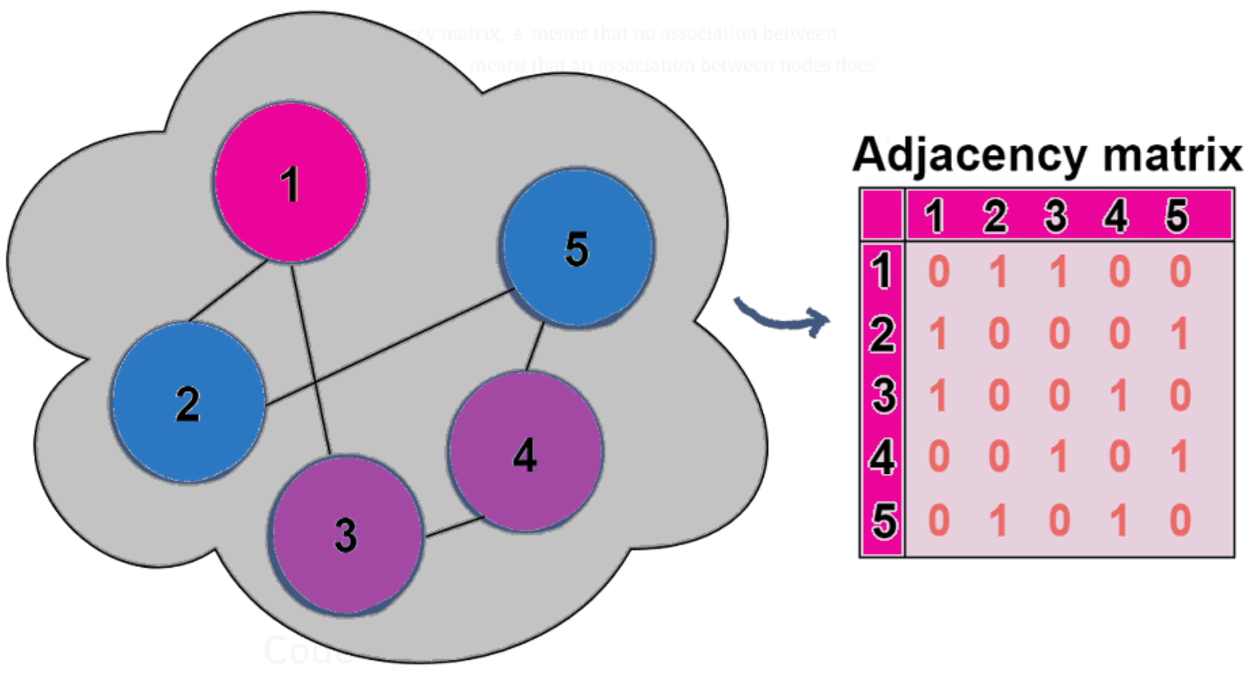
Applicatons of Adjacency matrix:

With the Dijkstra algorithm, one can find the shortest path from A to B using the adjacency matrix. This is the most common usage I think.

This is useful to:

1. Create routing tables in networks (how is a packet going to travel from your router to some other router, server, or whatever endpoint
2. Navigation systems
3. Public transportation connection search
4. Anything other that requires a shortest path search

However, there are surely other applications which might have nothing to do with the Dijkstra algorithm.
We think they are also used for e.g. efficient representation of sparse matrices in scientific computing.


cy matrix, a means that no association between
means that an association between nodes does
1
5
2
4
3
Adjacency matrix

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	0	1
3	1	0	0	1	0
4	0	0	1	0	1
5	0	1	0	1	0

Conclusion:

Through the project we tried to understand what is graph and its different components. The vertex, edges, adjacency and path.
Also the directed graph also known as diagraph. Through matrix representation we tried to understand that is easy to us through pictorial representation to understand graphs but we have to identify some methods for machines to make it understand . In the project we discuss one major method which is widely used i.e. through adjacency matrix.
Also we provide a code for better understanding.

References:

- Wikipedia.org
- Geeksforgeeks.org
- Tutorialspoint.com
- CSAcademy.com