Graphs in Data Structure

A graph is a non-linear data structure that represents a set of objects (vertices) and the connections between them (edges).

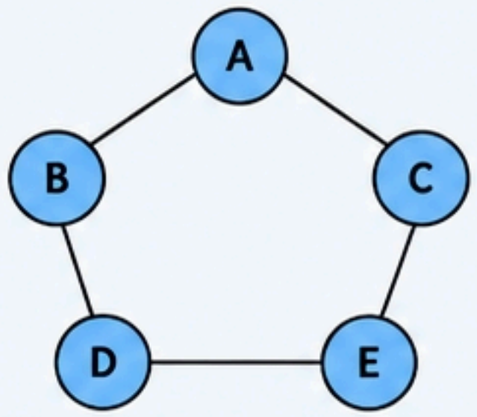
1. Definition

A graph **G** is a pair (**V**, **E**) where:

- **V** is a finite set of vertices (or nodes).
- **E** is a set of edges, where each edge connects two vertices.

Graphs are used to model real-world problems like networks, social connections, routing, scheduling, etc.

Example of a Graph



Vertices (V) = {A, B, C, D, E}

Edges (E) = {AB, AC, BD, CE, DE}

2. Terminologies

Vertex (Node)
The basic element of a graph.
(e.g., A, B, C, D, E)

Edge
A connection between two vertices.
(e.g., AB, AC, BD)

Adjacent Vertices
Vertices that are connected by an edge.
(e.g., A and B are adjacent)

Degree of a Vertex
Number of edges incident on a vertex.
(e.g., deg(B) = 3)

Path
A sequence of vertices connected by edges.
(e.g., A → B → D)

Cycle
A path that starts and ends at the same vertex.
(e.g., A → B → C → A)

Connected Graph
There is a path between every pair of vertices.

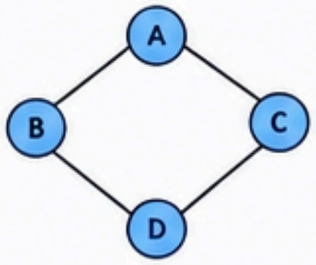
Disconnected Graph
At least one pair of vertices is not connected.

Weighted Graph
Edges have weights (cost, distance, etc.).

Unweighted Graph
Edges have no weights (only show connections).

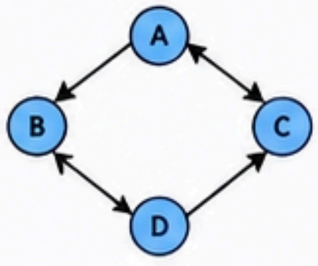
3. Types of Graphs

1. Undirected Graph



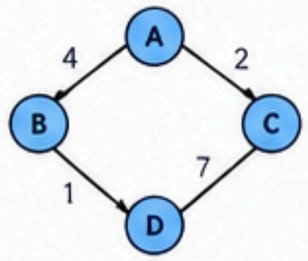
- Edges have no direction.
- (A, B) is same as (B, A).
- Example: Social network.

2. Directed Graph (Digraph)



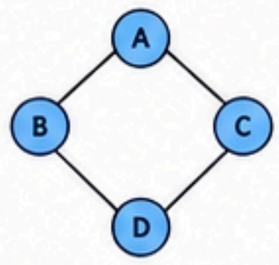
- Edges have direction.
- (A, B) is not same as (B, A).
- Example: Web pages, one-way roads.

3. Weighted Graph



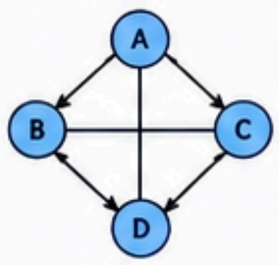
- Each edge has a weight (cost, distance, etc.).
- Example: Shortest path problem.

4. Unweighted Graph



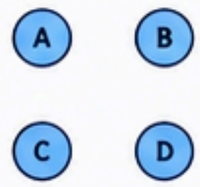
- All edges are equal (weight = 1).
- Example: Friends network.

5. Complete Graph



- Every pair of vertices is connected.
- Example: K_4 (complete graph with 4 vertices).

6. Null Graph



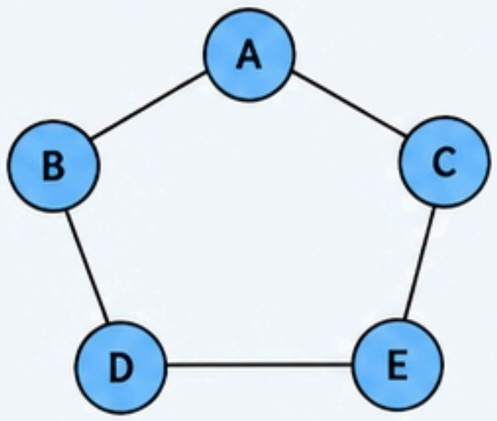
- No edges.
- Example: Set of independent vertices.

Representation of Graph in Memory

A graph can be stored in memory using different data structures. The most common methods are: **Adjacency Matrix, Adjacency List and Edge List.**

1. Example Graph

Consider the following undirected, unweighted graph with 5 vertices (A, B, C, D, E):



Vertices: {A, B, C, D, E}

Edges: {AB, AC, BD, CE, DE}

(Undirected and unweighted)

2. Adjacency Matrix Representation

An **adjacency matrix** is a 2D array of size V × V (V = number of vertices). If there is an edge between vertices i and j, the value is 1 (or the weight), otherwise 0.

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	0
C	1	0	0	0	1
D	0	1	0	0	1
E	0	0	1	1	0

How it works:

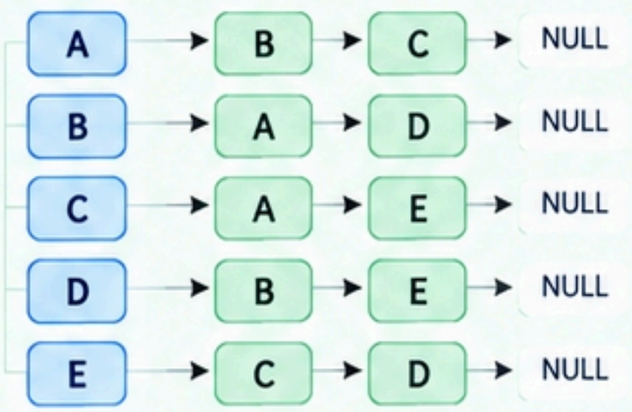
- Row i, Column j = 1 if edge (i, j) exists.
- For undirected graphs, matrix is symmetric (A[i][j] = A[j][i]).
- Space used: $O(V^2)$.

3. Adjacency List Representation

An **adjacency list** uses an array of linked lists (or arrays) where each index represents a vertex and stores the list of its adjacent vertices.

Vertex	Adjacent Vertices
A	B, C
B	A, D
C	A, E
D	B, E
E	C, D

(As linked lists)



Space used: O(V + E) (more efficient for sparse graphs).

4. Edge List Representation

An **edge list** stores all the edges of the graph in a simple list of pairs (or triples for weighted graphs).

Index	Vertex 1	Vertex 2
0	A	B
1	A	C
2	B	D
3	C	E
4	D	E

For weighted graphs (3-tuple):

Index	Vertex 1	Vertex 2	Weight
0	A	B	1
1	A	C	1
2	B	D	1
3	C	E	1
4	D	E	1

Space used: O(E) (best for very sparse graphs).

5. Comparison of Representations

Representation	Memory Used	Best For	Advantages	Disadvantages
Adjacency Matrix	$O(V^2)$	Dense graphs	• Easy to check edge existence • Simple implementation	• Wastes space for sparse graphs • Higher memory usage
Adjacency List	O(V + E)	Sparse graphs	• Memory efficient • Easy to traverse neighbors	• Slightly complex to check edge existence • Not as fast for dense graphs
Edge List	O(E)	Very sparse graphs / algorithms (e.g., Kruskal's MST)	• Compact representation • Useful for edge-based algorithms	• Slower to check adjacency • Not suitable for frequent neighbor access

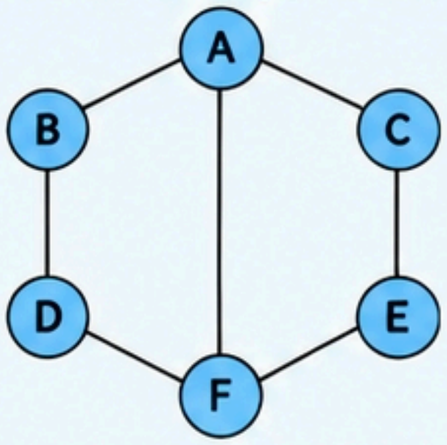
Key Takeaway: The choice of representation depends on the graph's density, size and the operations you need to perform.

Graph Traversal

Graph traversal means visiting (exploring) all the vertices of a graph in a systematic way.
The two most common traversal techniques are: **Depth First Search (DFS)** and **Breadth First Search (BFS)**.

1. Example Graph

Consider the following undirected graph with 6 vertices (starting vertex = A):



Vertices (V) = {A, B, C, D, E, F}

Edges (E) = {AB, AC, AD, BC, BF, CF, DE, EF}

(Undirected graph – edges are bi-directional)

2. Depth First Search (DFS)

DFS explores as far as possible along a branch before backtracking.

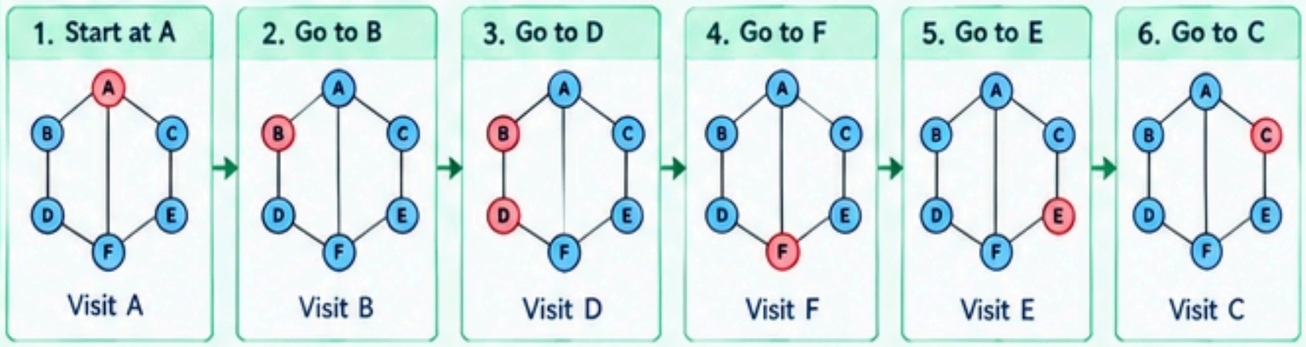
- Uses a **stack** (or recursion).
- Follows one path until it reaches a dead end, then backtracks.
- Can be implemented using recursion or an explicit stack.

Traversal Order (starting from A):

A → B → D → F → E → C

(One possible order; depends on the order in which adjacent vertices are chosen.)

DFS Steps (recursive approach)



3. Breadth First Search (BFS)

BFS explores all the neighbors of a vertex first, then moves to the next level.

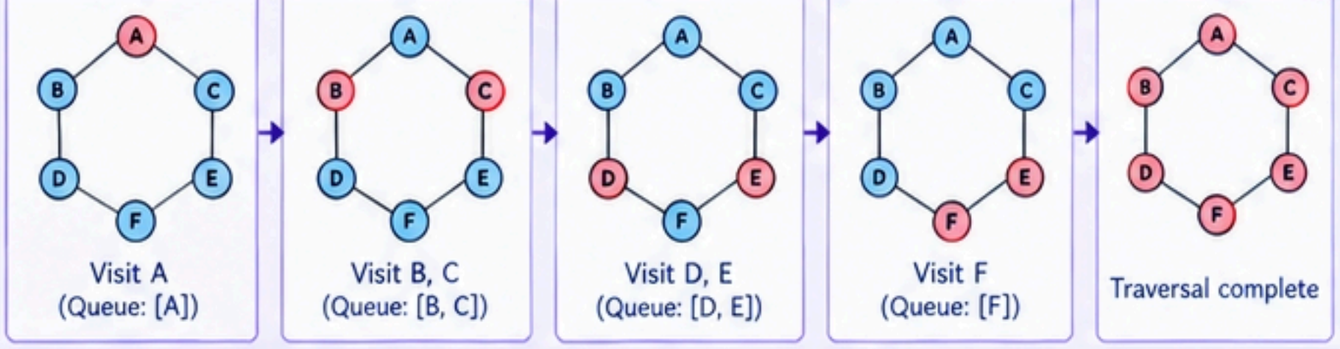
- Uses a queue.
- Visits vertices level by level (layer by layer).
- Finds the shortest path in an unweighted graph.

Traversal Order (starting from A):

A → B → C → D → E → F

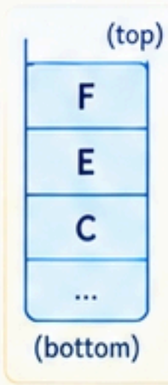
(Visits all level-1 neighbors first, then level-2, and so on.)

BFS Steps (queue based)



4. Working of Traversal Using Data Structures

DFS – Stack (or Recursion)



1. Push start vertex (A).
2. Visit and push unvisited adjacent vertex.
3. If no unvisited adjacent vertex, pop from stack (backtrack).

In recursion, the system call stack works like a stack.

BFS – Queue

1. Enqueue start vertex (A).
2. Dequeue a vertex, visit it.
3. Enqueue all unvisited adjacent vertices.

Queue ensures level order traversal.

5. Comparison: DFS vs BFS

Feature	Depth First Search (DFS)	Breadth First Search (BFS)
Traversal Order	Goes deep, then backtracks	Goes level by level
Data Structure Used	Stack (or recursion)	Queue
Shortest Path (unweighted)	Not guaranteed	Guaranteed
Time Complexity	O(V + E)	O(V + E)
Space Complexity	O(V) (worst case)	O(V) (worst case)
Best Use Cases	Topological sorting, cycle detection, connected components	Shortest path, level order traversal, network routing, etc.

Key Takeaways

- DFS and BFS are graph traversal algorithms.
- DFS uses a stack (or recursion) and explores deeply.
- BFS uses a queue and explores level by level.
- Both have time complexity O(V + E).
- Choice depends on the problem (e.g., shortest path → BFS, topological sorting → DFS).

Same graph • Different paths • Same goal – Visit all vertices →

Dijkstra's Algorithm

Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

1. Problem & Terminology

Given:

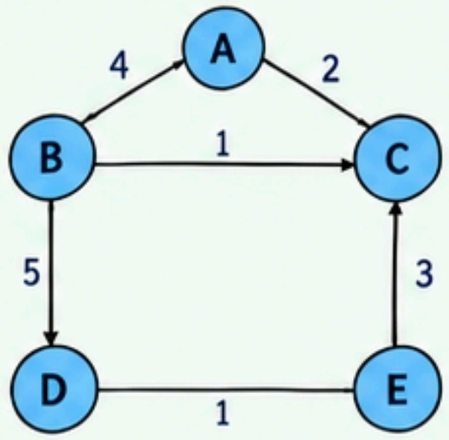
- A weighted graph with non-negative edge weights.
- A source vertex (starting point).
- Find the shortest path from source to all other vertices (or to a specific destination).

Key Terms:

- **Source:** Starting vertex.
- **Distance:** Shortest known distance from source.
- **Visited / Settled:** Vertex whose shortest distance is finalized.
- **Priority Queue:** Stores vertices with minimum distance (min-heap).

2. Example Graph

Find shortest paths from source vertex A to all other vertices.



Graph Representation (Adjacency List):

A : (B,4), (C,2)
B : (A,4), (C,1), (D,5)
C : (A,2), (B,1), (E,3)
D : (B,5), (E,1)
E : (C,3), (D,1)

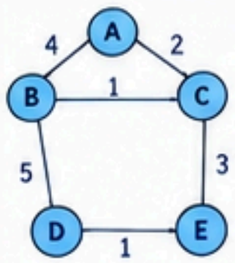
All edge weights are non-negative.

3. Algorithm Steps

1. **Initialize:** Set distance of source = 0, all other vertices = ∞. Put source in priority queue.
2. Extract the vertex with minimum distance from the queue.
3. For each neighbor of the extracted vertex, relax the edge (update distance if shorter).
4. Mark the vertex as visited (settled).
5. Repeat steps 2–4 until all vertices are visited (or queue is empty).
6. The distance array gives shortest distances. Use parent array to reconstruct the path.

4. Step-by-Step Execution (Source = A)

Step 0 – Initialization

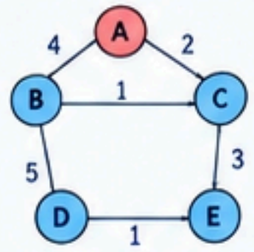


Vertex	Distance	Parent
A	0	–
B	∞	–
C	∞	–
D	∞	–
E	∞	–

Priority Queue: [(A, 0)]

Start from source A. Set A's distance to 0, others to ∞.

Step 1 – Visit A

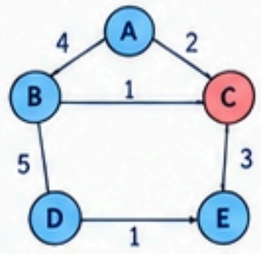


Vertex	Distance	Parent
A	0	–
B	4	A
C	∞	A
D	∞	–
E	∞	–

Priority Queue: [(C, 2), (B, 4)]

Relax edges from A:
B = 4, C = 2

Step 2 – Visit C

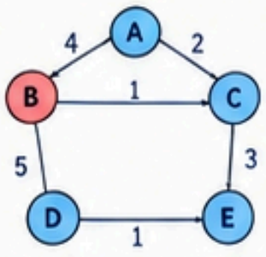


Vertex	Distance	Parent
A	0	–
B	3	C
C	2	A
D	∞	–
E	5	C

Priority Queue: [(C, 3), (E, 5), (D, 5)]

Relax edges from C:
B = 3 (better), E = 5

Step 3 – Visit B

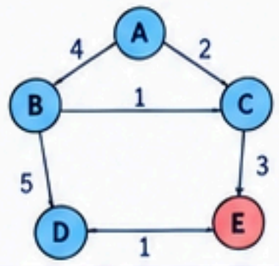


Vertex	Distance	Parent
A	0	–
B	3	C
C	2	A
D	8	B
E	5	C

Priority Queue: [(E, 5), (D, 8)]

Relax edges from B:
D = 8 (5 + 5)

Step 4 – Visit E

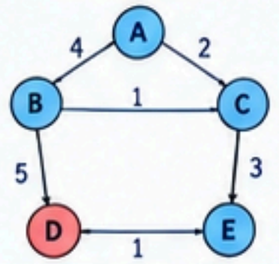


Vertex	Distance	Parent
A	0	–
B	3	C
C	2	A
D	6	E
E	5	C

Priority Queue: [(D, 6)]

Relax edges from E:
D = 6 (5 + 1)

Step 5 – Visit D



Vertex	Distance	Parent
A	0	–
B	3	C
C	2	A
D	6	E
E	5	C

Priority Queue: [] (empty)

All vertices visited.
Algorithm complete.

5. Final Result

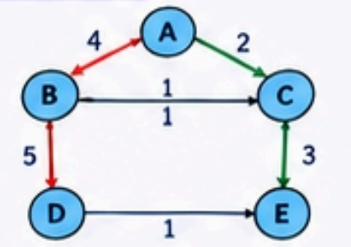
Shortest Distances from A

Vertex	Distance
A	0
B	3
C	2
D	6
E	5

Shortest Paths (using parent info)

A → B	A → C → B	(cost 3)
A → C	A → C	(cost 2)
A → D	A → C → E → D	(cost 6)
A → E	A → C → E	(cost 5)

Shortest Path Tree



= not in shortest path tree
= in shortest path tree

6. Complexity & Properties

Item	Details
Time Complexity	O((V + E) log V) using a min-heap (priority queue).
Space Complexity	O(V + E)
Works for	Graphs with non-negative weights only.
Does Not Work For	Graphs with negative edge weights (uses Bellman-Ford instead).
Greedy Approach	Always chooses the vertex with minimum current distance.

Key Takeaways:

- Dijkstra's algorithm finds shortest paths using a greedy approach.
- It works only for non-negative edge weights.
- Uses a priority queue (min-heap) for efficiency.
- Can find shortest paths to all vertices or a single destination.