NextGen Academy

Towards fulfilling a million dreams

Data Structures Using C

Module IV: Graph

June 20, 2023

NextGen Academy

Towards fulfilling a million dreams

Contents

This unit covers

- Introduction to Graphs and Terminologies: Providing an initial understanding of graphs, their components, and key terminologies used in graph theory.
- Types of Graphs: Exploring different categories of graphs, including directed, undirected, weighted, cyclic, and acyclic graphs, each with specific characteristics.
- Graph Representations: Discussing various methods to represent graphs, focusing on the adjacency matrix representation as one approach.
- Spanning Trees: Introducing the concept of spanning trees within a graph, which covers every vertex exactly once with no cycles.
- Minimum Spanning Tree: Explaining the minimum spanning tree, a spanning tree with the smallest total edge weight.
- Weighted Graphs: Exploring graphs where edges have weights or costs associated with them, often used to model real-world scenarios.
- Shortest Path Algorithms: Discussing algorithms used to find the shortest path between two vertices in a graph, such as Dijkstra's algorithm or the Bellman-Ford algorithm.
- Graph Traversal - Breadth-First Depth-First: Covering breadth-first traversal (BFS) and depth-first traversal (DFS), two fundamental approaches to visit nodes in a graph.
- Connectivity of Graphs: Explaining the concepts of connected components and determining the connectivity of a graph, which measures how the vertices are connected.

1 Graphs

A graph is an abstract data type (ADT) that consists of a set of objects that are connected to each other via links. These objects are called vertices and the links are called edges.

Usually, a graph is represented as G = V, E, where G is the graph space, V is the set of vertices and E is the set of edges. If E is empty, the graph is known as a forest.

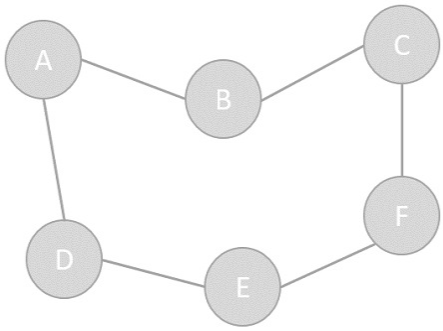
Before we proceed further, let's familiarize ourselves with some important terms

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.



1.1 Graph Terminologies

1.1.1 Graph Connectivity:

There are some important terminologies of graph given below.

- Graph connectivity explores the interconnectedness of nodes and edges in a graph.
- It is crucial for understanding network resilience, communication protocols, and social network structures.
- Key learning objectives include grasping the significance of connectivity and methods for identifying connected components in graphs.

1.1.2 Random Walks on Graphs:

- Random walks involve traversing the nodes of a graph unpredictably, guided by randomness.
- They are used in data analysis, search algorithms, and modeling complex systems.
- Learning objectives encompass understanding the concept of random walks, their applications, and their role in data-driven exploration.

1.1.3 Online Paging Algorithms:

- Online paging algorithms deal with real-time decision-making under uncertainty and limited information.
- They optimize resource allocation, with applications in web caching and memory management.
- Objectives include comprehending online algorithms, resource optimization, and real-world applications.

1.1.4 Adversary Models:

- Adversary models involve scenarios where adversaries aim to disrupt algorithms and decision-making processes.
- They are vital in computer security and game theory.

Learning objectives cover strategies for designing resilient algorithms and decision-making in adversarial settings

1.2 Graph Theory

Problem-solving in computer science refers to designing, analyzing, and implementing solutions to computational or algorithmic challenges. It involves

- Graph theory is a branch of discrete mathematics that studies the relationships between edges and vertices in geometric shapes, known as graphs.
- In mathematics, a graph is a collection of nodes or vertices and edges that connect pairs of nodes. It's often represented visually as a diagram. Graphs are used in various mathematical and computer science applications.
- A graph can refer to a chart or diagram representing data in data visualization. This could include bar graphs, line graphs, pie charts, etc., used to visually represent relationships between different variables.
- In the context of social networks, a graph represents relationships between individuals or entities. Nodes might represent people, and edges represent connections or relationships between them.
- A graph can also be a visual representation of a mathematical function. It consists of points plotted on a coordinate plane, where the x-axis represents one variable, and the y-axis represents another.
- In computer science, a graph database uses graph structures to store and query data. It's particularly useful for data with complex relationships.

2 Types of Graph

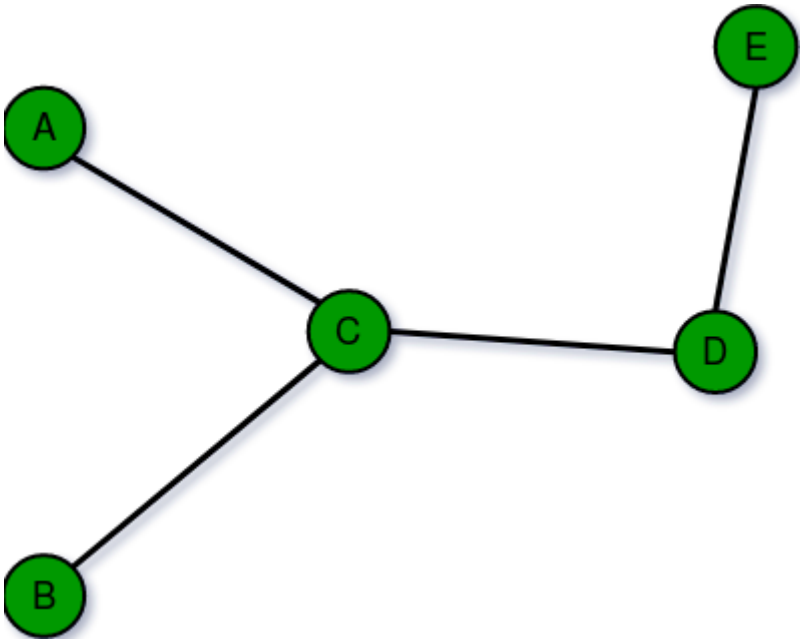
*A **Graph** is a non-linear data structure consisting of **nodes** and **edges**. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, A Graph consisting of a finite set of vertices(or nodes) and a set of edges that connect a pair of nodes*

1. **Undirected Graphs**: A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional.
2. **Directed Graphs**: A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.
3. **Weighted Graphs:** A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.
4. **Unweighted Graph**s: A graph in which edges have no weights or costs associated with them. Example: A social network graph where the edges represent friendships.
5. **Complete Graphs:** A graph in which each vertex is connected to every other vertex. Example: A tournament graph where every player plays against every other player.
6. **Bipartite Graphs:** A graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. Example: A job applicant graph where the vertices can be divided into job applicants and job openings.
7. **Trees**: A connected graph with no cycles. Example: A family tree where each person is connected to their parents.
8. **Cycles**: A graph with at least one cycle. Example: A bike-sharing graph where the cycles represent the routes that the bikes take.
9. **Sparse Graphs:** A graph with relatively few edges compared to the number of vertices. Example: A chemical reaction graph where each vertex represents a chemical compound and each edge represents a reaction between two compounds.
10. **Dense Graph**s: A graph with many edges compared to the number of vertices. Example: A social network graph where each vertex represents a person and each edge represents a friendship.

2.1 Types of Graphs:

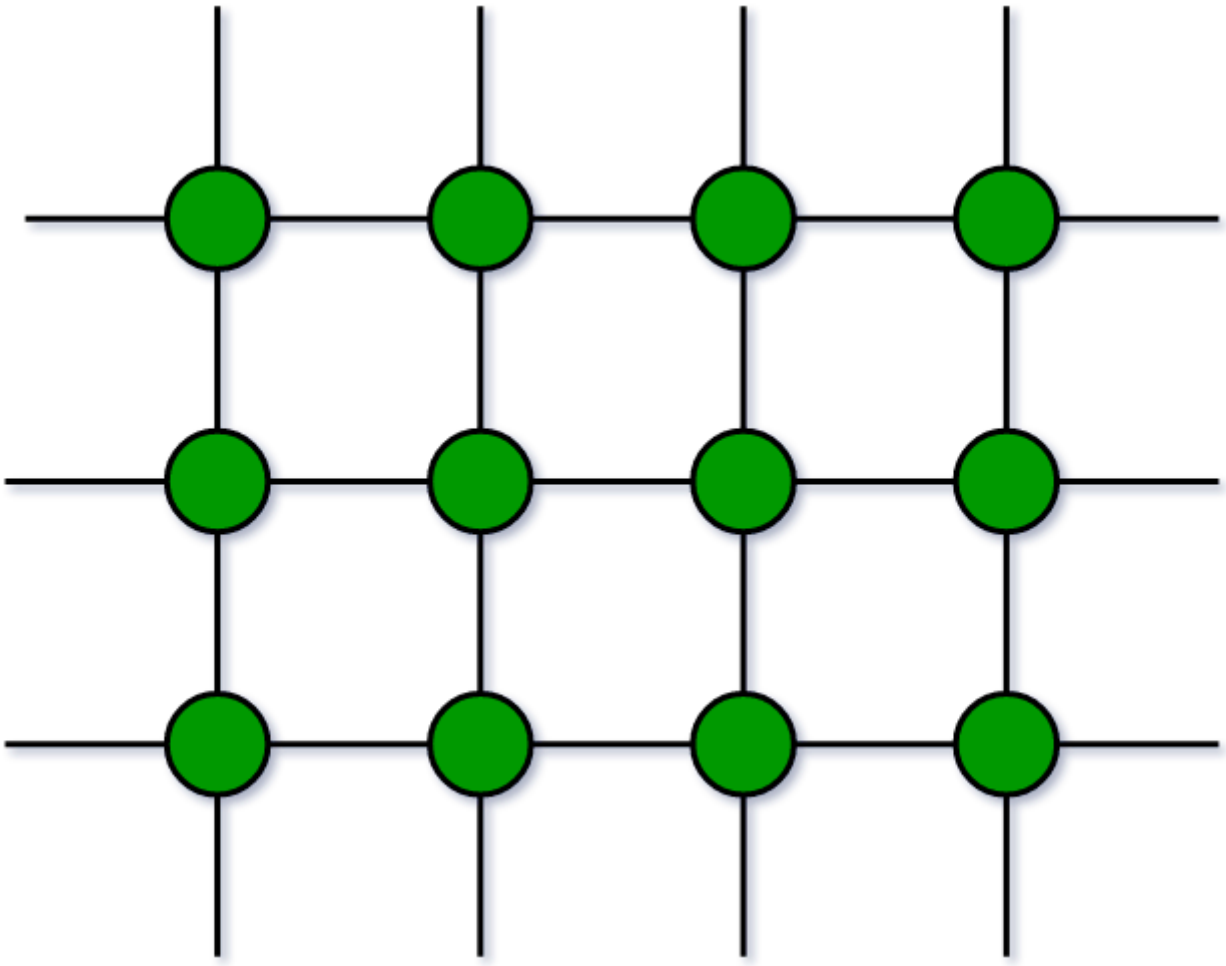
2.1.1 Finite Graphs

A graph is said to be finite if it has a finite number of vertices and a finite number of edges. A finite graph is a graph with a finite number of vertices and edges. In other words, both the number of vertices and the number of edges in a finite graph are limited and can be counted. Finite graphs are often used to model real-world situations, where there is a limited number of objects and relationships between the



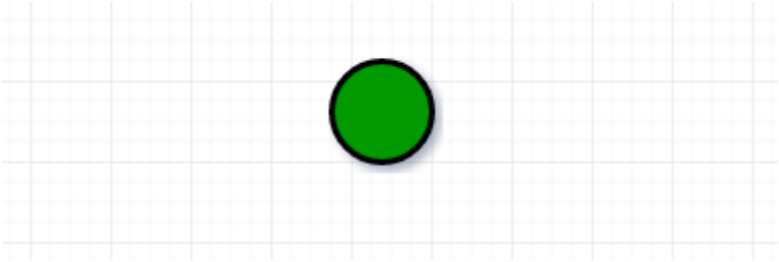
2.1.2 Infinite Graph:

A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.



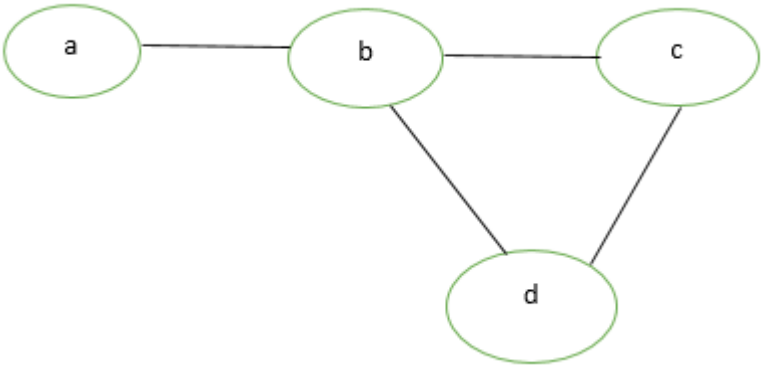
2.1.3 Trivial Graph:

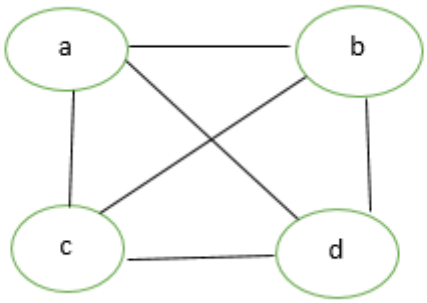
A graph is said to be trivial if a finite graph contains only one vertex and no edge. A trivial graph is a graph with only one vertex and no edges. It is also known as a singleton graph or a single vertex graph. A trivial graph is the simplest type of graph and is often used as a starting point for building more complex graphs. In graph theory, trivial graphs are considered to be a degenerate case and are not typically studied in detail



2.1.4 Simple Graph:

A simple graph is a graph that does not contain more than one edge between the pair of vertices. A simple railway track connecting different cities is an example of a simple graph.



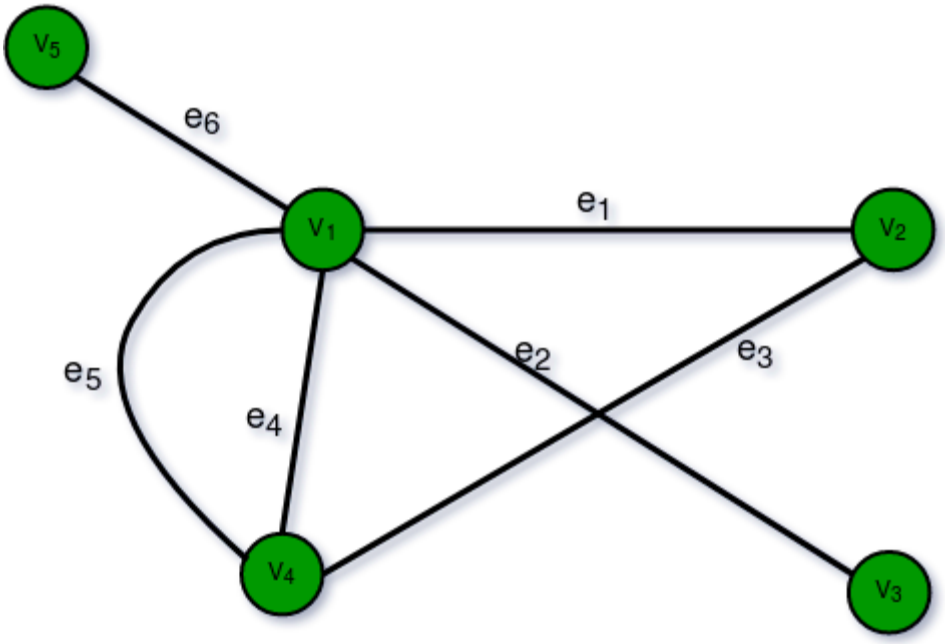


2.1.5 Multi Graph:

Any graph which contains some parallel edges but doesn't contain any self-loop is called a multigraph. For example a Road Map.

- **Parallel Edges:** If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.
- **Loop:** An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a

self-loop.

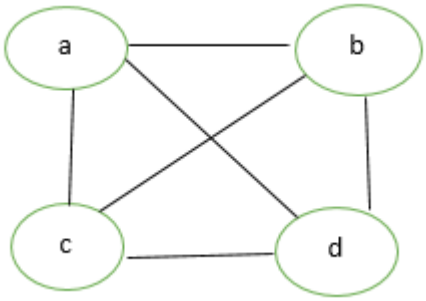


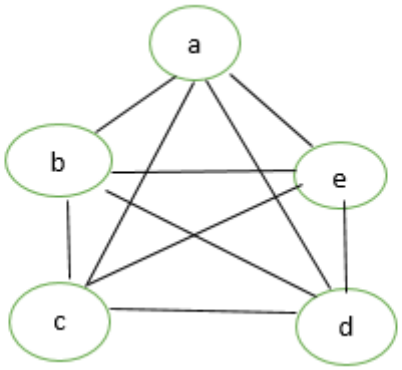
2.1.6 Null Graph:

A graph of order n and size zero is a graph where there are only isolated vertices with no edges connecting any pair of vertices.A null graph is a graph with no edges. In other words, it is a graph with only vertices and no connections between them. A null graph can also be referred to as an edgeless graph, an isolated graph, or a discrete graph

2.1.7 Complete Graph:

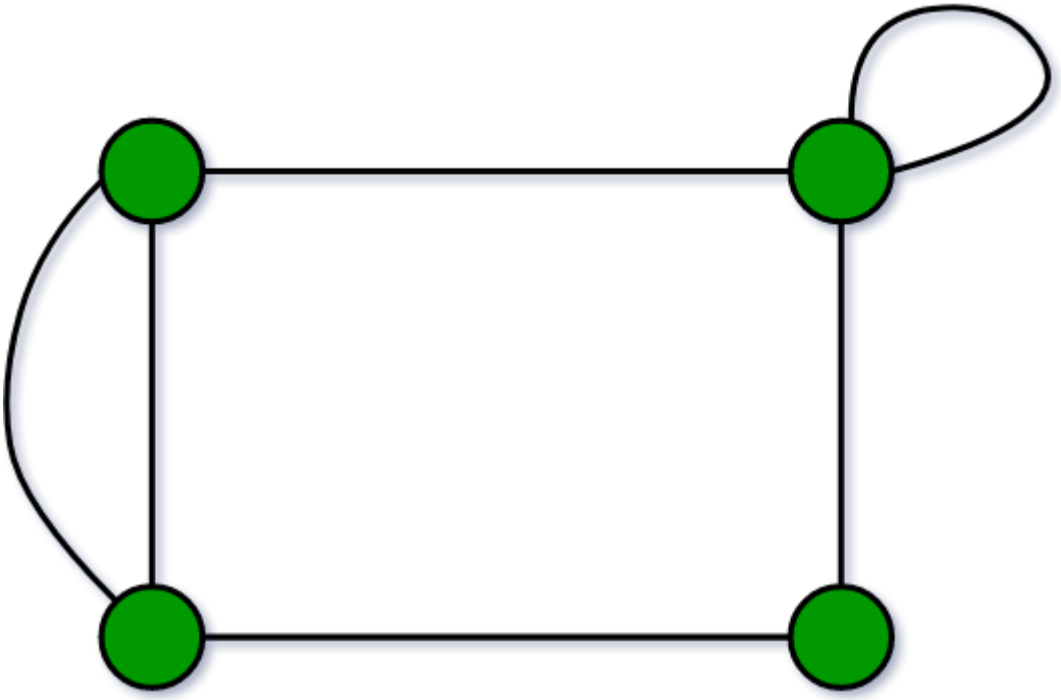
A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attached with n-1 edges or the rest of the vertices in the graph. A complete graph is also called Full Graph.





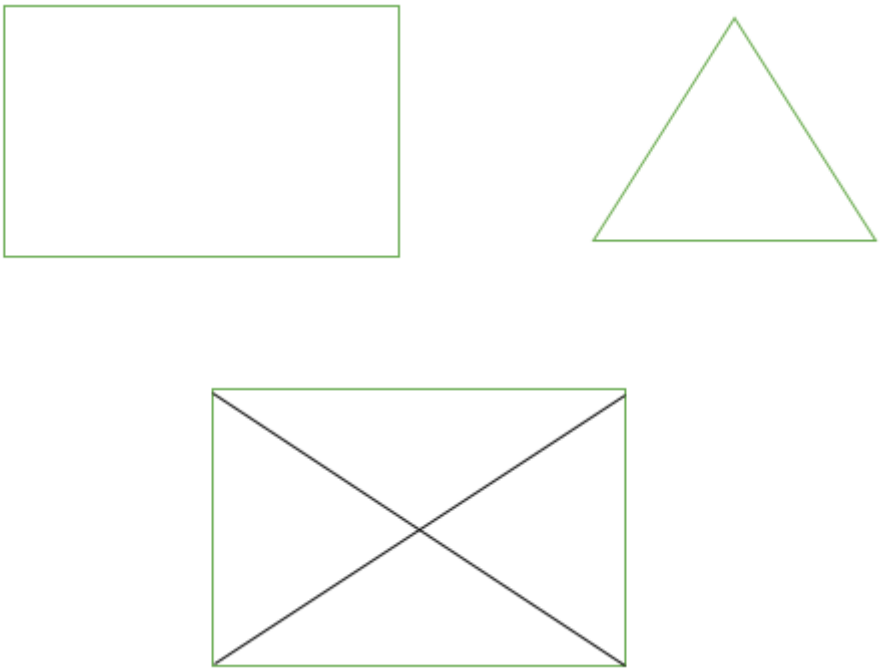
2.1.8 Pseudo Graph:

A graph G with a self-loop and some multiple edges is called a pseudo graph. A pseudograph is a type of graph that allows for the existence of loops (edges that connect a vertex to itself) and multiple edges (more than one edge connecting two vertices). In contrast, a simple graph is a graph that does not allow for loops or multiple edges.



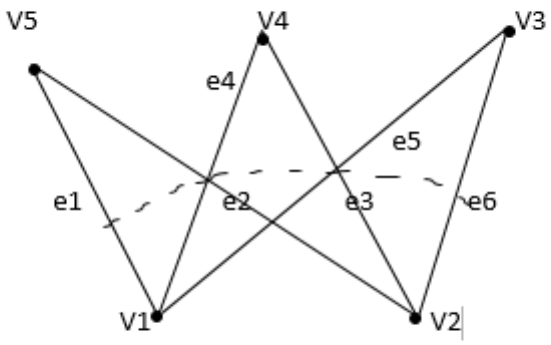
2.1.9 Regular Graph:

A simple graph is said to be regular if all vertices of graph G are of equal degree. All complete graphs are regular but vice versa is not possible. A regular graph is a type of undirected graph where every vertex has the same number of edges or neighbors. In other words, if a graph is regular, then every vertex has the same degree.



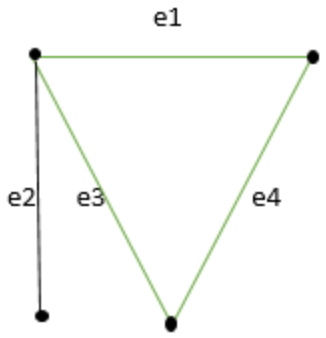
2.1.10 Bipartite Graph:

A graph G = (V, E) is said to be a bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has one end in V1(G) and another end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure: V1(G)={V5, V4, V3} and V2(G)={V1, V2}



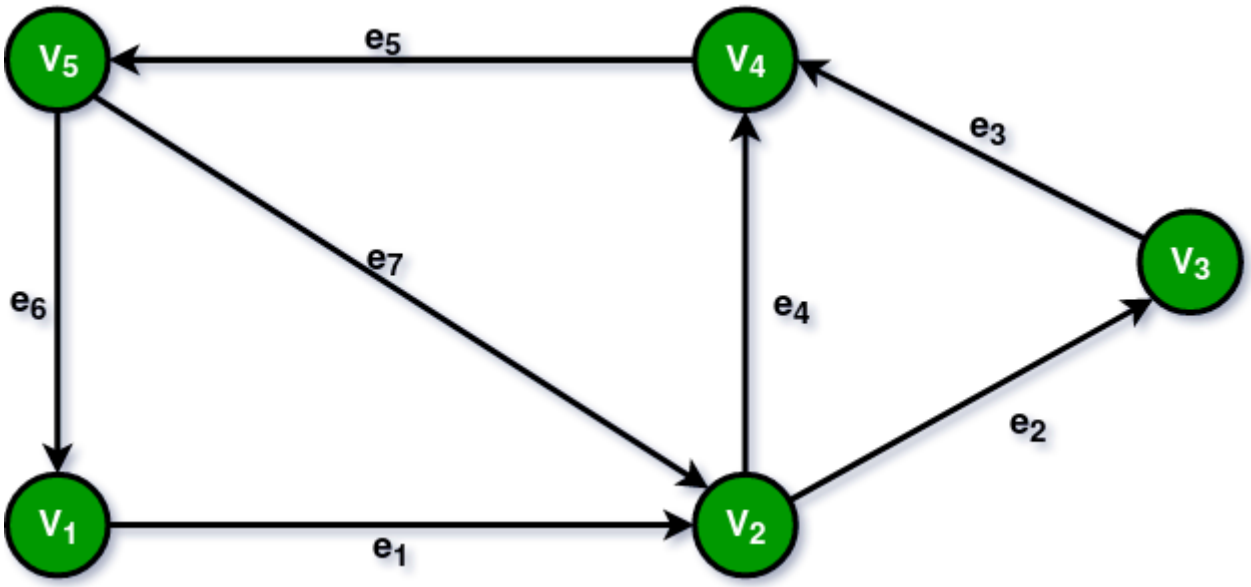
2.1.11 Labeled Graph:

If the vertices and edges of a graph are labeled with name, date, or weight then it is called a labeled graph. It is also called Weighted Graph.



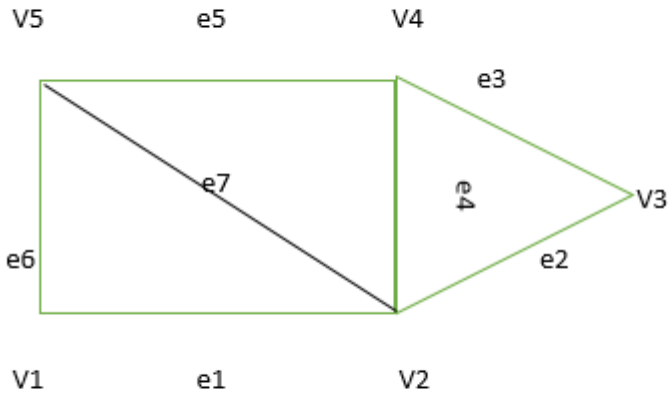
2.1.12 Digraph Graph:

A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) are called a Digraph. It is also called *Directed Graph*. The ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj. Here in the figure: e1 = (V1, V2) e2 = (V2, V3) e4 = (V2, V4)



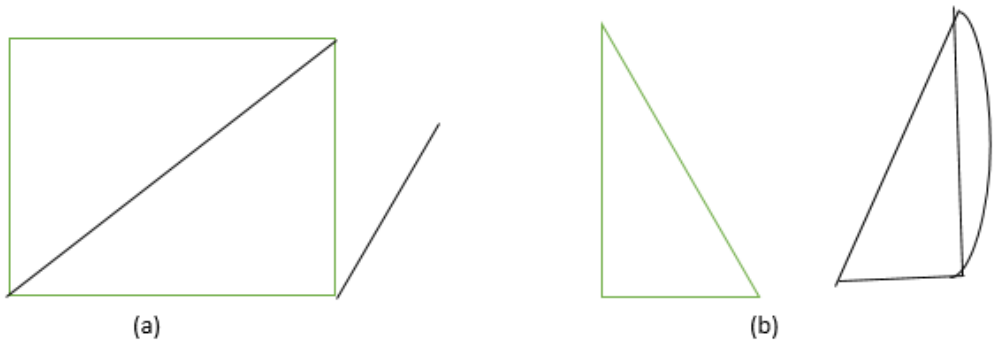
2.1.13 Subgraph:

A graph G1 = (V1, E1) is called a subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



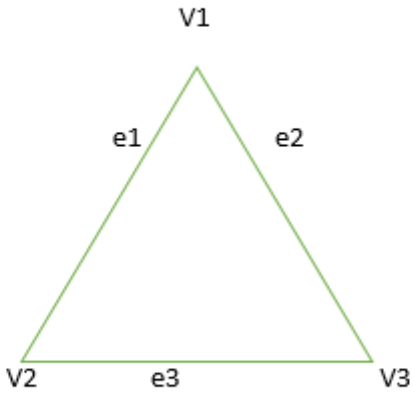
2.1.14 Connected or Disconnected Graph:

Graph G is said to be connected if any pair of vertices (Vi, Vj) of a graph G is reachable from one another. Or a graph is said to be connected if there exists at least one path between each and every pair of vertices in graph G, otherwise, it is disconnected. A null graph with n vertices is a disconnected graph consisting of n components. Each component consists of one vertex and no edge.



2.1.15 Cyclic Graph:

A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – (Vn, V1) are called cyclic graph.



2.1.16 Vertex disjoint subgraph:

Any two graph G1 = (V1, E1) and G2 = (V2, E2) are said to be vertex disjoint of a graph G = (V, E) if V1(G1) intersection V2(G2) = null. In the figure, there is no common vertex between G1 and G2.

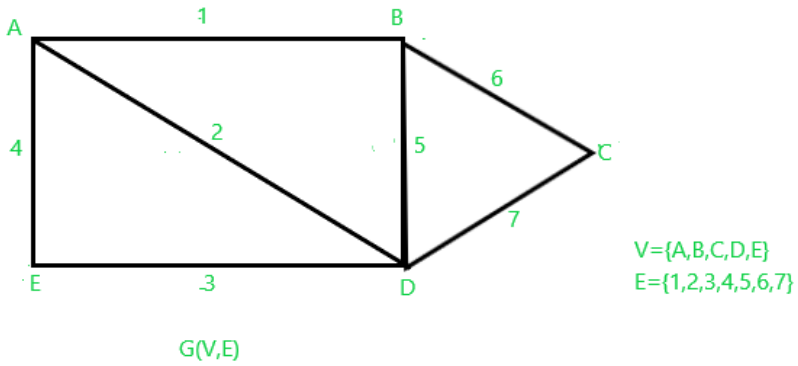
2.1.17 Edge disjoint subgraph:

A subgraph is said to be edge-disjoint if E1(G1) intersection E2(G2) = null. In the figure, there is no common edge between G1 and G2.

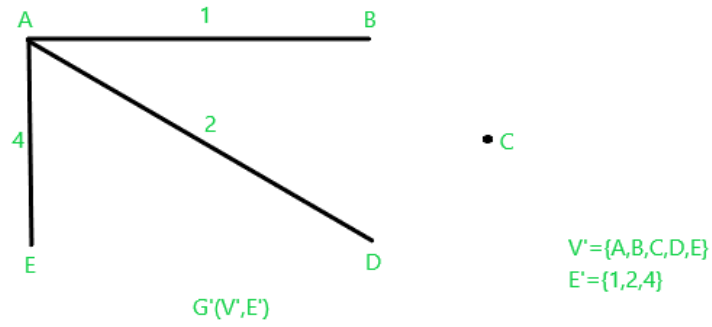
Note: Edge disjoint subgraph may have vertices in common but a vertex disjoint graph cannot have a common edge, so the vertex disjoint subgraph will always be an edge-disjoint subgraph.

2.1.18 Spanning Subgraph

Consider the graph G(V,E) as shown below. A spanning subgraph is a subgraph that contains all the vertices of the original graph G that is G'(V',E') is spanning if V'=V and E' is a subset of E.



So one of the spanning subgraph can be as shown below G'(V',E'). It has all the vertices of the original graph G and some of the edges of G.



This is just one of the many spanning subgraphs of graph G. We can create various other spanning subgraphs by different combinations of edges. Note that if we consider a graph G'(V',E') where V'=V and E'=E, then graph G' is a spanning subgraph of graph G(V,E).

2.2 Advantages of graphs:

1. Graphs can be used to model and analyze complex systems and relationships.
2. They are useful for visualizing and understanding data.
3. Graph algorithms are widely used in computer science and other fields, such as social network analysis, logistics, and transportation.
4. Graphs can be used to represent a wide range of data types, including social networks, road networks, and the internet.

2.3 Disadvantages of graphs:

1. Large graphs can be difficult to visualize and analyze.
2. Graph algorithms can be computationally expensive, especially for large graphs.
3. The interpretation of graph results can be subjective and may require domain-specific knowledge.
4. Graphs can be susceptible to noise and outliers, which can impact the accuracy of analysis results.

3 Graph Data Structure

A **Graph** is a non-linear 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)**.

3.1 Representations of Graph

Here are the two most common ways to represent a graph :

1. Adjacency Matrix
2. Adjacency List

3.1.1 Adjacency matrix

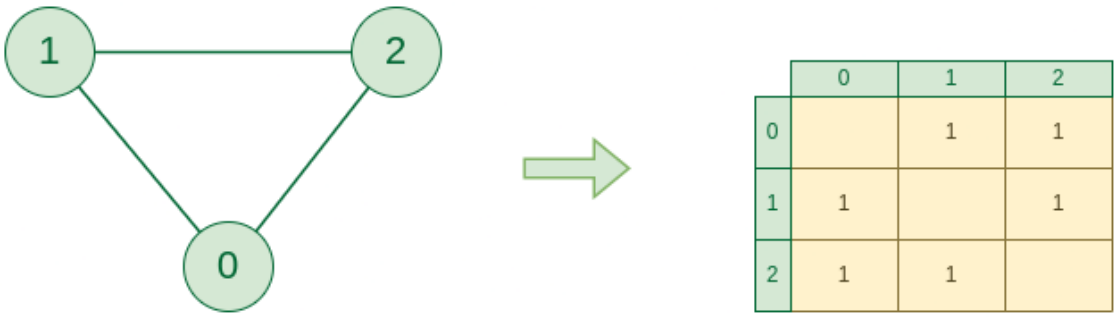
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's).

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*

Representation of Undirected Graph to Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat[destination]**) because we can go either way.



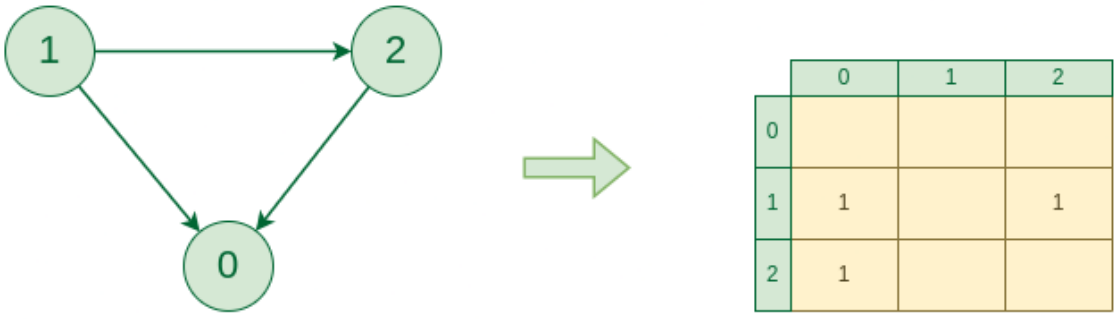
Undirected Graph | Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Undirected Graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



Directed Graph | Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Directed Graph to Adjacency Matrix

3.1.2 Adjacency List

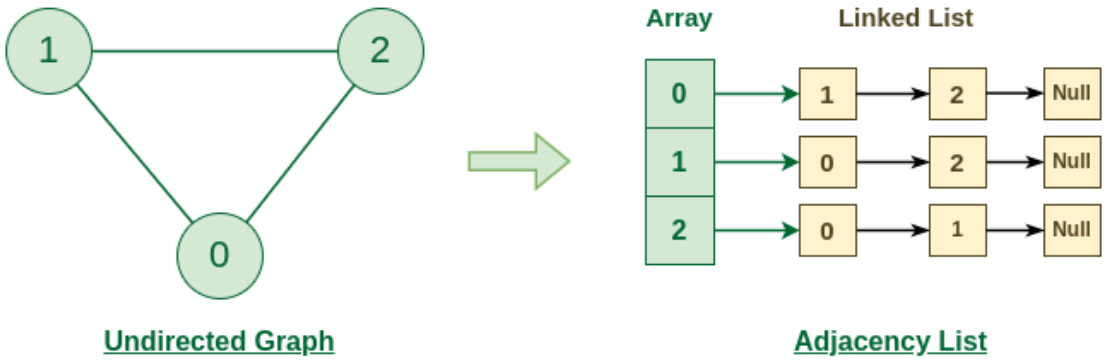
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph to Adjacency list:

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 1) So, insert vertices 2 and 1 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

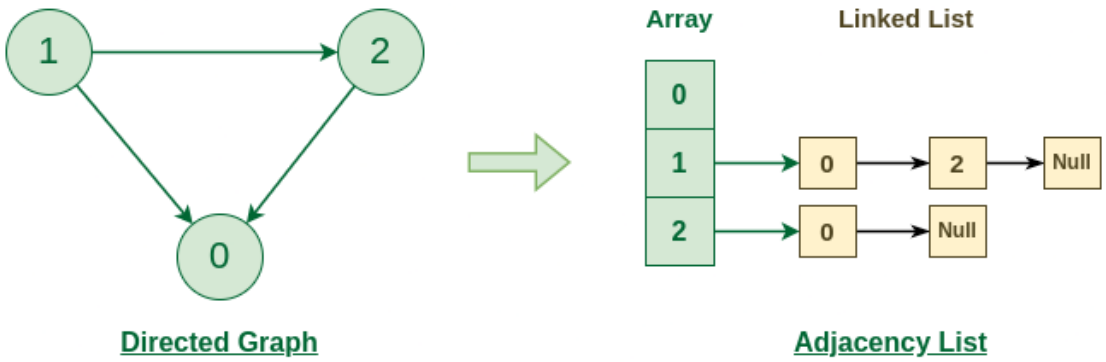


Graph Representation of Undirected graph to Adjacency List

Undirected Graph to Adjacency list

Representation of Directed Graph to Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

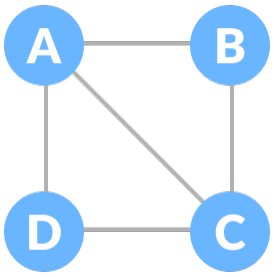


Graph Representation of Directed graph to Adjacency List

4 Spanning Tree and Minimum Spanning Tree

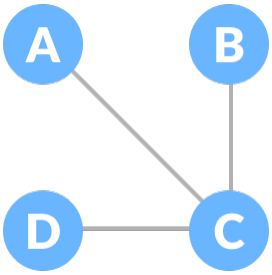
Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

An **undirected graph** is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Undirected Graph

A **connected graph** is a graph in which there is always a path from a vertex to any other vertex.



Connected Graph

4.1 Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

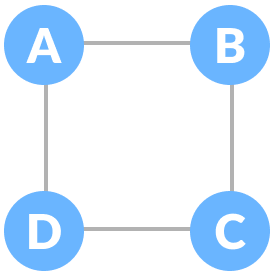
The edges may or may not have weights assigned to them.

If we have n = 4, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

Example of a Spanning Tree

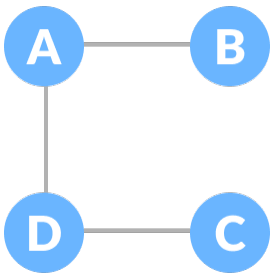
Let's understand the spanning tree with examples below:

Let the original graph be:

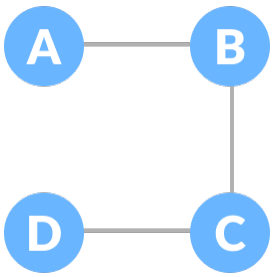


Normal graph

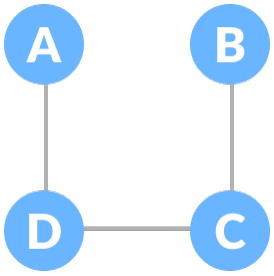
Some of the possible spanning trees that can be created from the above graph are:



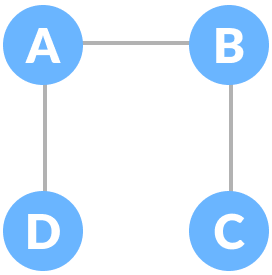
A spanning tree



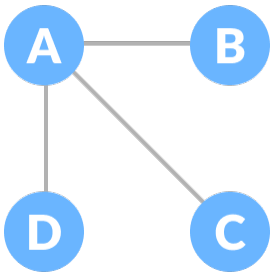
A spanning tree



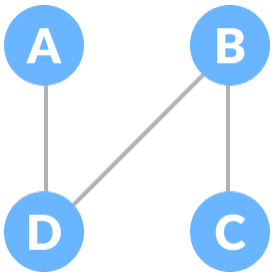
A spanning tree



A spanning tree



A spanning tree



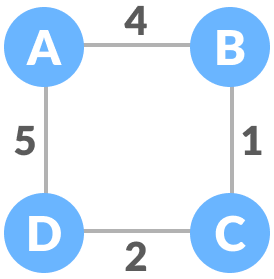
A spanning tree

4.2 Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

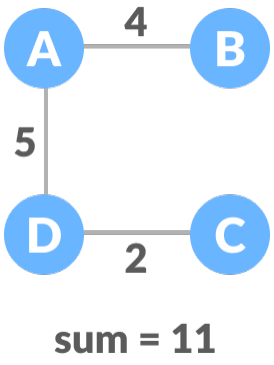
Example of a Spanning Tree

The initial graph is:

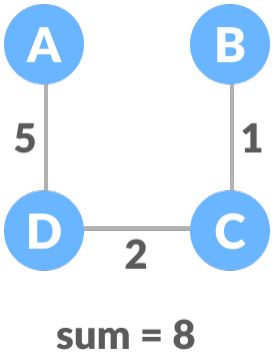


Weighted graph

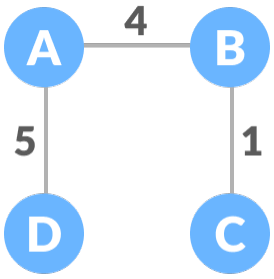
The possible spanning trees from the above graph are:



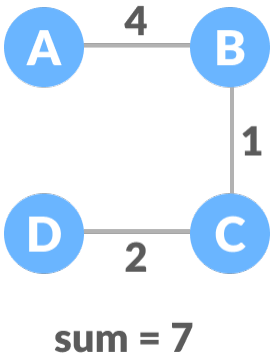
Minimum spanning tree – 1



Minimum spanning tree – 2

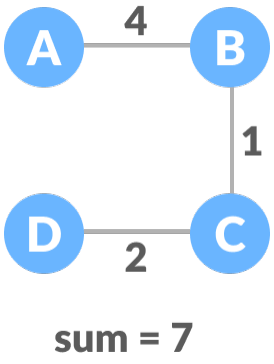


Minimum spanning tree – 3



Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

4.2.1 Spanning Tree Applications

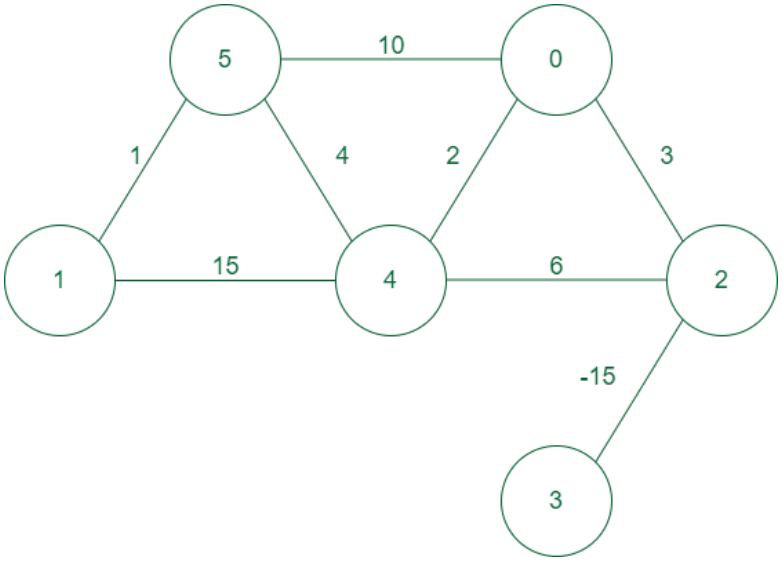
- Computer Network Routing Protocol
- Cluster Analysis
- Civil Network Planning

4.2.2 Minimum Spanning tree Applications

- To find paths in the map
- To design networks like telecommunication networks, water supply networks, and electrical grids.

5 Weighted graphs

A **weighted graph** is defined as a special type of graph in which the edges are assigned some weights which represent cost, distance, and many other relative measuring units.



Example of a weighted graph

5.1 Applications of Weighted Graph:

- **2D matrix games:** In 2d matrix, games can be used to find the optimal path for maximum sum along starting to ending points and many variations of it can be found online.
- **Spanning trees:** Weighted graphs are used to find the minimum spanning tree from graph which depicts the minimal cost to traverse all nodes in the graph.
- **Constraints graphs:** Graphs are often used to represent constraints among items. Used in scheduling, product design, asset allocation, circuit design, and artificial intelligence.
- **Dependency graphs:** Directed weighted graphs can be used to represent dependencies or precedence order among items. Priority will be assigned to provide a flow in which we will solve the problem or traverse the graph from highest priority to lowest priority. Such graphs are often used in large projects

in laying out what components rely on other components and are used to minimize the total time or cost to completion while abiding by the dependencies.

- **Compilers:** Weighted graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, and many other purposes such as query optimization in database languages.
- **Artificial Intelligence:** Weighted graphs are used in artificial intelligence for decision-making processes, such as in game trees for determining the best move in a game.
- **Image Processing:** Weighted graphs are used in image processing for segmentation, where the weight of the edges represents the similarity between two pixels.
- **Natural Language Processing:** Weighted graphs are used in natural language processing for text classification, where the weight of the edges represents the similarity between two words.

5.2 Real-Time Applications of Weighted Graph:

- **Transportation networks:** Using weighted graphs, we can figure things out like the path that takes the least time, or the path with the least overall distance. This is a simplification of how weighted graphs can be used for more complex things like a GPS system. Graphs are used to study traffic patterns, traffic light timings and much more by many big tech companies such as OLA, UBER, RAPIDO, etc. Graph networks are used by many map programs such as Google Maps, Bing Maps, etc.
- **Document link graphs:** Link weighted graphs are used to analyze relevance of web pages, the best sources of information, and good link sites by taking the count of the number of views as weights in the graph.
- **Epidemiology:** Weighted graphs can be used to find the maximum distance transmission from an infectious to a healthy person.
- **Graphs in quantum field theory:** Vertices represent states of a quantum system and the edges represent transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude. Research to find the maximum frequency along a path can be done using weighted graphs.
- **Social network graphs:** We can find which all users are connected in a network both directly(direct connection) and indirectly(indirect connection). But now weighted graphs are also used in social media for many purposes, for example, In recent times Instagram is using features like close friends which is not the same as all friends these features are being implemented using weighted graphs.
- **Network packet traffic graphs:** Network packet traffic graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

5.3 Advantages of Weighted Graph:

- **Better representation of real-world scenarios**: Weighted graphs are a more accurate representation of many real-world scenarios, where the relationships between entities have varying degrees of importance.

For example, in a road network, some roads may have higher speed limits or more lanes, and these differences can be represented using weights.

- **More accurate pathfinding**: In a weighted graph, finding the shortest path between two nodes takes into account the weights of the edges, which can lead to more accurate results. This is particularly useful in applications where finding the optimal path is critical, such as in logistics or transportation planning.
- **More efficient algorithms**: Many graph algorithms are more efficient when applied to weighted graphs, such as Dijkstra's algorithm for finding the shortest path. This is because the weights provide additional information that can be used to optimize the search.
- **More flexible analysis:** Weighted graphs allow for more flexible analysis of the relationships between nodes. For example, it is possible to calculate the average weight of edges, or to identify nodes with unusually high or low weights. This can provide insights into the structure of the graph and the relationships between its nodes.
- **Ability to model uncertainty**: Weighted graphs can be used to model uncertain or probabilistic relationships between nodes. For example, in a social network, the weight of an edge could represent the probability of a connection between two people, rather than a binary "friend" or "not friend" relationship.

5.4 Disadvantages of Weighted Graph:

- **Increased complexity:** Weighted graphs are more complex than unweighted graphs, and can be more difficult to understand and analyze. This complexity can make it harder to develop and debug algorithms that operate on weighted graphs.
- **Higher memory usage:** Weighted graphs require more memory than unweighted graphs, because each edge has an associated weight. This can be a problem for applications that have limited memory resources.
- **More difficult to maintain:** Weighted graphs can be more difficult to maintain than unweighted graphs, because changes to the weights of edges can have a ripple effect throughout the graph. This can make it harder to add or remove edges, or to update the weights of existing edges.
- **Not suitable for all applications:** Weighted graphs are not always the best choice for every application. For example, if the relationships between nodes are binary (i.e., either present or absent), an unweighted graph may be more appropriate.
- **Bias towards certain properties:** Weighted graphs can introduce bias towards certain properties, such as shortest path or highest weight. This can be a problem in applications where a more even distribution of weights is desired.

6 Shortest Path Algorithms

The shortest path problem is about finding a path between vertices in a graph such that the total sum of the edges weights is minimum. This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value

6.1 Types of Shortest Path Algorithms:

As we know there are various types of graphs (weighted, unweighted, negative, cyclic, etc.) therefore having a single algorithm that handles all of them efficiently is not possible. In order to tackle different problems, we have different shortest-path algorithms, which can be categorised into two categories:

6.1.1 Single Source Shortest Path Algorithms:

In this algorithm we determine the shortest path of all the nodes in the graph with respect to a single node i.e. we have only one source node in this algorithms.

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Multi-Source BFS
4. Dijkstra's algorithm
5. Bellman-Ford algorithm
6. Topological Sort
7. A* search algorithm

6.1.2 All Pair Shortest Path Algorithms:

Contrary to the single source shortest path algorithm, in this algorithm we determine the shortest path between every possible pair of nodes.

1. Floyd-Warshall algorithm
2. Johnson's algorithm

6.2 Shortest Path Algorithm using Depth-First Search(DFS):

DFS algorithm recursively explores the adjacent nodes untill it reaches to the depth where no more valid recursive calls are possible.

For DFS to be used as a shortest path algorithm, the graph needs to be acyclic i.e. a ***TREE,*** *the reason it won't work for cyclic graphs is because due to cycles, the destination node can have multiple paths from the source node and dfs will not be able to choose the best path.*

If there does not exist a path between source node and destination node then we can store -1 as the shortest path between those nodes.

Algorithm:

- Distance of source node to source node is initialized to 0.
- Start the DFS from the source node.
- As we go to a neighbouring nodes we set it's distance from source node = edge weight + distance of parent node.
- DFS ends when all the leaf nodes are visited.

Complexity Analysis:

- **Time Complexity**: O(N) where N is the number of nodes in the tree.
- **Auxiliary Space**: O(N) call stacks in case of skewed tree.

6.3 Breadth-First Search (BFS) for Shortest Path Algorithm:

BFS is a great shortest path algorithm for all graphs, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.

***Reason**: This works due to the fact that unlike DFS, BFS visits all the neighbouring nodes before exploring a single node to its neighbour. As a result, all nodes with distance '**d**' from the source are visited after all the nodes with distance **smaller than d.** In simple terms all the nodes with distance 1 from the node will be visited first, then distance 2 nodes, then 3 and so on.*

Algorithm:

- Use a queue to store a pair of values **{node, distance}**
- Distance of source node to source node is initialized to **0** i.e. push **{src, 0}** to the queue
- While queue is not empty do the following:
  - Get the **{node, distance}** value of top of the queue
  - Pop the top element of the queue
  - Store the distance for the node as the shortest distance for that node from the source
  - For all non visited neighbours of the current node, push **{neighbour, distance+1}** into the queue and mark it visited

Complexity Analysis:

- **Time Complexity**: O(N) where N is the number of nodes in the graph
- **Auxiliary Space:** O(N) for storing distance for each node.

7 Graph Traversal – Breadth first Traversal

*The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.*

7.1 Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

7.2 How does BFS work?

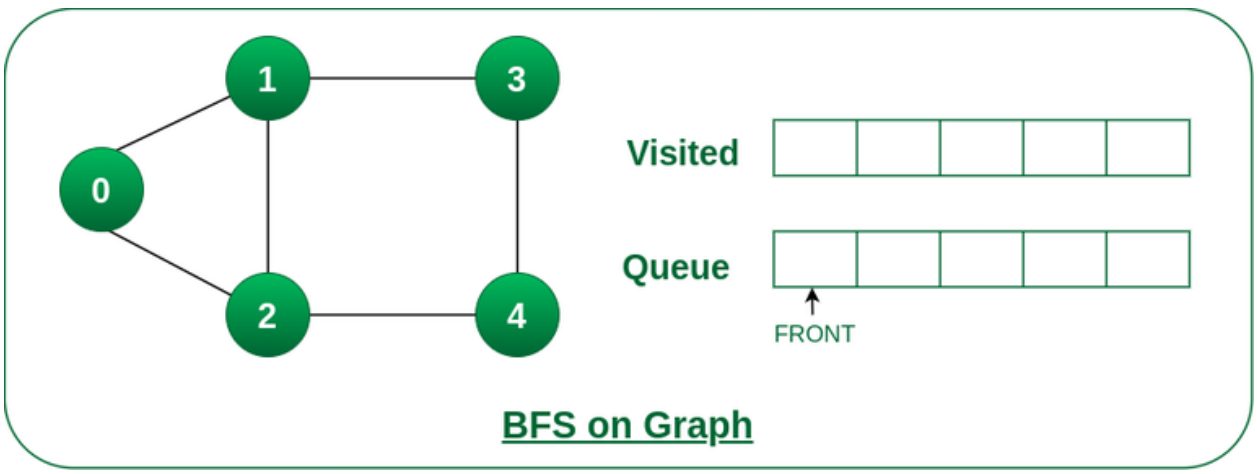
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

Illustration:

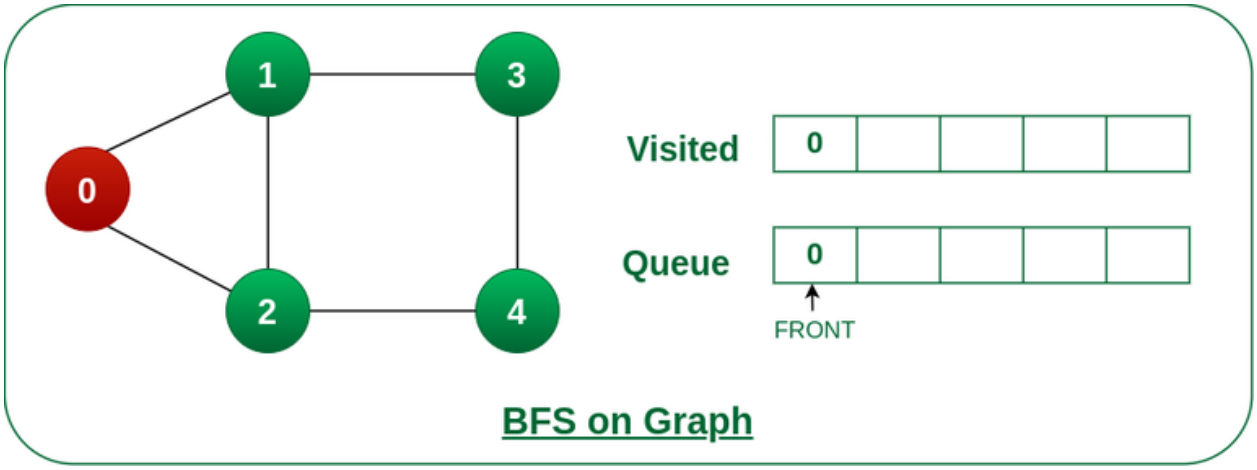
Let us understand the working of the algorithm with the help of the following example.

***Step1:** Initially queue and visited arrays are empty.*



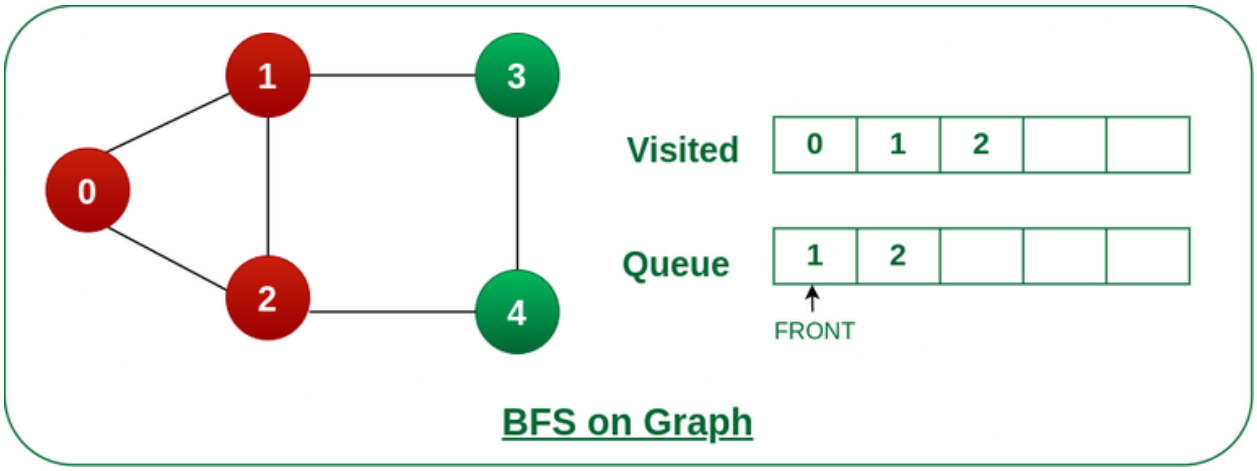
Queue and visited arrays are empty initially.

Step2: *Push node 0 into queue and mark it visited.*



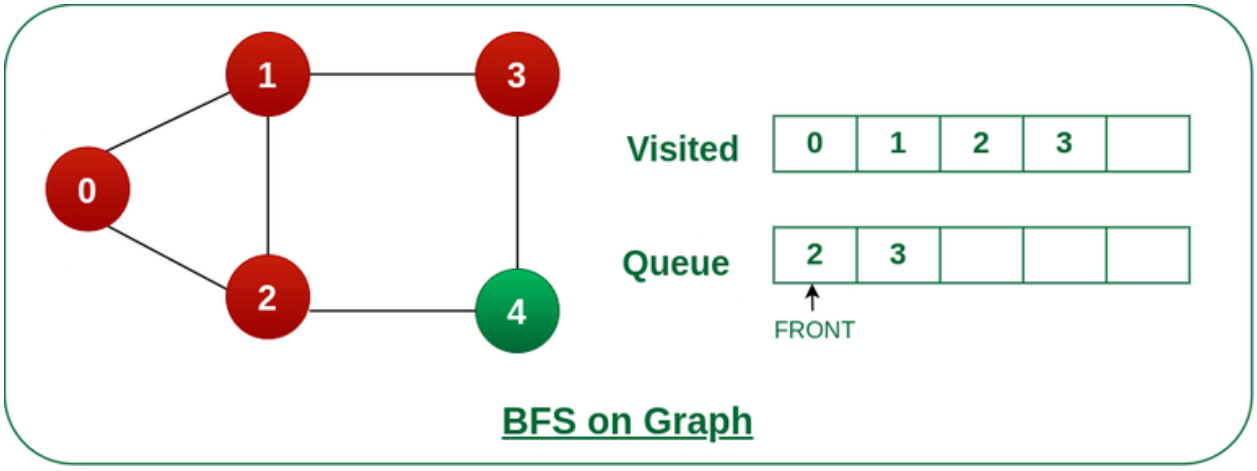
Push node 0 into queue and mark it visited.

Step 3: *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



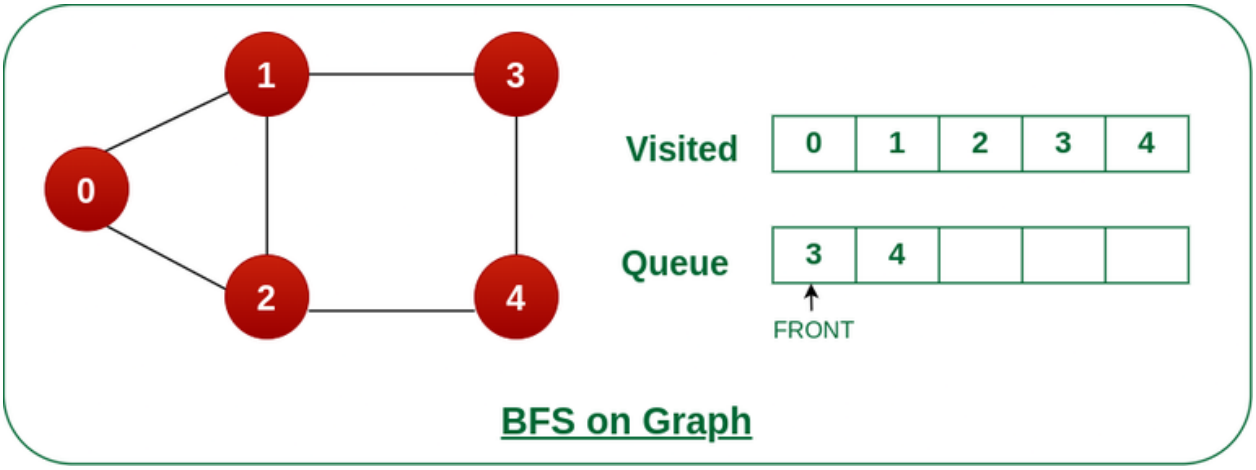
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



Remove node 1 from the front of queue and visited the unvisited neighbours and push

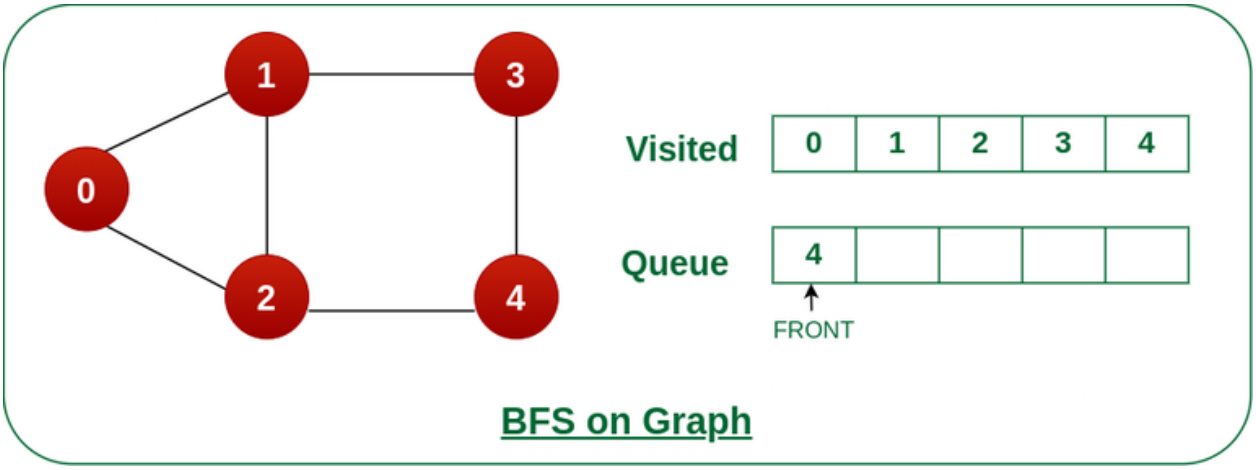
Step 5: *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

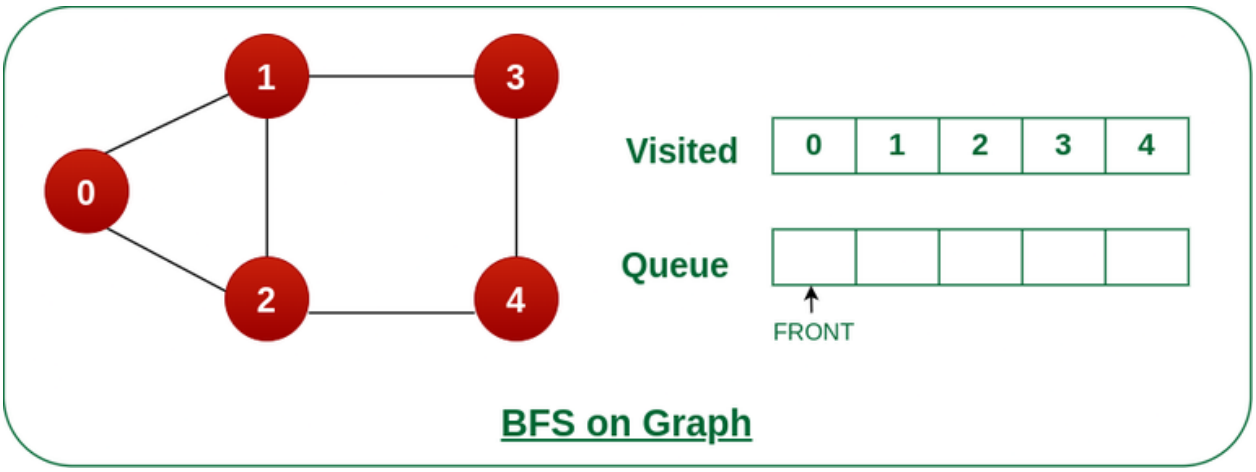
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: *Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

8 Depth first Traversal

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

8.1 Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

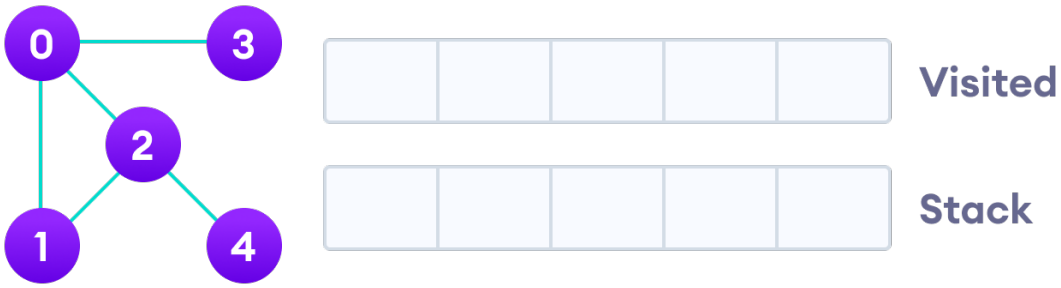
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

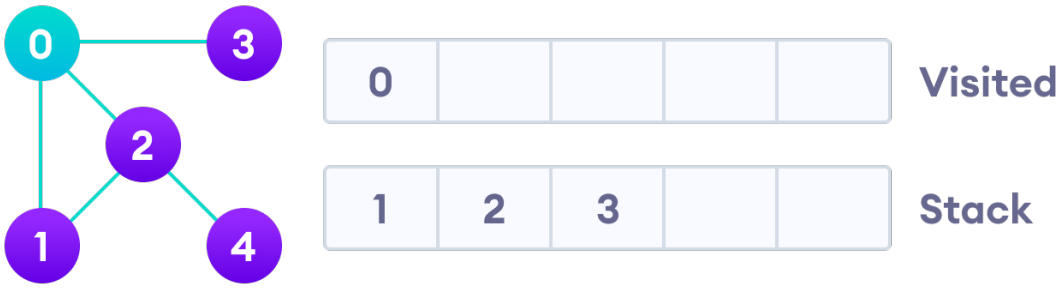
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



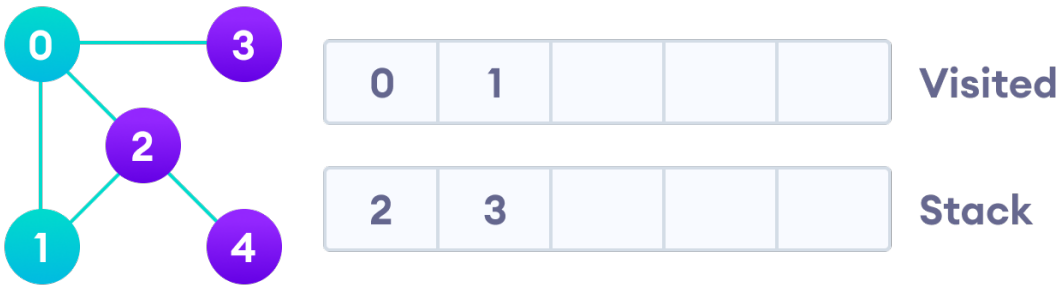
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



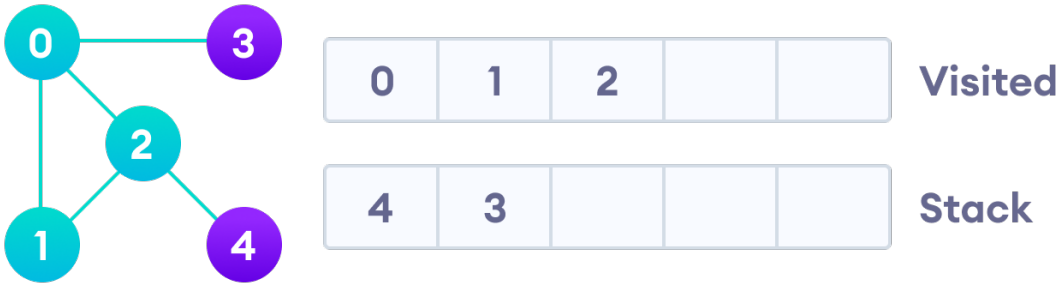
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

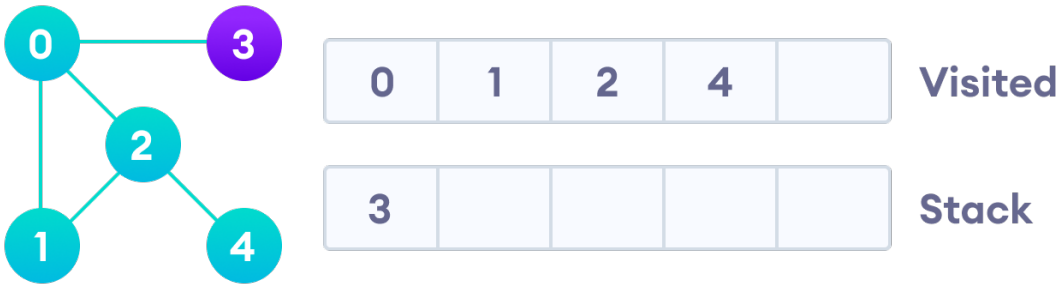


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

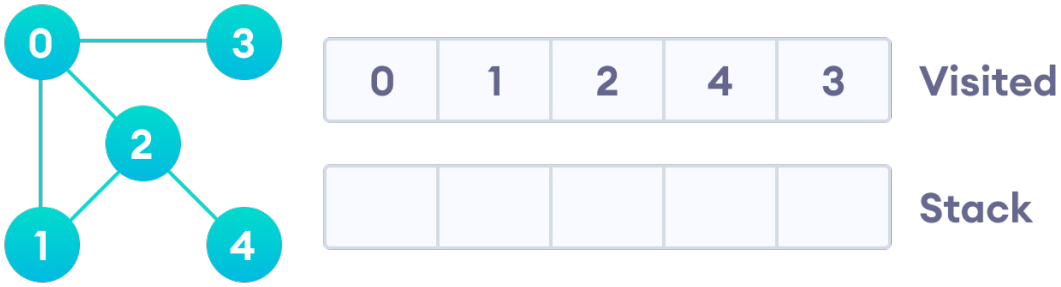


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

8.2 DFS Pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

DFS(G, u)

u.visited = true

for each v G.Adj[u]

if v.visited == false

DFS(G,v)

init() {

For each u G

u.visited = false

For each u G

DFS(G, u)

}

8.3 Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

8.4 Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

9 Connectivity of graphs

9.1 Connectivity in Undirected Graphs:

In graph theory, connectivity refers to the property of a graph to have a path connecting every pair of vertices. It is a fundamental concept that helps us understand the structure and relationships within a graph.

9.2 Connected Components:

A connected component of an undirected graph is a subgraph in which there is a path between every pair of vertices, and it is not possible to add an edge without breaking this property. In simpler terms, a connected component is a maximal connected subgraph.

For example, consider an undirected graph with three connected components:

1 - 2 3 6 - 7

|

4 – 5

In this graph, the connected components are {1, 2}, {3}, and {4, 5, 6, 7}.

9.2.1 Techniques for Finding Connected Components:

- **Depth-First Search (DFS):**

DFS is a graph traversal algorithm that can be used to find connected components. The idea is to start from a vertex, explore as far as possible along each branch before backtracking, and mark visited vertices.

Depth-First Search (DFS) traversal using a stack. Here's a breakdown of the steps:

1. **Create a Stack:**

- Create a stack with enough space to hold all the vertices in the graph.

2. **Choose a Starting Vertex:**

- Choose any vertex as the starting point for DFS traversal.

3. **Push Starting Vertex onto the Stack:**

- Push the chosen starting vertex onto the stack.

3. **Iterative DFS:**

a. Push Non-Visited Adjacent Vertex onto the Stack:

- While the stack is not empty, repeat the following:
  - If the vertex on the top of the stack has unvisited neighbors:
    - Push a non-visited neighbor of the top vertex onto the stack.
    - Mark the newly pushed vertex as visited.
    - If the vertex has no unvisited neighbors:
      - Pop the vertex from the stack.

Repeat Until Stack is Empty:

- Continue the process until the stack is empty.

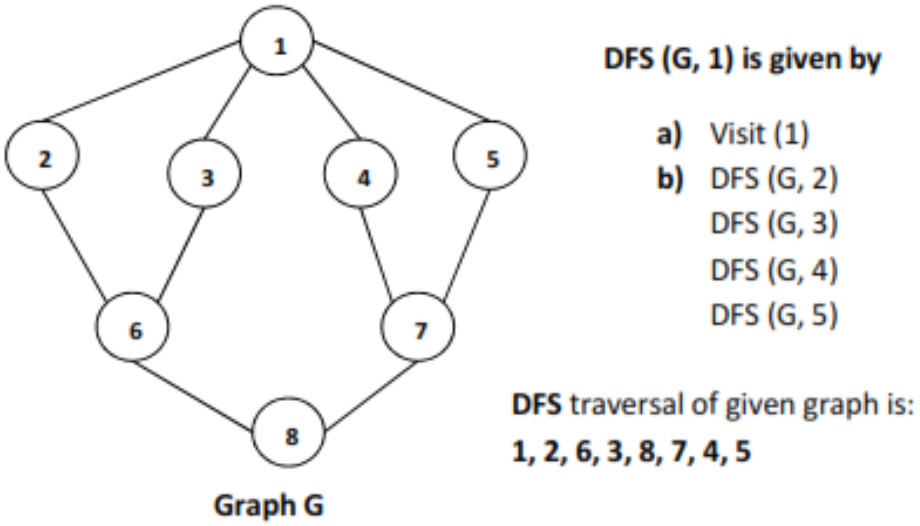


Fig – DFS Graph

C- Implementation of DFS:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100
struct Node {
int vertex;
struct Node* next;
};
struct Graph {
int numVertices;
struct Node** adjLists;
int* visited;
};
struct Stack {
int top;
int* array;
};
struct Stack* createStack(int size) {
struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
stack->top = -1;
stack->array = (int*)malloc(size * sizeof(int));
```

```
return stack;
}
void push(struct Stack* stack, int item) {
stack->array[++stack->top] = item;
}
int pop(struct Stack* stack) {
return stack->array[stack->top--];
}
struct Node* createNode(int vertex) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = vertex;
newNode->next = NULL;
return newNode;
}
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
graph->visited = (int*)malloc(vertices * sizeof(int));
for (int i = 0; i < vertices; ++i) {
graph->adjLists[i] = NULL;
graph->visited[i] = 0; // Mark all vertices as not visited
}
return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
struct Node* newNode = createNode(dest);
newNode->next = graph->adjLists[src];
graph->adjLists[src] = newNode;
}
void iterativeDFS(struct Graph* graph, int startVertex) {
struct Stack* stack = createStack(graph->numVertices);
push(stack, startVertex);
```

```
graph->visited[startVertex] = 1; // Mark the starting vertex as visited
while (stack->top != -1) {
int currentVertex = stack->array[stack->top];
printf("Visited Vertex: %d\n", currentVertex);
pop(stack);
struct Node* temp = graph->adjLists[currentVertex];
while (temp) {
int adjVertex = temp->vertex;
if (!graph->visited[adjVertex]) {
push(stack, adjVertex);
graph->visited[adjVertex] = 1; // Mark the adjacent vertex as visited
}
temp = temp->next;
}
}
free(stack->array);
free(stack);
}
int main() {
struct Graph* graph = createGraph(7);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 3);
addEdge(graph, 1, 4);
addEdge(graph, 2, 5);
addEdge(graph, 5, 6);
printf("DFS Order:\n");
iterativeDFS(graph, 0);
free(graph->adjLists);
free(graph->visited);
free(graph);
return 0;
}
```

- **Breadth-First Search (BFS):**

BFS is another traversal algorithm that can be used to find connected components. It explores vertices level by level, starting from the source vertex.

Algorithm: Breadth-First Search (BFS)

Input: Graph G, starting node A

Output: Processed nodes in BFS order

1. For each node in G, set STATUS = 1 (ready state).
2. Create an empty QUEUE.
3. Enqueue the starting node A and set its STATUS = 2 (waiting state).
4. Repeat Steps 5-6 until QUEUE is empty.
5. Dequeue a node N. Process it and set its STATUS = 3 (processed state).
6. Enqueue all neighbors of N that are in the ready state (STATUS = 1) and set their STATUS = 2 (waiting state).
7. EXIT

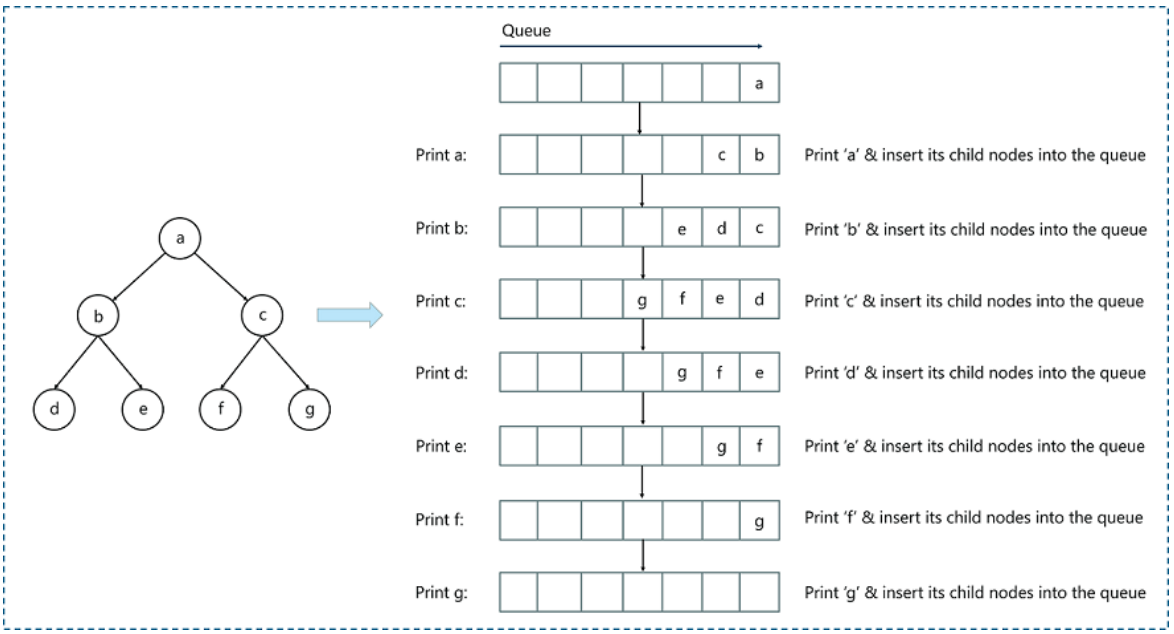


Fig – Breadth First Search (BFS)

C- Implementation of BFS:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
```

```
struct Node {
int vertex;
struct Node* next;
};
struct Graph {
int numVertices;
struct Node** adjLists;
int* status;
};
struct Queue {
int front, rear;
int* array;
};
struct Queue* createQueue(int size) {
struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
queue->front = queue->rear = -1;
queue->array = (int*)malloc(size * sizeof(int));
return queue;
}
void enqueue(struct Queue* queue, int item) {
queue->array[++queue->rear] = item;
if (queue->front == -1) {
queue->front = 0;
}
}
int dequeue(struct Queue* queue) {
int item = queue->array[queue->front++];
if (queue->front > queue->rear) {
queue->front = queue->rear = -1;
}
return item;
}
struct Node* createNode(int vertex) {
```

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = vertex;
newNode->next = NULL;
return newNode;
}
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
graph->status = (int*)malloc(vertices * sizeof(int));
for (int i = 0; i < vertices; ++i) {
graph->adjLists[i] = NULL;
graph->status[i] = 1; // Set STATUS = 1 for each node (ready state)
}
return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
struct Node* newNode = createNode(dest);
newNode->next = graph->adjLists[src];
graph->adjLists[src] = newNode;
}
void BFS(struct Graph* graph, int startVertex) {
struct Queue* queue = createQueue(graph->numVertices);
// Enqueue the starting node and set its STATUS = 2 (waiting state)
enqueue(queue, startVertex);
graph->status[startVertex] = 2;
while (queue->front != -1) {
// Dequeue a node N and process it
int current = dequeue(queue);
printf("Processed Node: %d\n", current);
// Enqueue neighbors of N that are in the ready state (STATUS = 1)
struct Node* adjList = graph->adjLists[current];
while (adjList) {
```

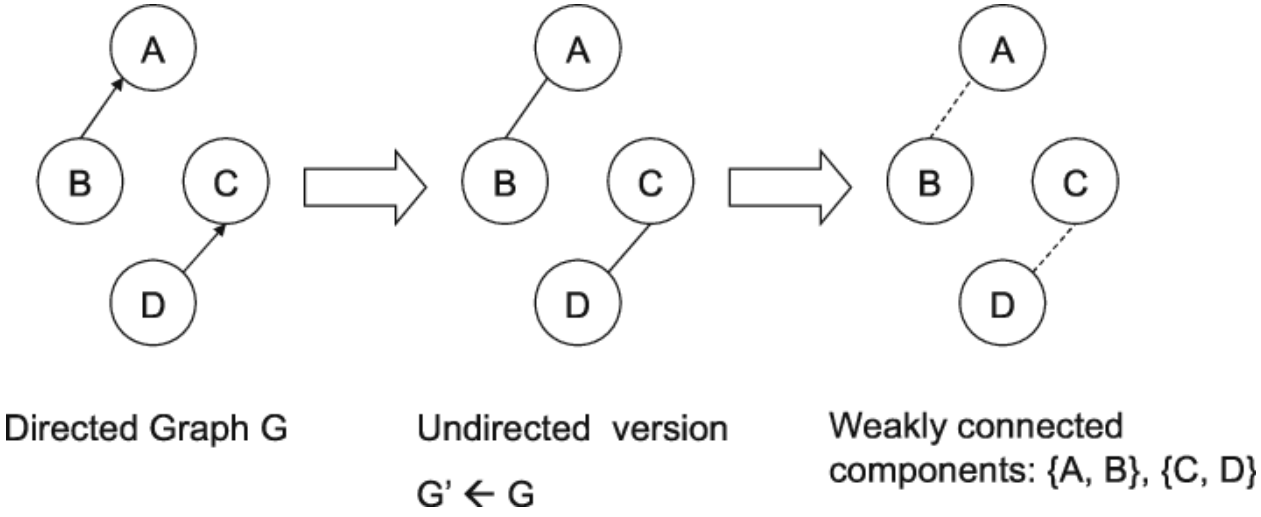
```
int neighbor = adjList->vertex;
if (graph->status[neighbor] == 1) {
enqueue(queue, neighbor);
graph->status[neighbor] = 2; // Set STATUS = 2 (waiting state)
}
adjList = adjList->next;
}
// Set the STATUS of N to 3 (processed state)
graph->status[current] = 3;
}
free(queue->array);
free(queue);
}
int main() {
struct Graph* graph = createGraph(6);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 3);
addEdge(graph, 1, 4);
addEdge(graph, 2, 5);
printf("BFS Order:\n");
BFS(graph, 0);
free(graph->adjLists);
free(graph->status);
free(graph);
return 0;
}
```

Connectivity in Directed Graphs:

In directed graphs, the concept of connectivity is extended to include both weakly connected components and strongly connected components.

Weakly Connected Components:

A directed graph is weakly connected if, when the directions of the edges are ignored, the underlying undirected graph is connected. In other words, there exists a path between every pair of vertices when disregarding edge directions.



Strongly Connected Components:

A directed graph is strongly connected if there is a directed path from any vertex to every other vertex. In other words, every vertex is reachable from every other vertex following the direction of edges.

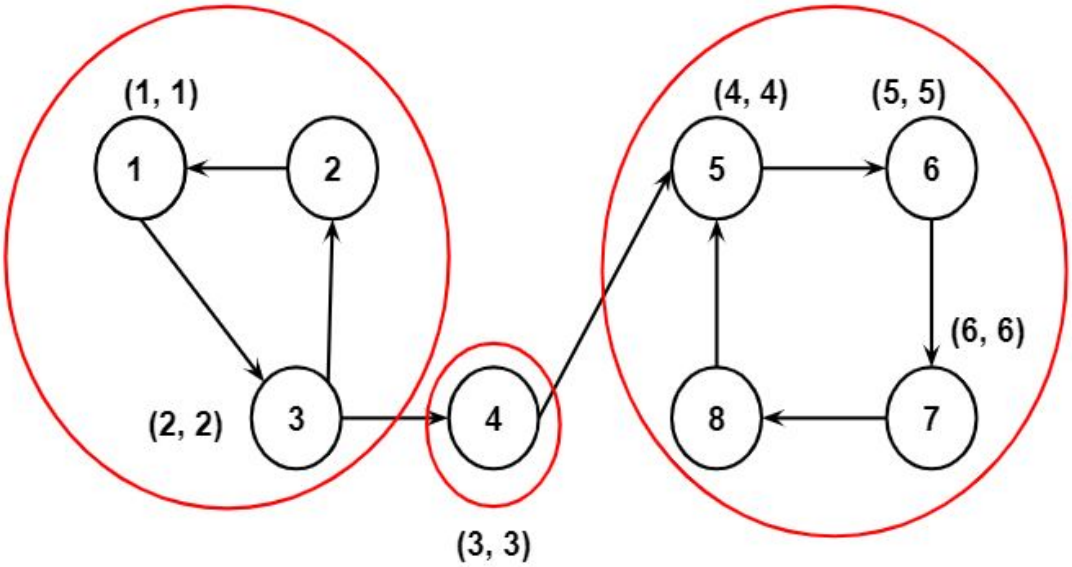


Fig – Strongly connected Component