**Introduction to Graph Data Structure**

**Graph Data Structure**is a [non-linear data structure](https://www.geeksforgeeks.org/introduction-to-hierarchical-data-structure) consisting of vertices and edges. It is useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structure can be used to analyze and understand the dynamics of team performance and player interactions on the field.

**Table of Content**

* [What is Graph Data Structure?](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#what-is-a-graph)
* [Components of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#components-of-a-graph)
* [Types Of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#types-of-graph)
* [Representation of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#representation-of-graphs)
  + [Adjacency Matrix Representation of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#adjacency-matrix-representation-of-graph-data-structure)
  + [Adjacency List Representation of Graph](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#adjacency-list-representation-of-graph)
* [Basic Operations on Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#basic-operations-on-graphs)
* [Difference between Tree and Graph](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#difference-between-tree-and-graph)
* [Real-Life Applications of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#reallife-applications-of-graph)
* [Advantages of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#advantages-of-graphs)
* [Disadvantages of Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#disadvantages-of-graphs)
* [Frequently Asked Questions(FAQs) on Graph Data Structure](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/#frequently-asked-questionsfaqs-on-graphs)

**What is Graph Data Structure?**

**Graph** is a [non-linear data structure](https://www.geeksforgeeks.org/introduction-to-hierarchical-data-structure) consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( **V**) and a set of edges( **E**). The graph is denoted by **G(V, E).**

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it’s the key to unlocking insights into team performance and player dynamics in sports.

**Components of Graph Data Structure**

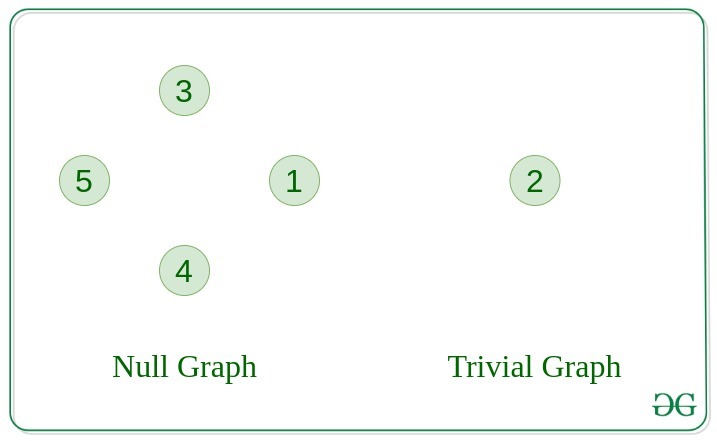
* **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
* **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

**Types Of Graphs in Data Structure and Algorithms**

**1. Null Graph**

A graph is known as a null graph if there are no edges in the graph.

**2. Trivial Graph**

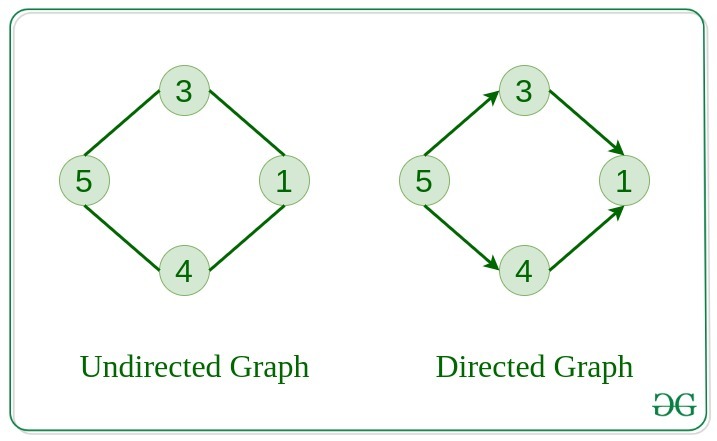
Graph having only a single vertex, it is also the smallest graph possible. 

**3. Undirected Graph**

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

**4. Directed Graph**

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

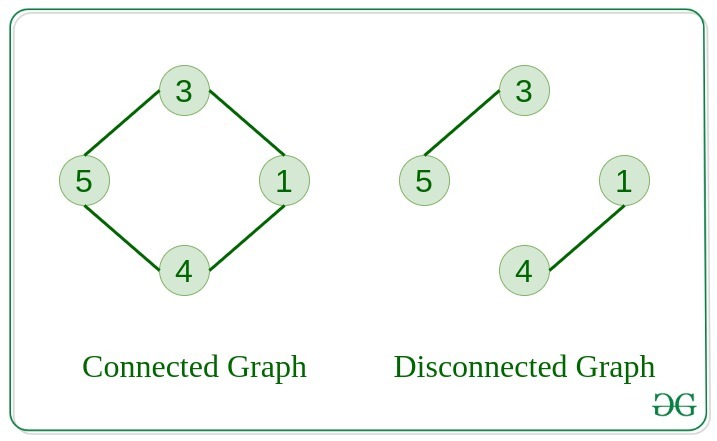


**5. Connected Graph**

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

**6. Disconnected Graph**

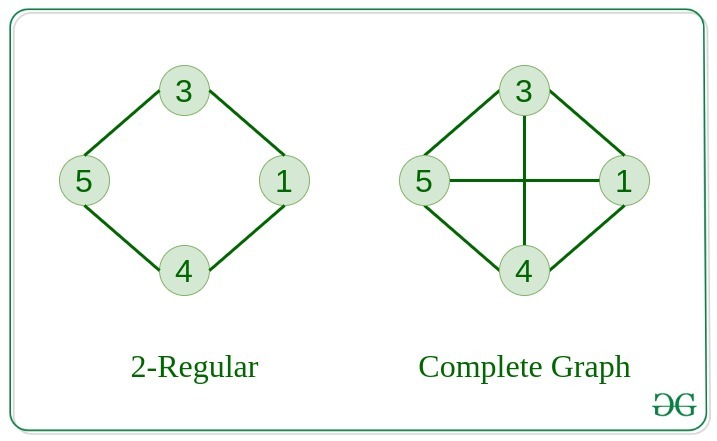
The graph in which at least one node is not reachable from a node is known as a disconnected graph.



**7. Regular Graph**

The graph in which the degree of every vertex is equal to K is called K regular graph.

**8. Complete Graph**

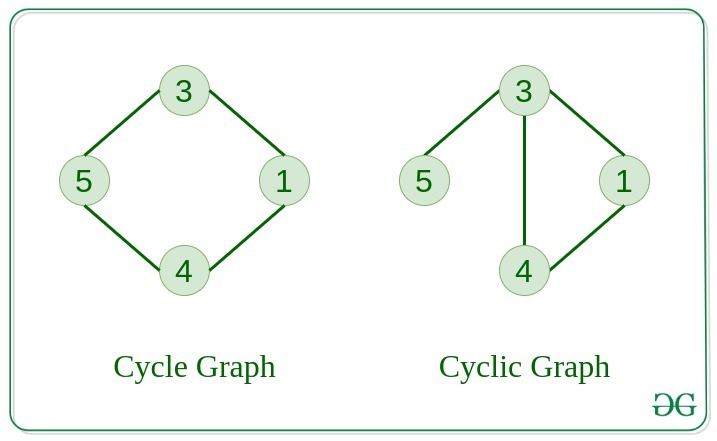
The graph in which from each node there is an edge to each other node.

**9. Cycle Graph**

The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.

**10. Cyclic Graph**

A graph containing at least one cycle is known as a Cyclic graph.

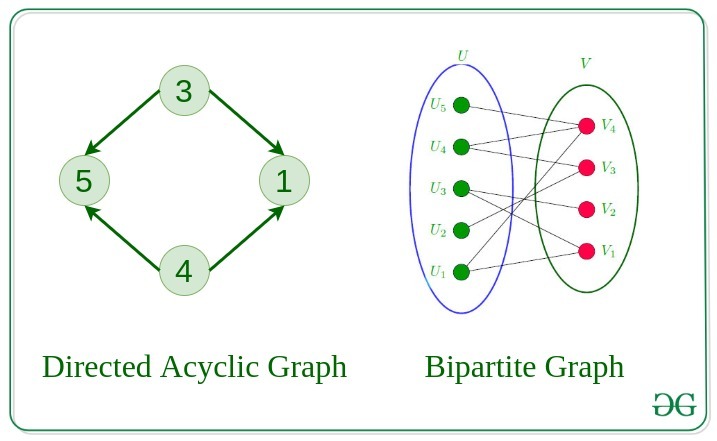


**11. Directed Acyclic Graph**

A Directed Graph that does not contain any cycle.

**12. Bipartite Graph**

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



**13. Weighted Graph**

* A graph in which the edges are already specified with suitable weight is known as a weighted graph.
* Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

**Representation of Graph Data Structure:**

There are multiple ways to store a graph: The following are the most common representations.

* Adjacency Matrix
* Adjacency List

[Adjacency Matrix Representation of Graph Data Structure:](https://www.geeksforgeeks.org/adjacency-matrix)

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

Below is the implementation of Graph Data Structure represented using Adjacency Matrix:

C++CJavaPythonC#JavaScript



1

def add\_edge(mat, i, j):

2

3

# Add an edge between two vertices

4

mat[i][j] = 1 # Graph is

5

mat[j][i] = 1 # Undirected

6

​

7

def display\_matrix(mat):

8

9

# Display the adjacency matrix

10

for row in mat:

11

print(" ".join(map(str, row)))

12

​

13

# Main function to run the program

14

if \_\_name\_\_ == "\_\_main\_\_":

15

V = 4 # Number of vertices

16

mat = [[0] \* V for \_ in range(V)]

17

​

18

# Add edges to the graph

19

add\_edge(mat, 0, 1)

20

add\_edge(mat, 0, 2)

21

add\_edge(mat, 1, 2)

22

add\_edge(mat, 2, 3)

23

​

24

# Optionally, initialize matrix directly

25

"""

26

mat = [

27

[0, 1, 0, 0],

28

[1, 0, 1, 0],

29

[0, 1, 0, 1],

30

[0, 0, 1, 0]

31

]

32

"""

33

​

34

# Display adjacency matrix

35

print("Adjacency Matrix:")

36

display\_matrix(mat)

**Output**

Adjacency Matrix Representation

0 1 1 0

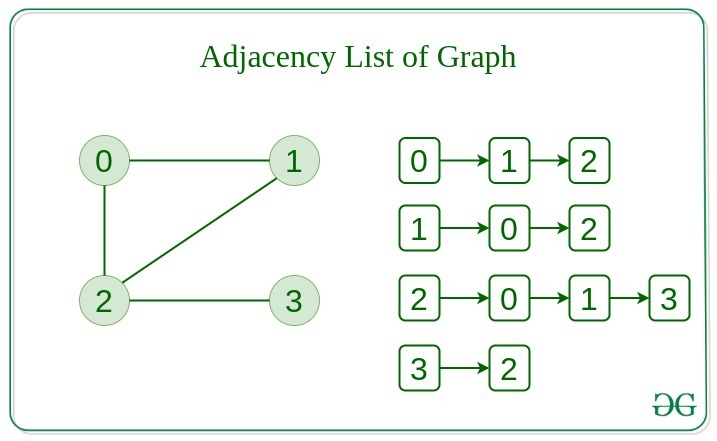
1 0 1 0

1 1 0 1

0 0 1 0

[Adjacency List Representation of Graph:](https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa)

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Below is the implementation of Graph Data Structure represented using Adjacency List:

C++CJavaPythonC#JavaScript



1

def add\_edge(adj, i, j):

2

adj[i].append(j)

3

adj[j].append(i) # Undirected

4

​

5

def display\_adj\_list(adj):

6

for i in range(len(adj)):

7

print(f"{i}: ", end="")

8

for j in adj[i]:

9

print(j, end=" ")

10

print()

11

​

12

# Create a graph with 4 vertices and no edges

13

V = 4

14

adj = [[] for \_ in range(V)]

15

​

16

# Now add edges one by one

17

add\_edge(adj, 0, 1)

18

add\_edge(adj, 0, 2)

19

add\_edge(adj, 1, 2)

20

add\_edge(adj, 2, 3)

21

​

22

print("Adjacency List Representation:")

23

display\_adj\_list(adj)

**Output**

Adjacency List Representation:

0: 1 2

1: 0 2

2: 0 1 3

3: 2

**Comparison between Adjacency Matrix and Adjacency List**

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as [Prim’s](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5) and [Dijkstra](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7) adjacency matrix is used to have less complexity.

| **Action** | **Adjacency Matrix** | **Adjacency List** |
| --- | --- | --- |
| Adding Edge | O(1) | O(1) |
| Removing an edge | O(1) | O(N) |
| Initializing | O(N\*N) | O(N) |

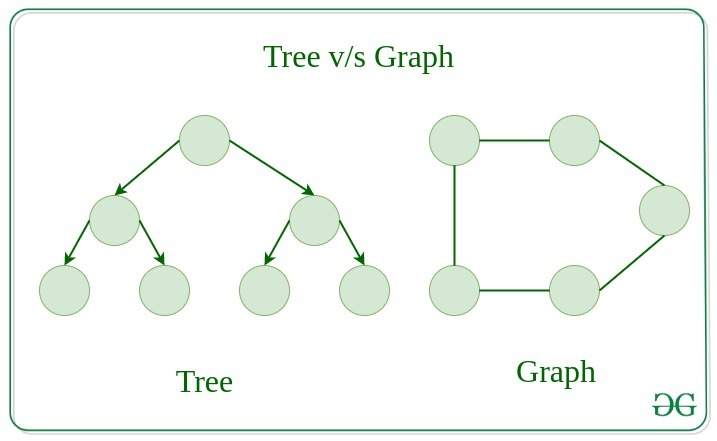
**Basic Operations on Graph Data Structure:**

Below are the basic operations on the graph:

* Insertion or Deletion of Nodes in the graph
  + [Add and Remove vertex in Adjacency List representation of Graph](https://www.geeksforgeeks.org/add-and-remove-vertex-in-adjacency-list-representation-of-graph)
  + [Add and Remove vertex in Adjacency Matrix representation of Graph](https://www.geeksforgeeks.org/add-and-remove-vertex-in-adjacency-matrix-representation-of-graph)
* Insertion or Deletion of Edges in the graph
  + [Add and Remove Edge in Adjacency List representation of a Graph](https://www.geeksforgeeks.org/add-and-remove-edge-in-adjacency-list-representation-of-a-graph)
  + [Add and Remove Edge in Adjacency Matrix representation of a Graph](https://www.geeksforgeeks.org/add-and-remove-edge-in-adjacency-matrix-representation-of-a-graph)
* Searching in Graph Data Structure- Search an entity in the graph.
* Traversal of Graph Data Structure- Traversing all the nodes in the graph.

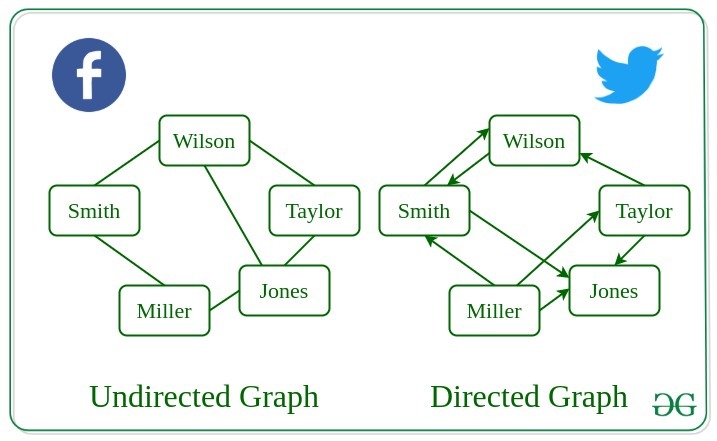
**Difference between Tree and Graph:**

Tree is a restricted type of Graph Data Structure, just with some more rules. Every tree will always be a graph but not all graphs will be trees. [Linked List](https://www.geeksforgeeks.org/data-structures/linked-list), [Trees](https://www.geeksforgeeks.org/binary-tree-data-structure), and [Heaps](https://www.geeksforgeeks.org/heap-data-structure) all are special cases of graphs.



**Real-Life Applications of Graph Data Structure:**

Graph Data Structure has numerous real-life applications across various fields. Some of them are listed below:



* If we recall all the previous data structures that we have studied like array, linked list, tree, etc. All these had some restrictions on structure (mostly linear and tree hierarchical which means no loops). Graph allows random connections between nodes which is useful in many real world problems where do have restrictions of previous data structures.
* Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can seen as graph problems.
* Used to represent the topology of computer networks, such as the connections between routers and switches.
* Used to represent the connections between different places in a transportation network, such as roads and airports.
* **Neural Networks:**Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
* **Compilers:**Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
* **Robot planning:**Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.
* Dependencies in a software project (or any other type of project) can be seen as graph and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.
* For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

**Advantages of Graph Data Structure:**

* Graph Data Structure used to represent a wide range of relationships as we do not have any restrictions like previous data structures (Tree cannot have loops and have to be hierarchical. Arrays, Linked List, etc are linear)
* They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
* Any real world problem where we certain set of items and relations between them can be easily modeled as a graph and a lot of standard graph algorithms like BFS, DFS, Spanning Tree, Shortest Path, Topological Sorting and Strongly Connected
* Graph Data Structure can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

**Disadvantages of Graph Data Structure:**

* Graph Data Structure can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.
* Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
* Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
* Graph Data Structure can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

**Adjacency Matrix Representation**

**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. An adjacency matrix is a simple and straightforward way to represent graphs and is particularly useful for dense graphs.

**Table of Content**

* [1. Adjacency Matrix for Undirected and Unweighted graph:](https://www.geeksforgeeks.org/adjacency-matrix/#1-adjacency-matrix-for-undirected-and-unweighted-graph)
* [2. Adjacency Matrix for Undirected and Weighted graph:](https://www.geeksforgeeks.org/adjacency-matrix/#2-adjacency-matrix-for-undirected-and-weighted-graph)
* [3. Adjacency Matrix for Directed and Unweighted graph:](https://www.geeksforgeeks.org/adjacency-matrix/#3-adjacency-matrix-for-directed-and-unweighted-graph)
* [4. Adjacency Matrix for Directed and Weighted graph:](https://www.geeksforgeeks.org/adjacency-matrix/#4-adjacency-matrix-for-directed-and-weighted-graph)
* [Properties of Adjacency Matrix](https://www.geeksforgeeks.org/adjacency-matrix/#properties-of-adjacency-matrix)
* [Applications of Adjacency Matrix:](https://www.geeksforgeeks.org/adjacency-matrix/#applications-of-adjacency-matrix)
* [Advantages of Adjacency Matrix:](https://www.geeksforgeeks.org/adjacency-matrix/#advantages-of-adjacency-matrix)
* [Disadvantages of Adjacency Matrix:](https://www.geeksforgeeks.org/adjacency-matrix/#disadvantages-of-adjacency-matrix)

**What is Adjacency Matrix?**

Adjacency Matrix is a square matrix used to represent a finite graph by storing the relationships between the nodes in their respective cells. For a graph with **V** vertices, the adjacency matrix A is an V**X V**matrix or 2D array.

**1. Adjacency Matrix for Undirected and Unweighted graph:**

Consider an Undirected and Unweighted graph **G**with **4 vertices** and **3 edges**. For the graph G, the adjacency matrix would look like:

Here's how to interpret the matrix:

* **A[i][i] ​= 1,** there is an edge between vertex i and vertex j.
* **A[i][i] ​= 0,** there is NO edge between vertex i and vertex j.

Below is a program to create an adjacency matrix for an unweighted and undirected graph:

C++CJavaPythonC#JavaScript



1

def add\_edge(mat, i, j):

2

3

# Add an edge between two vertices

4

mat[i][j] = 1 # Graph is

5

mat[j][i] = 1 # Undirected

6

​

7

def display\_matrix(mat):

8

9

# Display the adjacency matrix

10

for row in mat:

11

print(" ".join(map(str, row)))

12

​

13

# Main function to run the program

14

if \_\_name\_\_ == "\_\_main\_\_":

15

V = 4 # Number of vertices

16

mat = [[0] \* V for \_ in range(V)]

17

​

18

# Add edges to the graph

19

add\_edge(mat, 0, 1)

20

add\_edge(mat, 0, 2)

21

add\_edge(mat, 1, 2)

22

add\_edge(mat, 2, 3)

23

​

24

# Optionally, initialize matrix directly

25

"""

26

mat = [

27

[0, 1, 0, 0],

28

[1, 0, 1, 0],

29

[0, 1, 0, 1],

30

[0, 0, 1, 0]

31

]

32

"""

33

​

34

# Display adjacency matrix

35

print("Adjacency Matrix:")

36

display\_matrix(mat)

**Output**

Adjacency Matrix Representation

0 1 1 0

1 0 1 0

1 1 0 1

0 0 1 0

**2. Adjacency Matrix for Undirected and Weighted graph:**

Consider an Undirected and Weighted graph **G**with **5 vertices** and **5 edges**. For the graph G, the adjacency matrix would look like:

Here's how to interpret the matrix:

* **A[i][j] ​= INF**, then there is no edge between vertex i and j
* **A[i][j] ​= w,** then there is an edge between vertex i and j having weight = w.

**3. Adjacency Matrix for Directed and Unweighted graph:**

Consider an Directed and Unweighted graph **G**with 4 vertices and 4 edges. For the graph G, the adjacency matrix would look like:

Here's how to interpret the matrix:

* **A[i][j] ​= 1,** there is an edge from vertex i to vertex j
* **A[i][j] ​= 0,** No edge from vertex i to j.

**4. Adjacency Matrix for Directed and Weighted graph:**

Consider an Directed and Weighted graph **G**with 5 vertices and 6 edges. For the graph G, the adjacency matrix would look like:

Here's how to interpret the matrix:

* **A[i][j] ​= INF**, then there is no edge from vertex i to j
* **A[i][j] ​= w,** then there is an edge from vertex i having weight w

**Properties of Adjacency Matrix**

* **Diagonal Entries**: The diagonal entries A[i][i] are usually set to 0 (in case of unweighted) and INF in case of weighted, assuming the graph has no self-loops.
* **Undirected Graphs**: For undirected graphs, the adjacency matrix is symmetric. This means A[i][j] ​= A[j][i]​ for all i and j.

**Applications of Adjacency Matrix:**

* **Graph Representation**: The adjacency matrix is one of the most common ways to represent a graph computationally.
* **Connectivity**: By examining the entries of the adjacency matrix, one can determine whether the graph is connected or not. If the graph is undirected, it is connected if and only if the corresponding adjacency matrix is irreducible (i.e., there is a path between every pair of vertices). In directed graphs, connectivity can be analyzed using concepts like strongly connected components.
* **Degree of Vertices**: The degree of a vertex in a graph is the number of edges incident to it. In an undirected graph, the degree of a vertex can be calculated by summing the entries in the corresponding row (or column) of the adjacency matrix. In a directed graph, the in-degree and out-degree of a vertex can be similarly determined.

**Advantages of Adjacency Matrix:**

* **Simple:**Simple and Easy to implement.
* **Space Efficient for Dense Graphs:**Space efficient when the graph is dense as it requires V \* V space to represent the entire graph.
* **Faster access to Edges:**Adjacency Matrix allows constant look up to check whether there exists an edge between a pair of vertices.

**Disadvantages of Adjacency Matrix:**

* **Space inefficient for Sparse Graphs:**Takes up O(V\* V) space even if the graph is sparse.
* **Costly Insertions and Deletions:**Adding or deleting a vertex can be costly as it requires resizing the matrix.
* **Slow Traversals:**Graph traversals like DFS, BFS takes O(V \* V) time to visit all the vertices whereas Adjacency List takes only O(V + E).