GRAPHS: CONCEPTS AND ALGORITHMS

AN OVERVIEW OF KEY GRAPH CONCEPTS AND ALGORITHMS

AGENDA

Graphs: The Graph ADT, Data Structures for Graphs: Edge List, Adjacency List -Adjacency

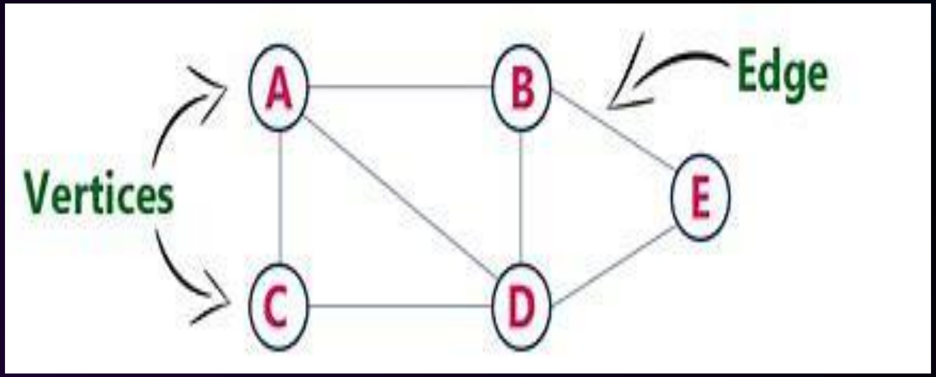
Map - Adjacency Matrix structure, Directed Acyclic graph: Topological Sorting,

Shortest Path Algorithm: All Pairs Shortest Paths: Floyd-Warshall_s Algorithm

GRAPH :

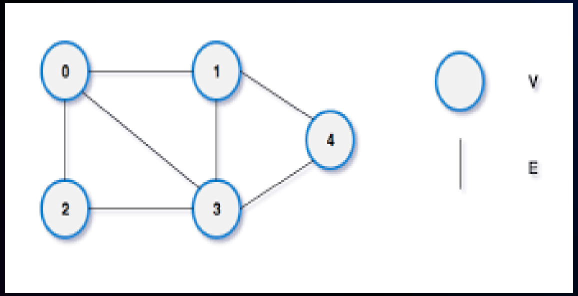
1. A graph is a non-linear data structure consisting of a set of vertices (or nodes) and a set of edges that connect pairs of vertices. Graphs are used to represent connections or relationships between objects.

2. Each node contains an element
3. Each edge connects two nodes together

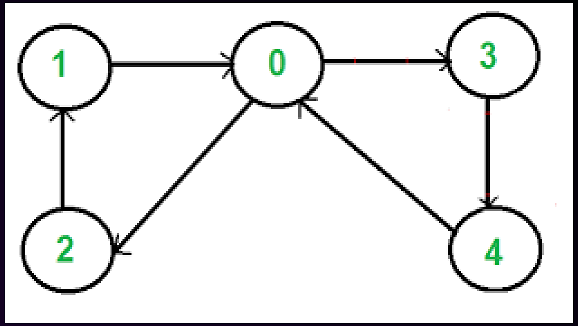


TYPES OF GRAPHS :

- An **undirected graph** is one in which the edges do not have a direction
- 'graph' denotes undirected graph
- Here (v1,v2) in E is un-ordered

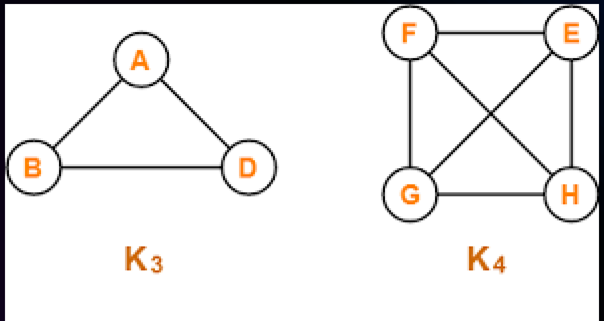


- **Directed graph** is one in which the edges have a direction
- Also called as 'digraph'
- Here (v1, v2) in E is ordered.



COMPLETE GRAPH :

- A complete graph is a graph that has the maximum number of edges
- For **undirected graph** with "n" vertices(nodes), the maximum numb er of edges is " n(n-1)/2 "
- For **directed graph** with n vertices, the maximum number of edges is " n(n-1) "



Examples of Complete Graph

CYCLIC GRAPH :

- **Definition**: A cyclic graph, also known as a cyclic or non-acyclic graph, is a graph that contains at least one cycle. A cycle in a graph is a path where the first and last vertices are the same, and no vertex appears more than once in the path.

- **Characteristics**:
  - Contains at least one cycle.
  - Paths can loop back on themselves.
  - Can have multiple cycles.

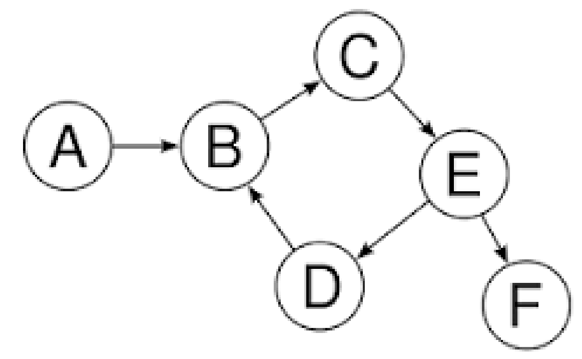
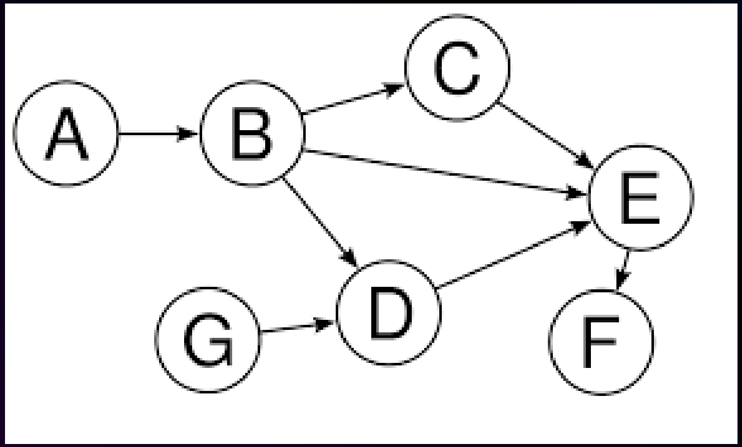


Figure 5 : Cyclic Graph

ACYCLIC GRAPH :

- **Definition**: An acyclic graph, also known as a cycle-free or directed acyclic graph (DAG), is a graph that does not contain any cycles. In other words, it is impossible to traverse from a vertex back to itself by following the edges of the graph.

- **Characteristics**:
  - Does not contain any cycles.
  - Paths do not loop back on themselves.
  - Directed acyclic graphs (DAGs) are particularly common and useful in various applications.



GRAPH ADT :

- The Graph Abstract Data Type (ADT) defines the basic operations and properties associated with a graph. The Graph ADT abstracts away the implementation details of how the graph is represented and focuses on the fundamental operations that can be performed on the graph.
- Operations:
- AddVertex(vertex): Adds a new vertex to the graph.
- AddEdge(vertex1, vertex2): Adds an edge between the specified vertices. For a directed graph, the edge goes from vertex1 to vertex2.
- RemoveVertex(vertex): Removes the specified vertex from the graph, along with any edges connected to it.
- RemoveEdge(vertex1, vertex2): Removes the edge between the specified vertices.
- GetNeighbors(vertex): Returns a list of neighboring vertices connected to the specified vertex.
- GetVertices(): Returns a list of all vertices in the graph.

EDGE LIST :

- The Edge List is a simple and straightforward way to represent a graph.
- Each edge of the graph is represented as an object.
- The object typically contains information about the source vertex, destination vertex, and any associated weight (if the graph is weighted).
- An array or list of edge objects is used to store all the edges of the graph.
- Advantages of Edge List Representation:
- Simplicity: Edge List is straightforward to implement and understand.
- Space Efficiency: Edge List typically requires less memory compared to other representations for sparse graphs.
- Flexibility: Edge List allows easy addition and removal of edges from the graph.

ADJACENCY LIST -ADJACENCY :

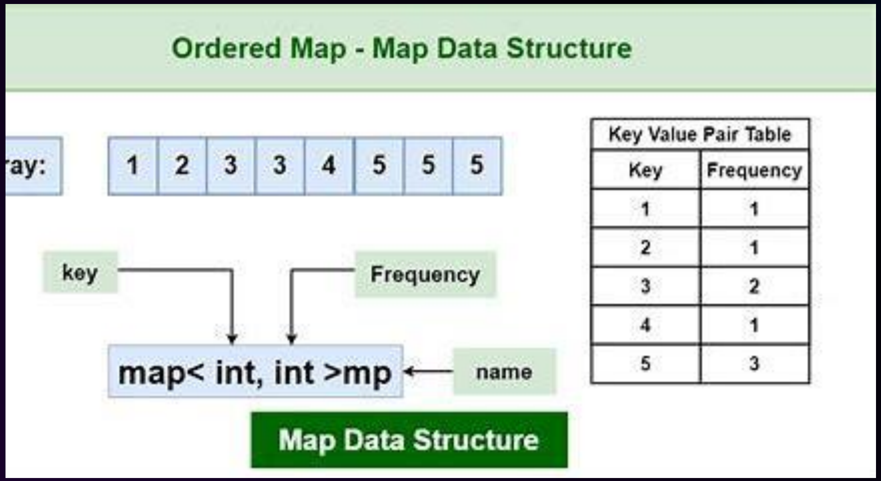
- The Adjacency List representation is a popular way to represent graphs. This representation is particularly efficient for sparse graphs where the number of edges is much smaller than the maximum possible number of edges.

- Adjacency List Representation:
- Each vertex of the graph is associated with a list (or array) containing the vertices it is adjacent to.
- For a directed graph, each vertex maintains two lists: one for outgoing edges (adjacent vertices) and another for incoming edges.
- For an undirected graph, each vertex maintains a single list containing its adjacent vertices.
- The adjacency lists can be implemented using arrays, linked lists, or other suitable data structures.

MAP :

1. Map data structure (also known as a dictionary, associative array, or hash map) is defined as a data structure that stores a collection of key-value pairs, where each key is associated with a single value.

2. Maps are widely used in many applications :
3. database indexing
4. network routing
5. web programming

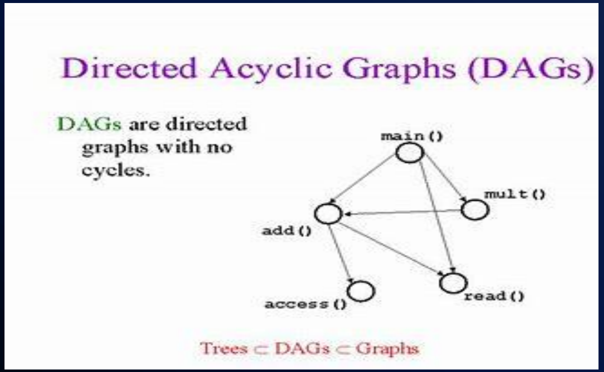


Key Value Pair Table

Key	Frequency
1	1
2	1
3	2
4	1
5	3

MAP - ADJACENCY MATRIX STRUCTURE :

- The Adjacency Matrix is a commonly used way to represent graphs, especially for dense graphs where the number of edges is close to the maximum possible number of edges. In this representation, a 2D matrix is used to store whether there is an edge between each pair of vertices.
- **Adjacency Matrix Representation:**
- In the Adjacency Matrix representation:
  - A 2D matrix of size $V \times V$ (where V is the number of vertices in the graph) is used.
  - Each row and column of the matrix represents a vertex in the graph.
  - If there is an edge from vertex i to vertex j, then the cell at row i and column j is set to 1 (for an unweighted graph) or contains the weight of the edge (for a weighted graph).
  - If there is no edge from vertex i to vertex j, then the cell at row i and column j is set to 0 or some special value (for unweighted and weighted graphs, respectively).
  - For undirected graphs, the adjacency matrix is symmetric (i.e. $matrix[i][j]=matrix[j][i]$).


Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no cycles.

Trees ⊂ DAGs ⊂ Graphs

DIRECTED ACYCLIC GRAPH:

- A Directed Acyclic Graph (DAG) is a special type of graph that contains directed edges and does not contain any cycles. In other words, it is a directed graph with no directed cycles.

- **Characteristics of Directed Acyclic Graphs:**
- **Directed Edges**: In a DAG, edges have a direction, meaning they point from one vertex to another. The direction indicates a one-way relationship between vertices.
- **No Cycles**: Unlike general directed graphs, a DAG does not contain any directed cycles. A directed cycle is a sequence of vertices in which there is a directed edge from each vertex to the next, and the last vertex points back to the first vertex.
- **Directed Acyclic**: The term "acyclic" indicates the absence of cycles. Since a DAG has no directed cycles, it is acyclic.

DIRECTED ACYCLIC GRAPH: TOPOLOGICAL SORTING :

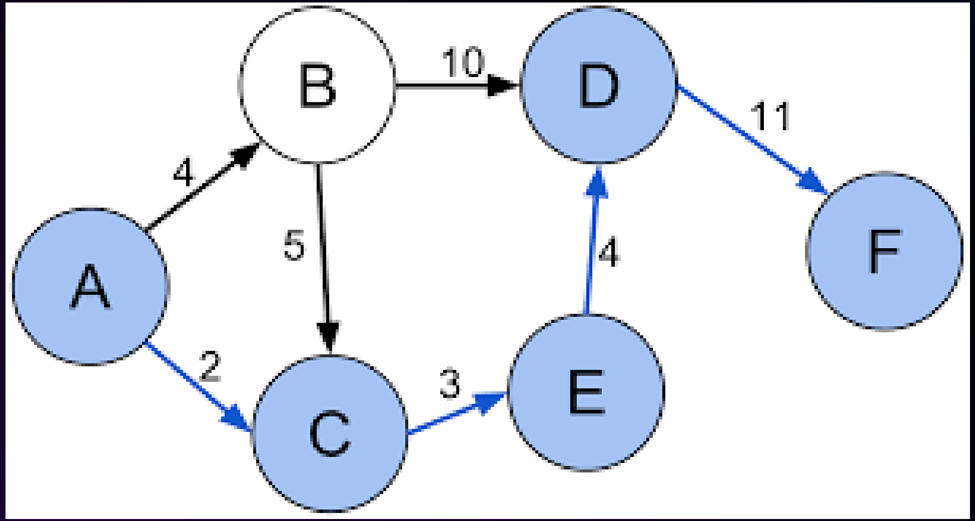
- A Directed Acyclic Graph (DAG) is a graph that contains directed edges and does not contain any cycles. Topological sorting is a special ordering of the vertices of a directed graph such that for every directed edge $u \rightarrow v$, vertex u comes before vertex v in the ordering. Topological sorting is only possible for DAGs because cycles create dependencies that make it impossible to define a linear ordering.

- **Topological Sorting Algorithm:**
- Perform a depth-first search (DFS) traversal of the graph.
- During the DFS traversal, when a vertex has no outgoing unvisited edges, mark it as visited and add it to the topological ordering.
- Continue this process until all vertices are visited.

SHORTEST PATH ALGORITHM :

- Shortest path algorithms are used to find the shortest path between two vertices in a graph. There are several algorithms designed for this purpose, each with its own characteristics and use cases. Here are some of the most commonly used shortest path algorithms:

- **Dijkstra's Algorithm**:
- **Bellman-Ford Algorithm**:
- **Floyd-Warshall Algorithm**:
- ***A* Algorithm****: A* (A-star) algorithm
- **Bidirectional Search**:



SHORTEST PATH ALGORITHM :

- **Dijkstra's Algorithm:** Dijkstra's algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It is based on the principle of greedy selection, iteratively selecting the vertex with the shortest distance from the source and updating the distances to its neighboring vertices.
- **Bellman-Ford Algorithm**: Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph, even if the graph contains negative edge weights or negative cycles. It iterates over all edges multiple times, relaxing them to update the shortest distance estimates.
- **Floyd-Warshall Algorithm:** Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights. It iteratively considers all pairs of vertices and calculates the shortest path lengths using dynamic programming.
- *A Algorithm**: A* (A-star) algorithm is a heuristic search algorithm used for finding the shortest path from a start vertex to a goal vertex in a weighted graph. It uses a heuristic function to estimate the cost of reaching the goal from each vertex and combines it with the actual cost of reaching the vertex to make informed decisions during the search.
- **Bidirectional Search**: Bidirectional search is a technique used to find the shortest path between two vertices by simultaneously performing two breadth-first searches, one from the source vertex and one from the destination vertex. It terminates when the searches meet in the middle, effectively reducing the search space.

FLOYD-WARSHALLS ALGORITHM:

- **Floyd-Warshall Algorithm:**
- **Initialization**: Initialize a 2D array *dist* of size $V \times V$, where V is the number of vertices in the graph. Set *dist*[*i*][*j*] to the weight of the edge from vertex *i* to vertex *j* if the edge exists, or ∞∞ if there is no direct edge. Set *dist*[*i*][*i*] to 0 for all *i* (distance from a vertex to itself is 0).
- **Dynamic Programming Step**: Iterate over all vertices *k* from 1 to V (intermediate vertices). For each pair of vertices *i* and *j*, update *dist*[*i*][*j*] to the minimum of:
  - The current value of *dist*[*i*][*j*].
  - The sum of the distances from vertex *i* to vertex *k* and from vertex *k* to vertex *j*, i.e., *dist*[*i*][*k*]+*dist*[*k*][*j*].
- **Optimization**: To reconstruct the shortest paths, keep track of the predecessors for each pair of vertices during the dynamic programming step. This can be achieved by storing an additional 2D array that keeps track of the predecessor vertex for each pair of vertices.
- **Termination**: After V iterations of the dynamic programming step, the *dist* array will contain the shortest paths between all pairs of vertices.

THANK YOU