DATA STRUCTURES AND ALGORITHMS

- LECTURE 9-

(2nd year of study)

Prof. Igor Lazov, PhD

Skopje, 2023/24

Contents

9.1. Introduction

- 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)** or **G = {V, E}**.

9.1. Introduction

- Graph data structures are a powerful tool for representing and analyzing complex *relationships* between *objects* or *entities*.
- They are particularly useful in fields such as: social network analysis, recommendation systems, and computer networks.
- In the field of sports data science, graph data structures can be used to analyze and understand the *dynamics* of team performance and player interactions on the field.

9.1. Introduction

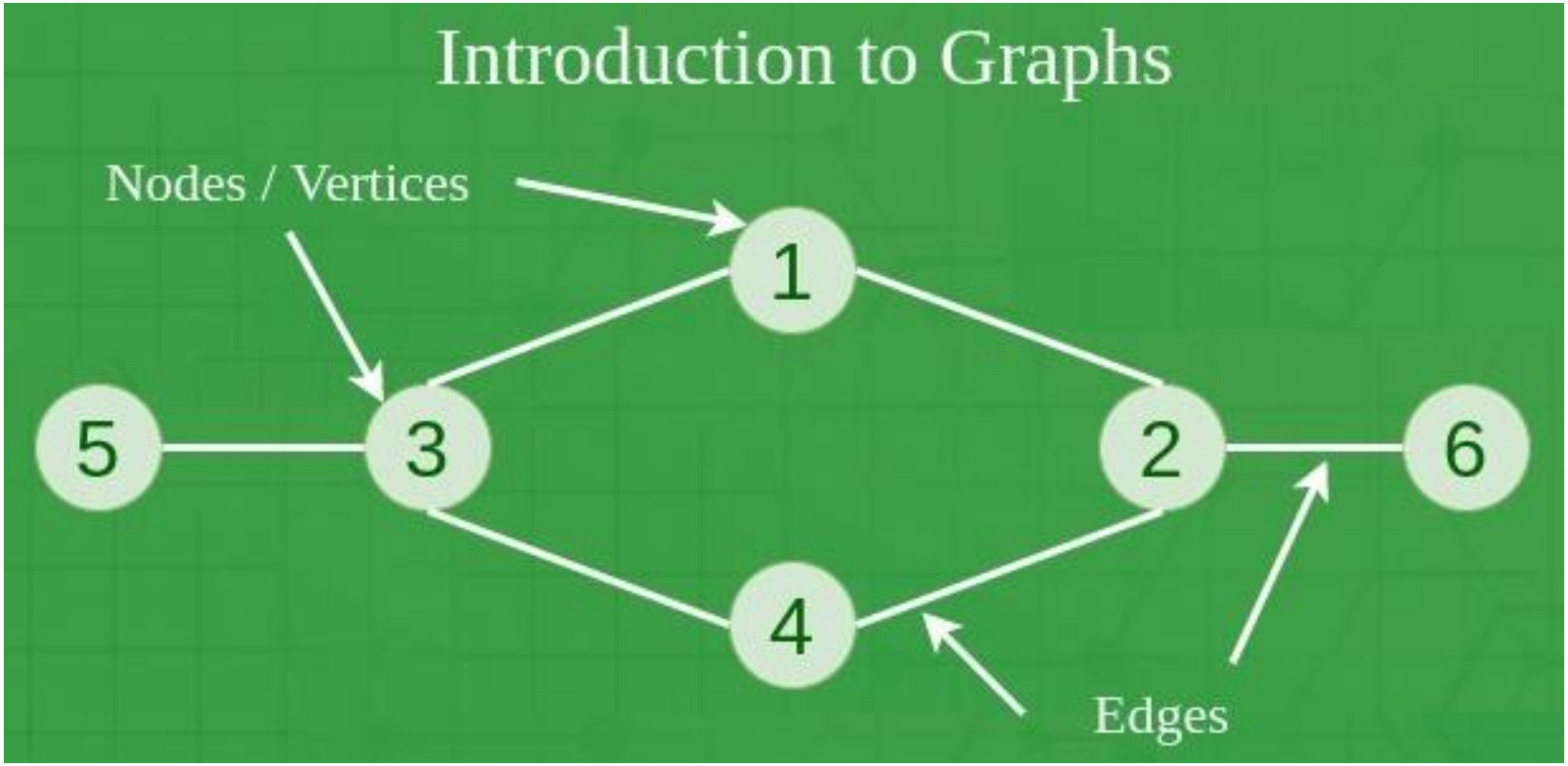
- Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges.
- This web of connections is exactly what a graph data structure represents, and it's the key to unlocking insights into team performance and player dynamics in sports.

9.1. Introduction

- **Components of a Graph**
  - **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as 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.

9.1. Introduction

- *Graph data structure* has ***nodes/vertices***, and ***edges***.



9.2. Types of Graphs

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.

Null Graph

Trivial Graph

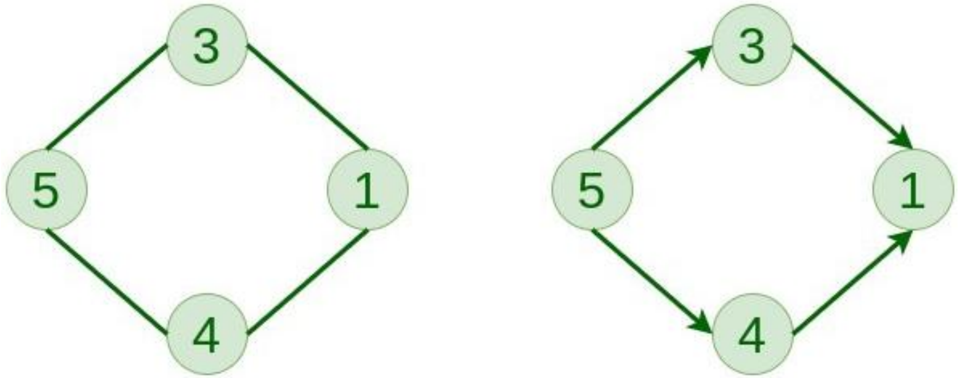
9.2. Types of Graphs

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.



Undirected Graph   Directed Graph

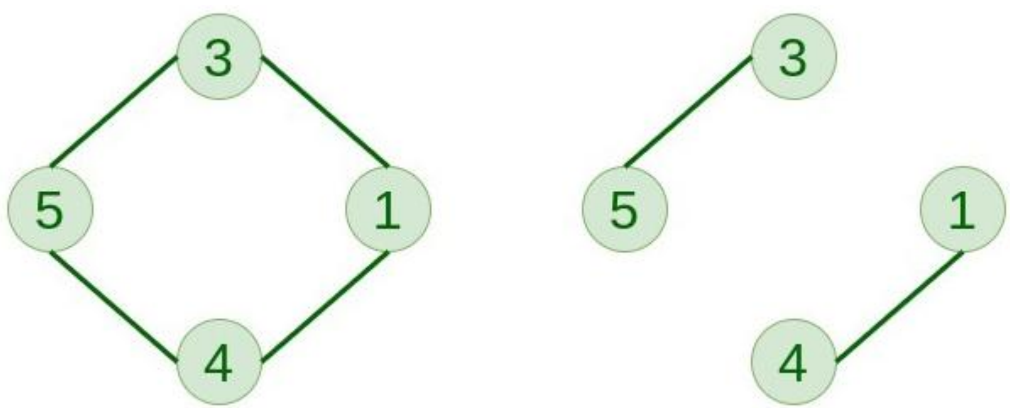
9.2. Types of Graphs

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.



Connected Graph　　Disconnected Graph

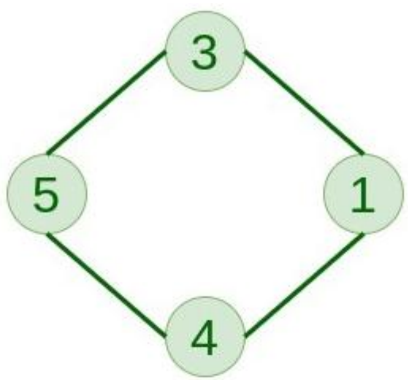
9.2. Types of Graphs

7. Regular Graph

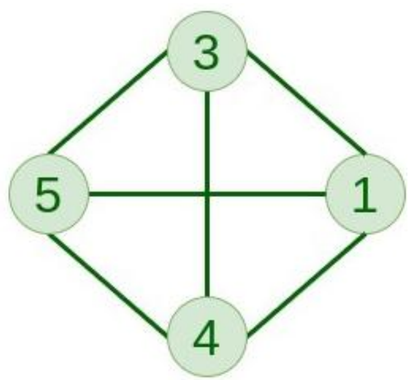
- The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

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



2-Regular



Complete Graph

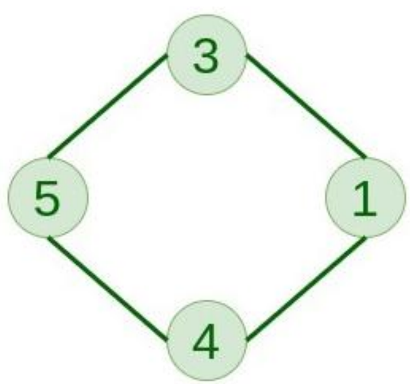
9.2. Types of Graphs

9. Cycle Graph

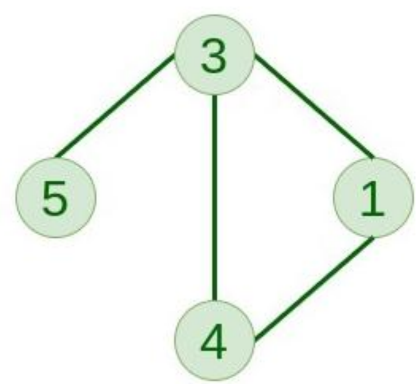
- The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

10. Cyclic Graph

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



Cycle Graph



Cyclic Graph

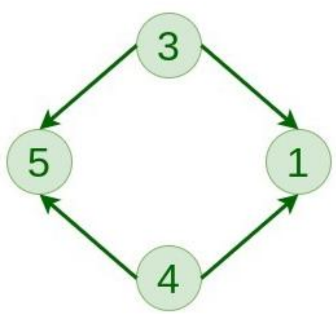
9.2. Types of Graphs

11. Directed Acyclic Graph

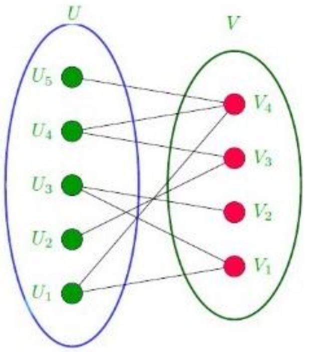
- A Directed Graph that does not contain any cycle is Directed acyclic graph (DAG). They are commonly used to represent *dependencies* between *tasks* in a *project schedule;* no cycles exist.

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. It is by definition that *a graph is bipartite if it does not contain odd length cycles.*



Directed Acyclic Graph



Bipartite Graph

9.2. Types of Graphs

13. Weighted Graph

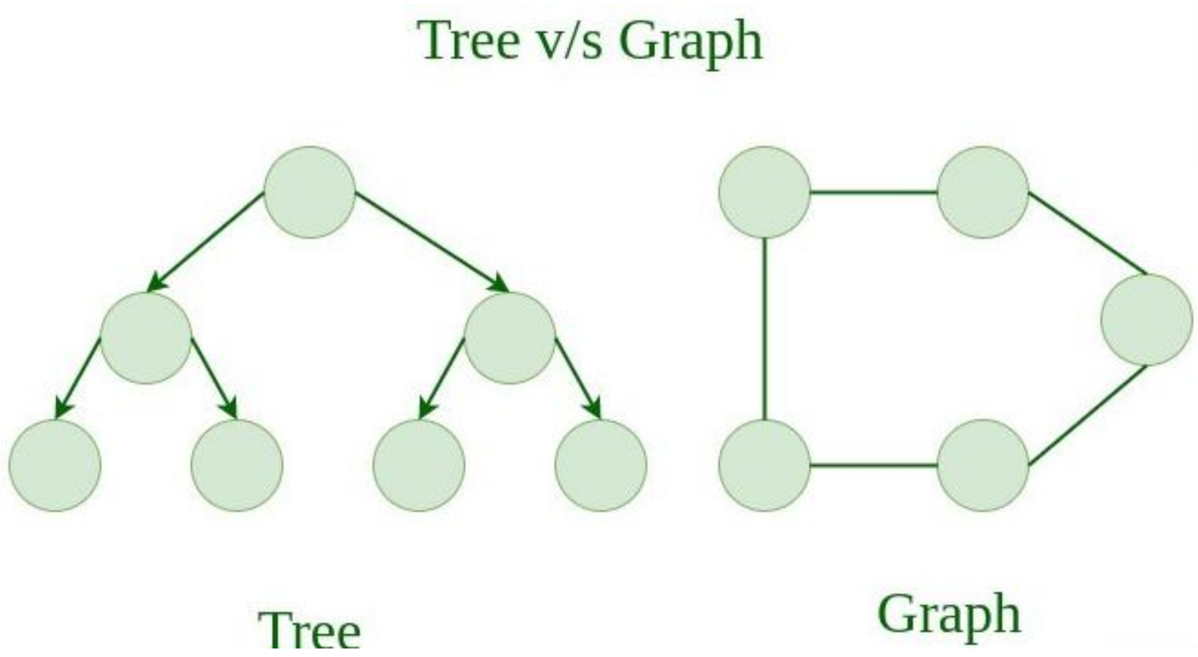
- A graph in which edges are already specified with suitable *weight* or *cost* is known as a *weighted graph*. **Example**: A road network graph where the weights can represent the *distance* between two cities.

14. Unweighted Graph

- A graph in which edges have no *weights* or *costs* associated with them is known as an *unweighted graph*. **Example**: A social network graph where the edges represent *friendships*.

9.2. Types of Graphs

- **Trees** are the restricted types of graphs, just with some more rules. Every tree will *always* be a graph, but *not all* graphs will be trees. *Linked List*, *Trees*, and *Heaps*, all are special cases of graphs.

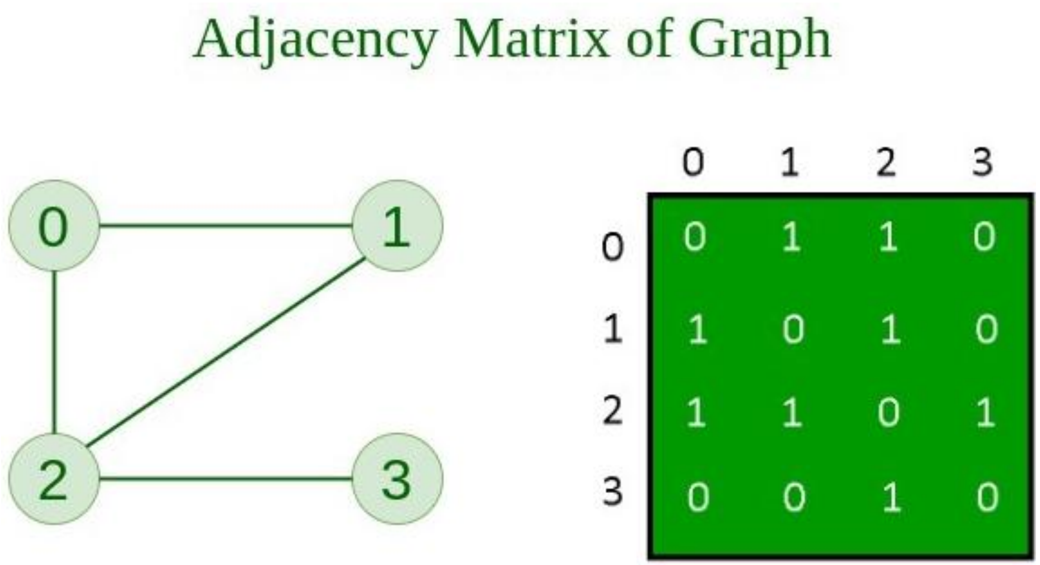


9.2. Types of Graphs

- Consider a given set V = {V1, V2, …, Vn} of ***n*** vertices.
- In an ***undirected graph***, there can be maximum ***n*(n-1)/2*** edges. We can choose to have (or not have) any of the n*(n-1)/2 edges.
- So, the *total number of undirected graphs* (not necessarily connected) that can be constructed out of a given set of ***n*** vertices is ***2^(n*(n-1)/2)***.
- The *maximum number of edges* in an ***acyclic undirected graph*** with ***n*** vertices is: ***n-1***.
- But, *acyclic graph* with the *maximum number of edges* is, actually, a ***spanning tree*** and therefore, the answer is: ***n-1***.

9.3. Representation of Graphs

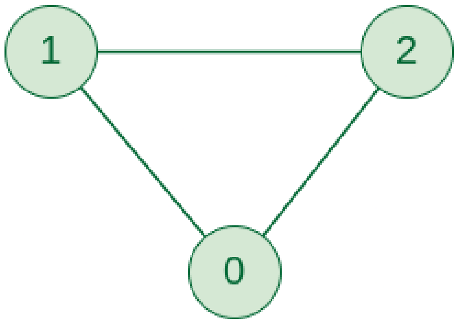
- There are two ways to store a graph:
  - Adjacency Matrix & Adjacency List
- **Adjacency Matrix**
  - In this method, the graph is stored in the form of the 2D matrix where ***rows*** and ***columns*** denote *vertices*. Each entry in the matrix represents the ***weight of the edge*** between those vertices.



9.3. Representation of Graphs

- Let us assume there are **n** vertices in the graph. So, create a matrix **adjMat[n][n]** having dimension n*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:**
  - Initially, the entire Matrix is initialized to **0.** If there is an *edge* from source to destination, we insert **1** to both cases of **adjMat[destination]** because we can go either way.
- **Representation of Directed Graph to Adjacency Matrix:**
  - 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]**.

9.3. Representation of Graphs

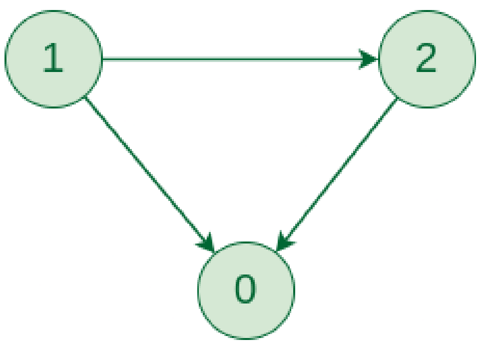


Undirected Graph

	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix



Directed Graph

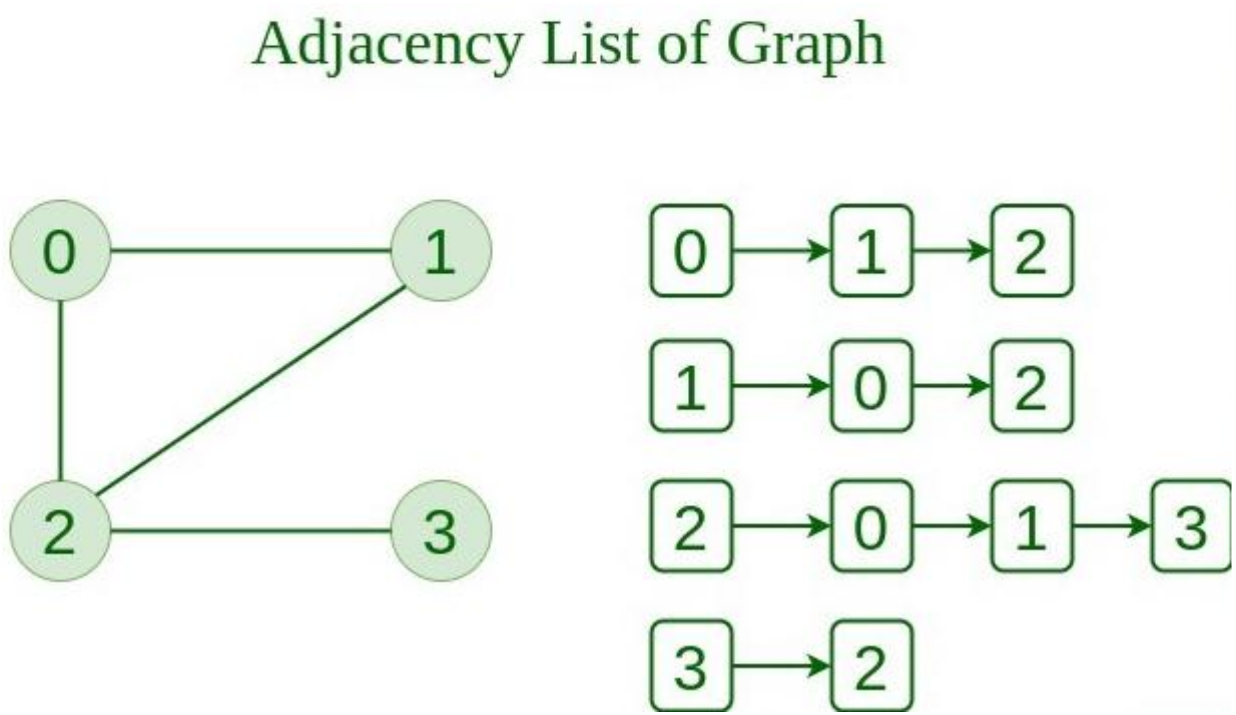
	0	1	2
0			
1	1		1
2	1		

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

9.3. Representation of Graphs

- **Adjacency List**
  - 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.



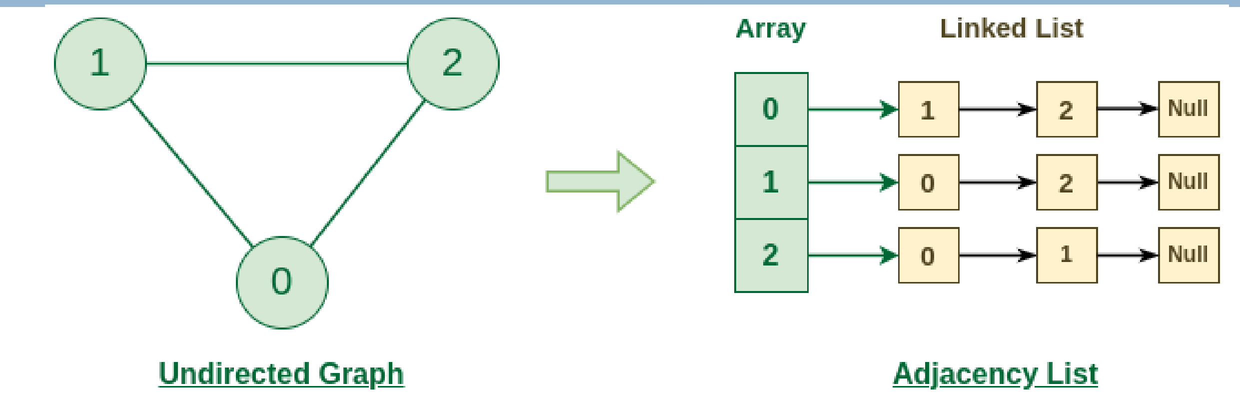
9.3. Representation of Graphs

- An array of Lists is used to store *edges* between two *vertices*. The size of array is equal to the number of **vertices 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 us 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.

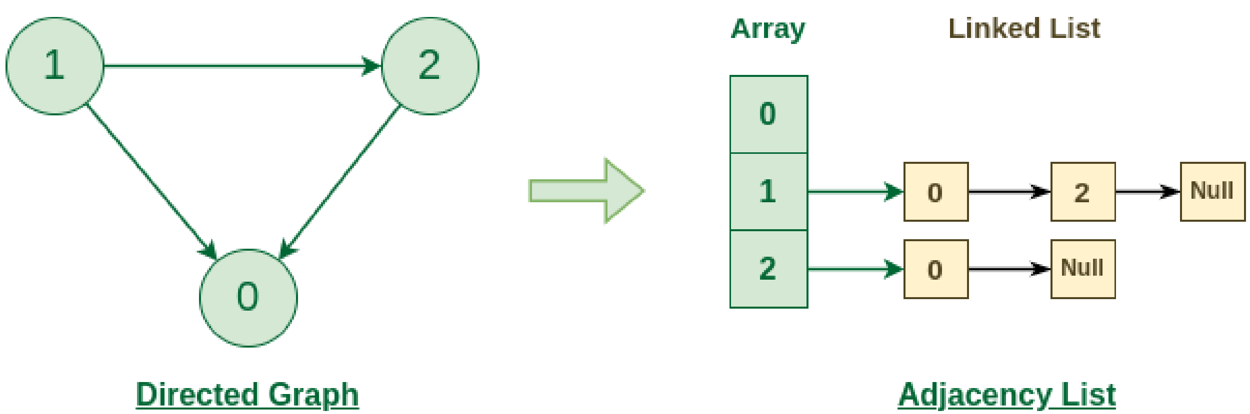
9.3. Representation of Graphs

- **Representation of Undirected Graph to Adjacency list:**
  - For the undirected graph with 3 vertices, an array of list will be created of size 3, where each indices represent the vertices. Vertex 0 has neighbours: 1 and 2. So, insert vertex 1 and 2 at indices 0 of array. Similarly, vertex 1 has two neighbours: 0 and 2. So, insert vertex 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.
- **Representation of Directed Graph to Adjacency list:**
  - For the directed graph with 3 vertices, an array of list will be created of size 3, where each indices represent the vertices. Vertex 0 has no neighbours. Vertex 1 has two neighbours: 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.

9.3. Representation of Graphs



Graph Representation of Undirected graph to Adjacency List



Graph Representation of Directed graph to Adjacency List

9.3. Representation of Graphs

- **Adjacency Matrix** compared to **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. In algorithms such as: Prim's and Dijkstra, adjacency matrix is used to have less complexity.

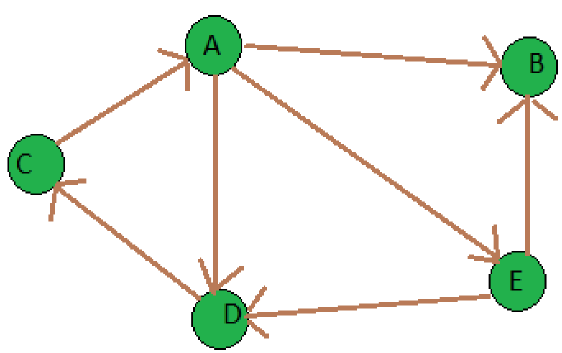
Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

9.3. Representation of Graphs

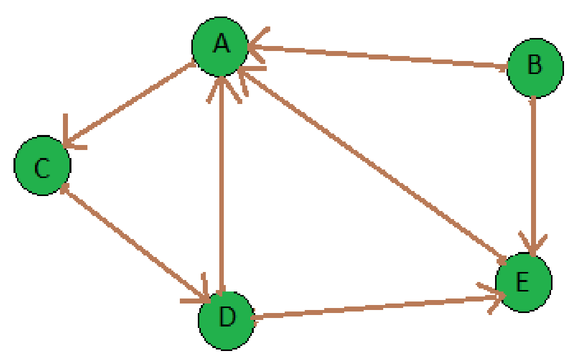
- **Distance** between two Vertices:
  - The number of edges that are available in the *shortest path* between vertex A and vertex B is basically the *Distance* between A and B.
  - Notation used: **d(A, B)**
- The **Eccentricity** of a Vertex:
  - The maximum distance from a vertex V to all other vertices is considered as the *Eccentricity* of that vertex.
  - Notation used: **e(V)**

9.3. Representation of Graphs

- **Transpose** of a *directed graph* G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G.
- That is, if G contains an edge **(u,v)**, then the converse/transpose/reverse of G contains an edge **(v,u)**, and vice versa.



(i)



(ii)

9.4. Applications of Graphs

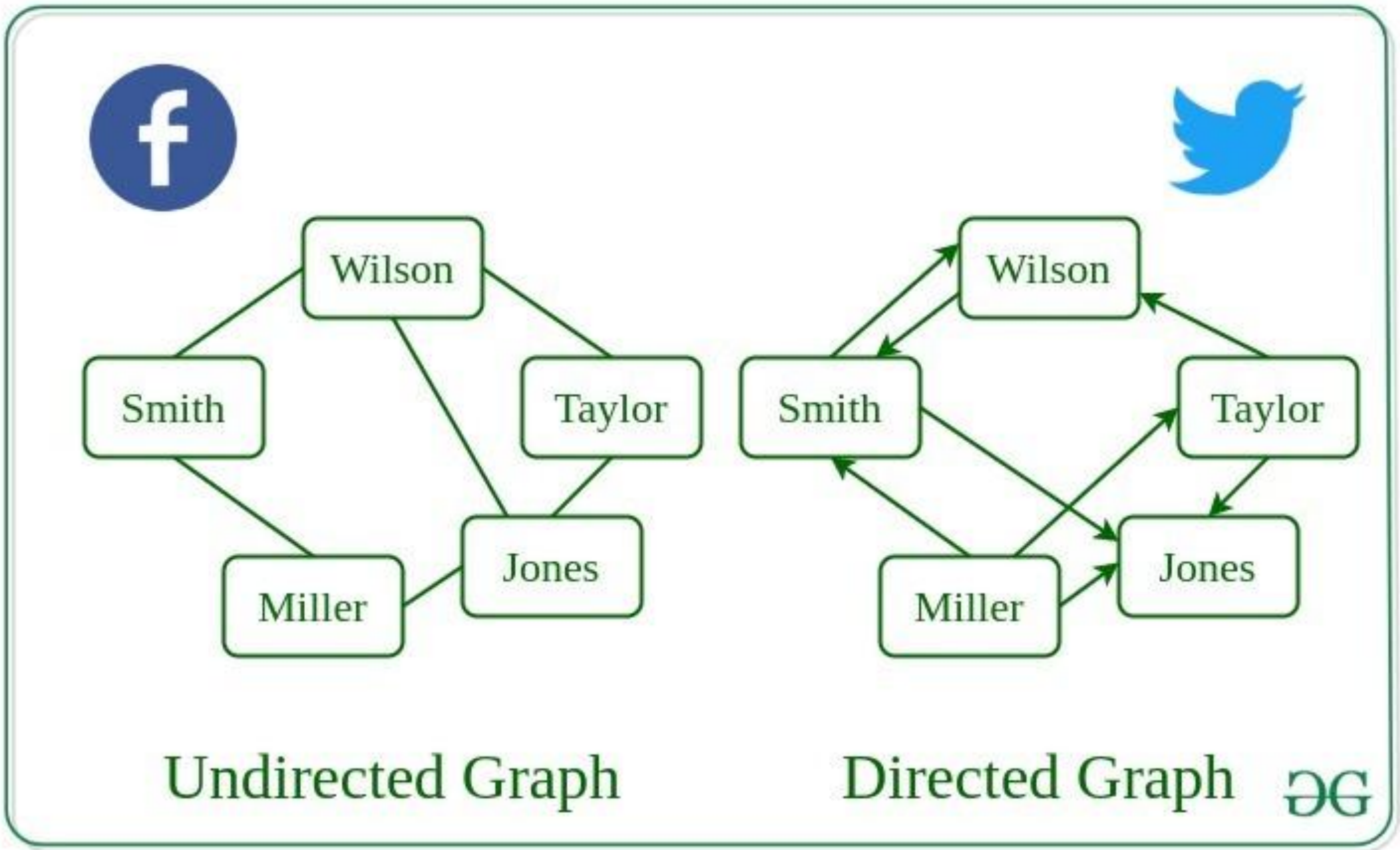
- **Basic Operations on Graphs**
  - Insertion of Nodes/Edges in the graph – Insert a node into the graph.
  - Deletion of Nodes/Edges in the graph – Delete a node from the graph.
  - Searching on Graphs – Search an entity in the graph.
  - Traversal of Graphs – Traversing all the nodes in the graph.

9.4. Applications of Graphs

- **Usage of Graphs**
  - Maps can be represented using graphs and then can be used by computers to provide various services, like the *shortest path* between two cities.
  - When various tasks depend on each other then this situation can be represented using a Directed Acyclic graph, and we can find the order in which tasks can be performed using topological sort.
  - State Transition Diagram represents what can be the legal moves from current states. In game of tic-tac-toe this can be used.

9.4. Applications of Graphs

- Real-Life Applications of Graph:



9.4. Applications of Graphs

- Graph data structures can be used to represent the **interactions** between players on a team, such as: *passes*, *shots*, and *tackles*. Analyzing these interactions can provide insights into team dynamics and areas for improvement.
- Commonly used to represent **social networks**, such as: networks of friends on social media.
- Graphs can be used to represent the **topology of computer networks**, such as: the connections between routers and switches.
- Graphs are used to represent the **connections** between different places in a transportation network, such as: roads and airports.

9.4. Applications of Graphs

- **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:** Graphs are used *extensively* in compilers. They can be used for type inference, data flow analysis, register allocation, etc. 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.

9.4. Applications of Graphs

- **When to use Graphs:**
  - When you need to represent and analyze the relationships between different objects or entities.
  - When you need to perform network analysis.
  - When you need to identify key players, influencers or bottlenecks in a system.
  - When you need to make predictions or recommendations.
  - *Modeling networks*: Graphs are commonly used to model various types of networks, such as social networks, transportation networks, and computer networks. In these cases, vertices represent ***nodes*** in the network, and edges represent the ***connections*** between them.

9.4. Applications of Graphs

- *Finding paths*: Graphs are often used in algorithms for finding paths between two vertices in a graph, such as: ***shortest path algorithms.*** For example, graphs can be used to find the fastest route between two cities on a map or the most efficient way to travel between multiple destinations.
- *Representing data relationships*: Graphs can be used to represent relationships between data objects, such as in a database or data structure. In these cases, vertices represent ***data objects***, and edges represent the ***relationships*** between them.
- *Analyzing data*: Graphs can be used to analyze and visualize complex data, such as: in data clustering algorithms or machine learning models. In these cases, vertices represent ***data points***, and edges represent the ***similarities or differences*** between them.

9.4. Applications of Graphs

- **Advantages of Graphs:**
  - Graphs are a versatile data structure that can be used to represent a wide range of relationships and data structures.
  - They can be used to model and solve a wide range of problems, including: path finding, data clustering, network analysis, and machine learning.
  - Graph algorithms are often very efficient and can be used to solve complex problems quickly and effectively.
  - Graphs can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

9.4. Applications of Graphs

- **Disdvantages of Graphs:**
  - Graphs 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.
  - Graphs 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.

9.4. Applications of Graphs

- **Summary:**
  - Graph data structures are a powerful tool for representing and analyzing *relationships* between objects or entities.
  - Graphs can be used to represent the *interactions* between different objects or entities, and then analyze these interactions to *identify* patterns, clusters, communities, key players, influencers, bottlenecks and anomalies.
  - In sports data science, graph data structures can be used to analyze and understand the *dynamics* of team performance and player *interactions* on the field. They can be used in a variety of fields, such as: Sports, Social media, transportation, cybersecurity and many more.