Graph data structure

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components:

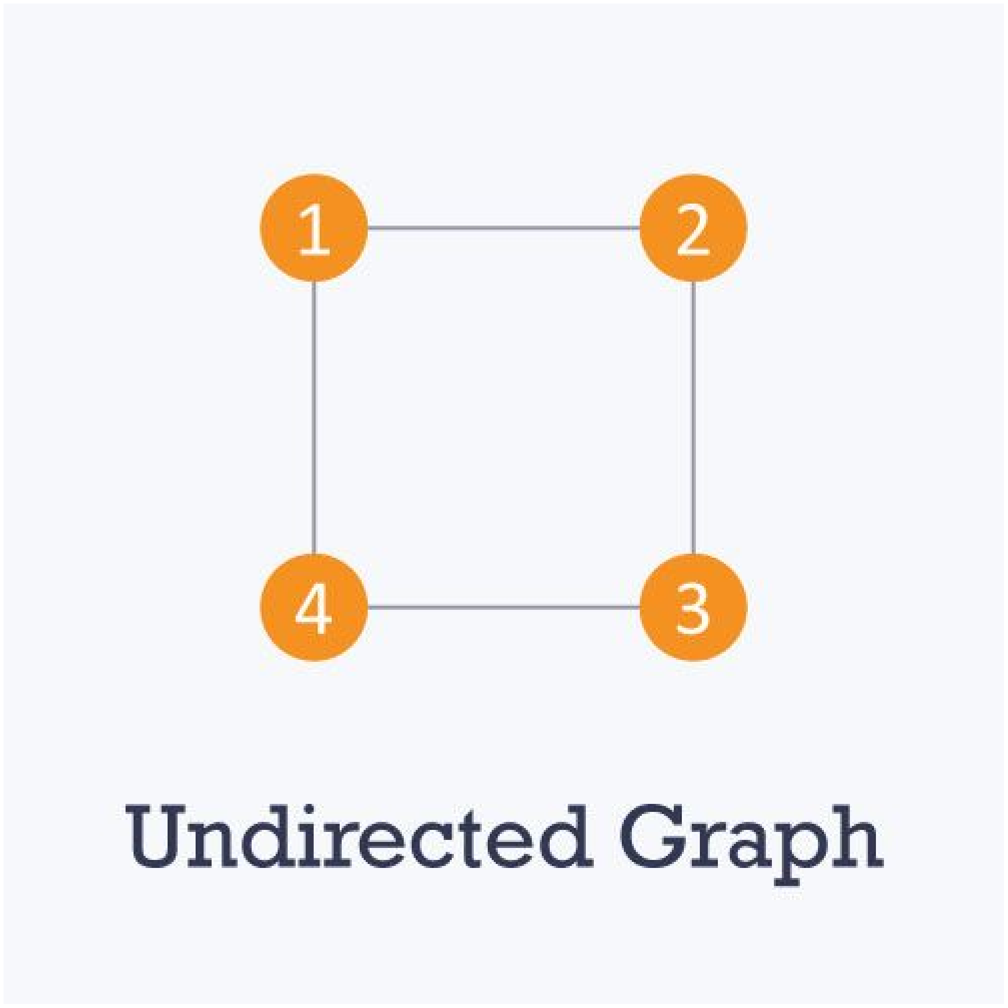
- **Nodes:** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges:** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Types of nodes:

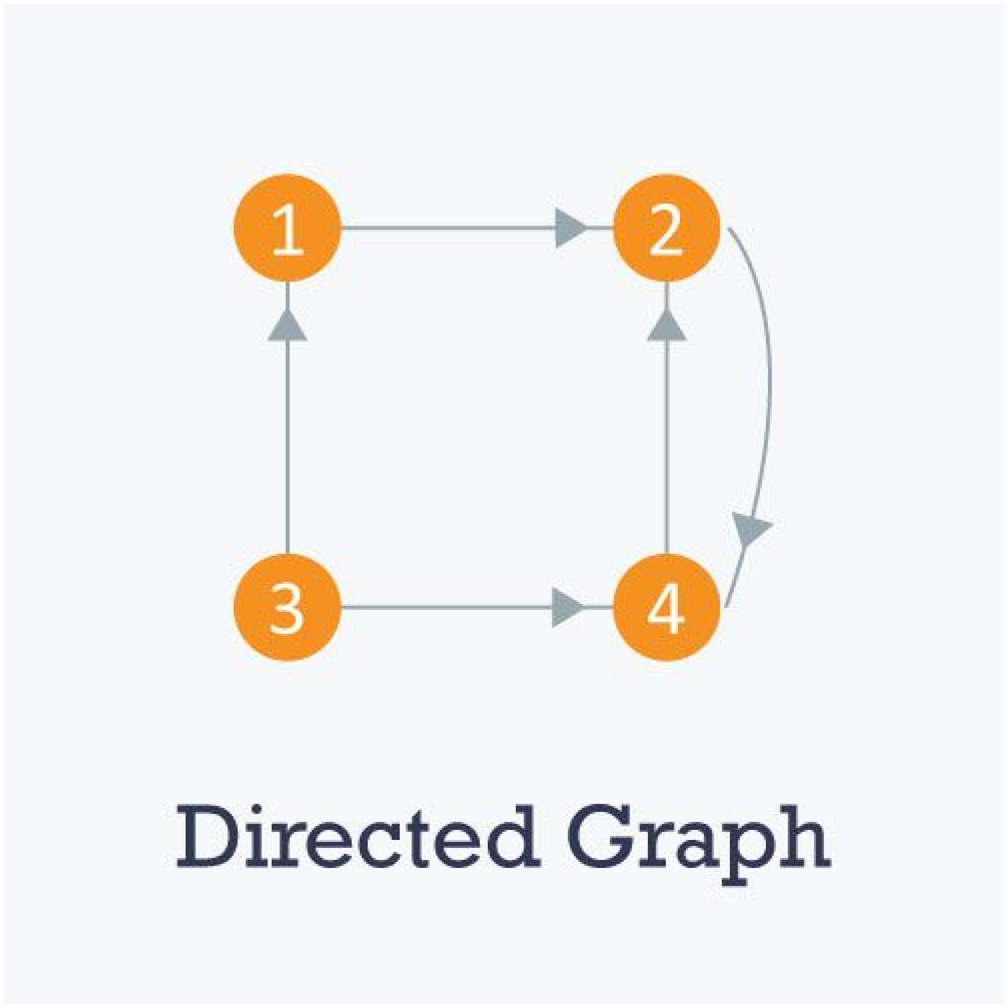
- Root node: The root node is the ancestor of all other nodes in a graph. It does not have any ancestor. Each graph consists of exactly one root node. Generally, you must start traversing a graph from the root node.
- Leaf nodes: In a graph, leaf nodes represent the nodes that do not have any successors. These nodes only have ancestor nodes. They can have any number of incoming edges but they will not have any outgoing edges.

Types of graphs:

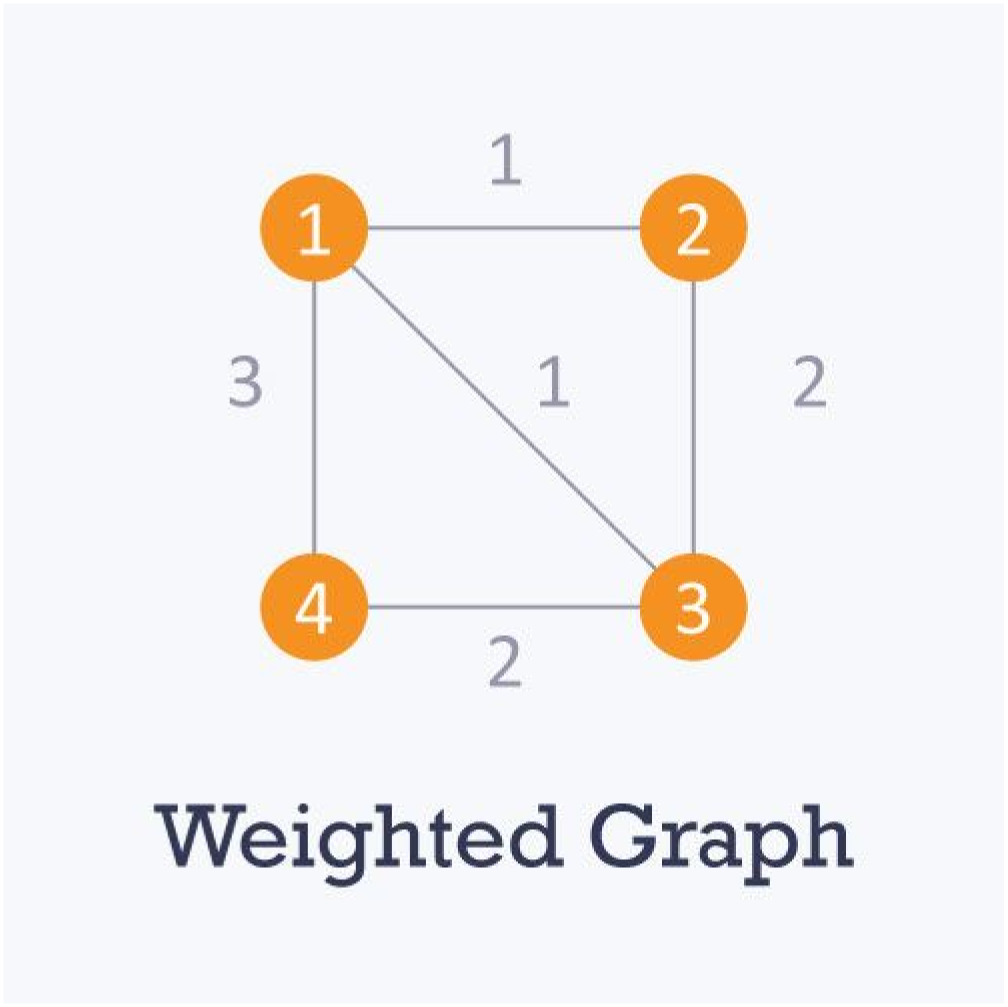
- Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



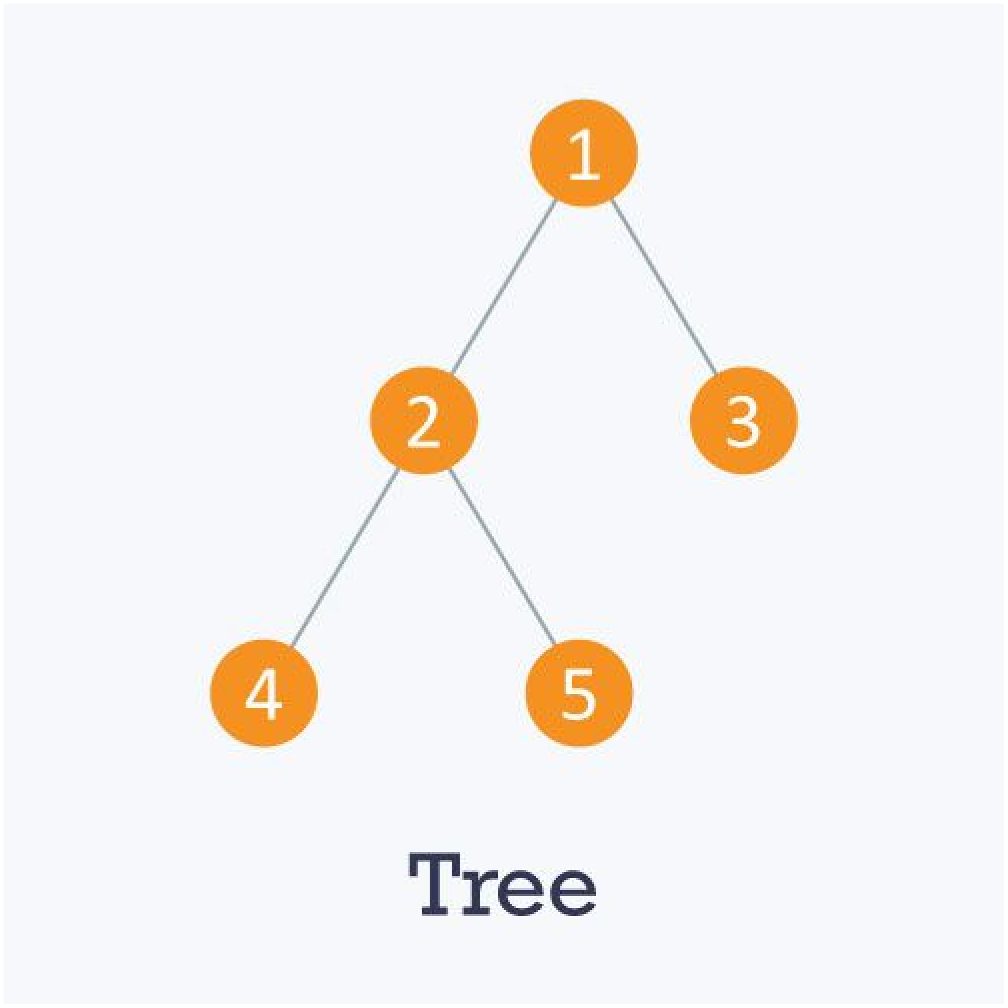
- **Directed:** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



- **Weighted:** **I**n a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
  - 1 -> 2 -> 3
  - 1 -> 3
  - 1 -> 4 -> 3
- Therefore the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



- **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.
- A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.
- ***Note*****:** A root node has no parent.
- A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



Graph representation:

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

Adjacency matrix:

An adjacency matrix is a VxV binary matrix A. Element Ai,j is 1 if there is an edge from vertex i to vertex j else Ai,jis 0.

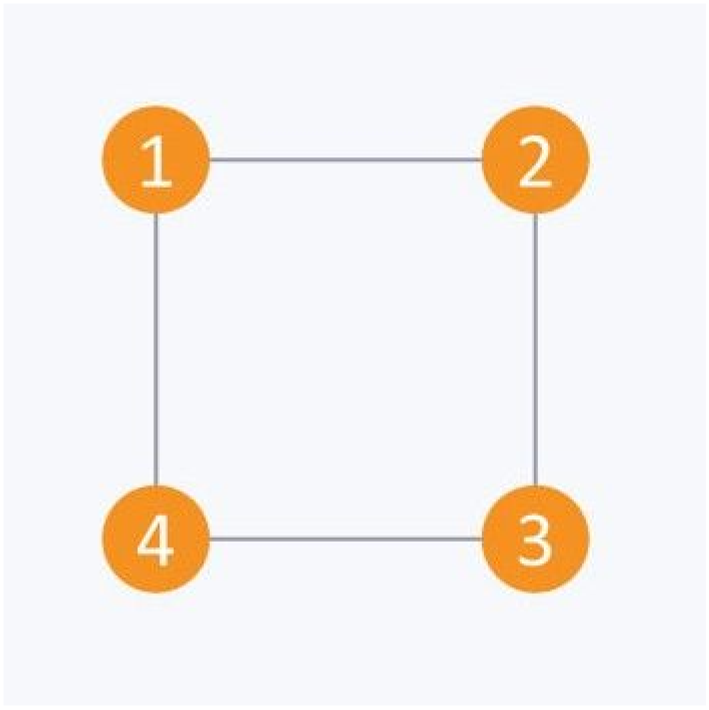
Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in Ai,j, the weight or cost of the edge will be stored.

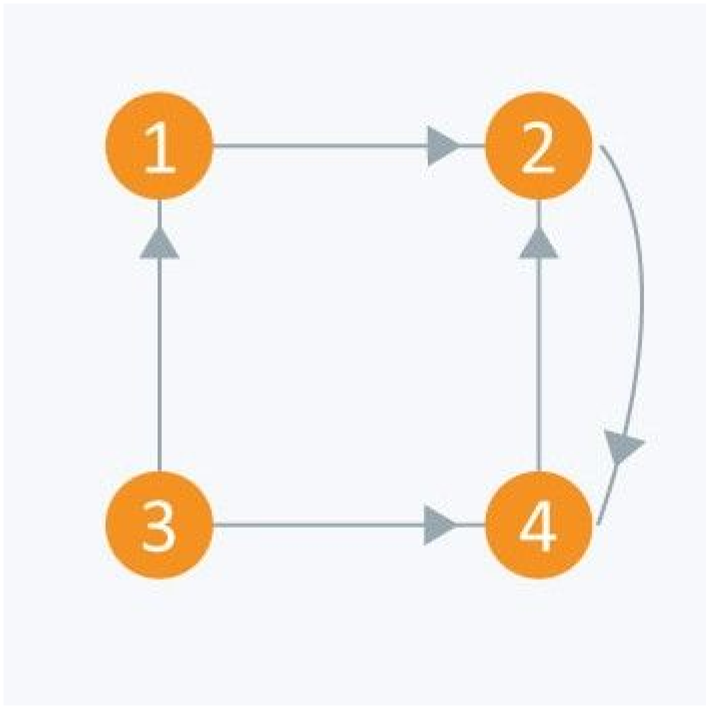
In an undirected graph, if Ai,j = 1, then Aj,i = 1. In a directed graph, if Ai,j = 1, then Aj,i may or may not be 1.

Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is O(V2).

The adjacency matrix of the following graph is:
i/j : 1 2 3 4
1 : **0 1 0 1**
2 **: 1 0 1 0**
3 **: 0 1 0 1**
4 **: 1 0 1 0**



The adjacency matrix of the following graph is:
i/j: 1 2 3 4
1 : 0 1 0 0
2 : 0 0 0 1
3 : 1 0 0 1
4 : 0 1 0 0



Consider the directed graph given above. Let's create this graph using an adjacency matrix and then show all the edges that exist in the graph.

<u>Input file</u>:

```
4       // nodes
5       //edges
1 2     //showing edge from node 1 to node 2
2 4     //showing edge from node 2 to node 4
3 1     //showing edge from node 3 to node 1
3 4     //showing edge from node 3 to node 4
4 2     //showing edge from node 4 to node 2
```

Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex − Adds a vertex to the graph.**
- **Add Edge − Adds an edge between the two vertices of the graph.**

- **Display Vertex − Displays a vertex of the graph.**

Adjacency matrix code in c:

```
#include<stdio.h>
#include<conio.h>
#define max 20
int adj[max][max];
int n;
main() {
  int choice;
  int node, origin, destin;
  create_graph();
  while (1) {
     printf("1.Insert a node\n");
     printf("2.Delete a node\n");
     printf("3.Dispaly\n");
     printf("4.Exit\n");
     printf("Enter your choice : ");
     scanf("%d", &choice);
     switch (choice) {
     case 1:
        insert_node();
        break;
     case 2:
        printf("Enter a node to be deleted : ");
        fflush(stdin);
        scanf("%d", &node);
        delete_node(node);
        break;
     case 3:
        display();
        break;
     case 4:
        exit(0);
     default:
        printf("Wrong choice\n");
        break;
        }
    }
    getch();
}

create_graph() {
    int i, max_edges, origin, destin;

    printf("Enter number of nodes : ");
    scanf("%d", &n);
    max_edges = n * (n - 1);

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

display() {
    int i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++)
            printf("%4d", adj[i][j]);
        printf("\n");
    }
}

insert_node() {
    int i;
    n++;
    printf("The inserted node is %d \n", n);
    for (i = 1; i <= n; i++) {
        adj[i][n] = 0;
        adj[n][i] = 0;
    }
}

void delete_node(char u) {
    int i, j;
    if (n == 0) {
        printf("Graph is empty\n");
        return;
    }
    if (u > n) {
        printf("This node is not present in the graph\n");
        return;
    }
    for (i = u; i <= n - 1; i++)
        for (j = 1; j <= n; j++) {
            adj[j][i] = adj[j][i + 1];
            adj[i][j] = adj[i + 1][j];
        }
    n--;
}
```

Adjacency matrix code in “java”:

/This is a java program to represent graph as a adjacency matrix

```
import java.util.Scanner;
```

```
public class Represent_Graph_Adjacency_Matrix
{
  private final int vertices;
  private int[][] adjacency_matrix;

  public Represent_Graph_Adjacency_Matrix(int v)
  {
     vertices = v;
     adjacency_matrix = new int[vertices + 1][vertices + 1];
  }

  public void makeEdge(int to, int from, int edge)
  {
     try
     {
        adjacency_matrix[to][from] = edge;
     }
     catch (ArrayIndexOutOfBoundsException index)
     {
        System.out.println("The vertices does not exists");
     }
  }

  public int getEdge(int to, int from)
  {
     try
     {
        return adjacency_matrix[to][from];
     }
     catch (ArrayIndexOutOfBoundsException index)
     {
        System.out.println("The vertices does not exists");
     }
     return -1;
  }

  public static void main(String args[])
  {
     int v, e, count = 1, to = 0, from = 0;
     Scanner sc = new Scanner(System.in);
```

```
Represent_Graph_Adjacency_Matrix graph;
try
{
    System.out.println("Enter the number of vertices: ");
    v = sc.nextInt();
    System.out.println("Enter the number of edges: ");
    e = sc.nextInt();

    graph = new Represent_Graph_Adjacency_Matrix(v);

    System.out.println("Enter the edges: <to> <from>");
    while (count <= e)
    {
        to = sc.nextInt();
        from = sc.nextInt();

        graph.makeEdge(to, from, 1);
        count++;
    }

    System.out.println("The adjacency matrix for the given graph is: ");
    System.out.print("  ");
    for (int i = 1; i <= v; i++)
        System.out.print(i + " ");
    System.out.println();

    for (int i = 1; i <= v; i++)
    {
        System.out.print(i + " ");
        for (int j = 1; j <= v; j++)
            System.out.print(graph.getEdge(i, j) + " ");
        System.out.println();
    }

}
catch (Exception E)
{
    System.out.println("Somthing went wrong");
}

sc.close();
```

```
  }
}
```