GRAPHS AND NETWORK DATA STRUCTURE

Graph Representation

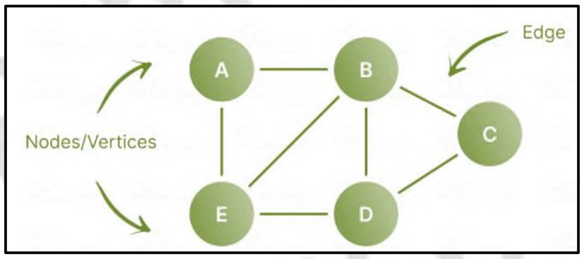
Graph Data Structure

A collection of interconnected nodes that have data, which represent relationships and connections between objects. Think of it like how STI eLMS features work, from Student, Class, Module, Group, Page, Comment, to Grades, anything that has data is a node.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of **nodes/vertices** (V), which represent objects.
- A collection of **edges** (E), represented as ordered pairs of vertices (u ,v). This represents the connections between these objects.

For example, on STI eLMS, people are nodes, and their friends are edges.



In the graph:

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

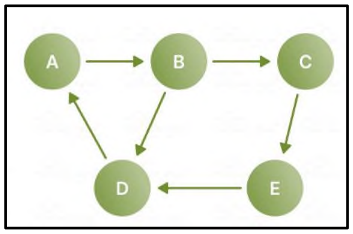
Terminologies

- **Adjacency**: A vertex is adjacent to another vertex if there is an edge connecting them. Vertices A and C are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that goes from vertex point A to vertex point B is called a **path**. A-E, B-E, and A-B are paths from vertex A to vertex E.
- **Directed Graph**: A graph in which an edge (u, v) does not necessarily mean that there is an edge (v, u) as well. Arrows represent the edges in such a graph to show the direction of the edge.

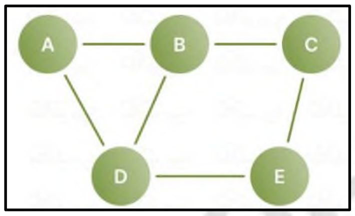
Types of Graph Data Structures

Graphs can be categorized based on their characteristics and properties:

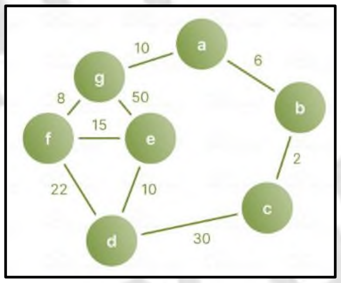
1. **Directed Graph**: Edges have a direction, meaning they go from one node to another in a specific way. For example, on STI eLMS, if Den follows Anthony, there is an edge from Den to Anthony but not necessarily from Den to Patricia and Sarah.



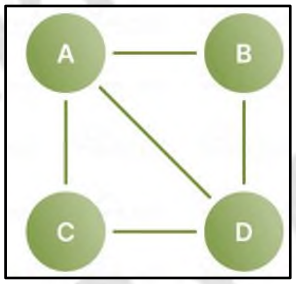
2. **Undirected Graph**: Edges do not have a direction. They connect two nodes without any particular order. For example, on STI eLMS, if Den is friends with Patricia, then Patricia is also friends with Den. The edge goes both ways.



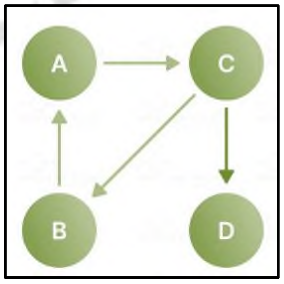
3. **Weighted Graph**: Edges have weights or costs associated with them. These weights can represent distances, costs, or any other metric. For example, a road map where the weights on the edges represent the distance between cities.



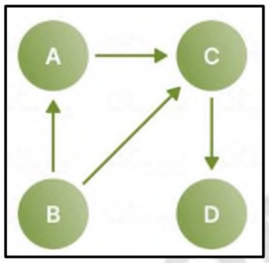
4. **Unweighted Graph**: All edges have the same weight, typically considered as 1. For example, a simple social network where each friendship has the same importance.



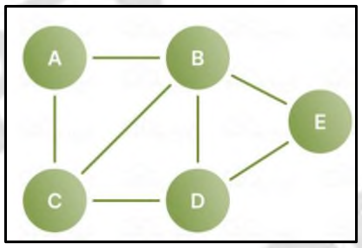
5. **Cyclic Graph**: Contains at least one cycle, meaning it is possible to start at a node and follow a path that leads back to the same node. For example, a graph representing routes between cities where some routes form a loop.



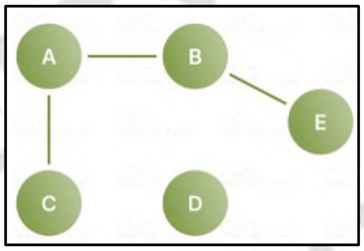
6. **Acyclic Graph**: An acyclic graph does not contain any cycles. For example, a family tree where no child node points back to its ancestors.



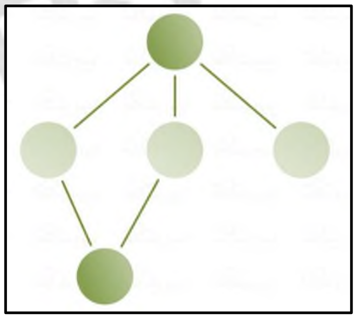
7. **Connected Graph**: Has a path between every pair of nodes. For example, a small network where every computer can reach every other computer directly or indirectly.



8. **Disconnected Graph**: Has at least one (1) pair of nodes with no path between them. For example, a network of isolated groups of people.



9. **Bipartite Graph**: Has its nodes divided into two (2) sets such that no two (2) nodes within the same set are adjacent. For example, a graph representing students and courses where edges show which student is enrolled in which course.

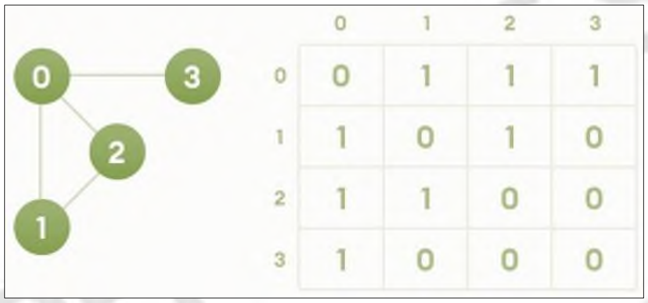


Graph Representation

Graphs are commonly represented in two (2) ways: Adjacency Matrix and Adjacency List.

Adjacency Matrix
A 2D array of V x V vertices. Each row and column presents a vertex. If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.

For example:



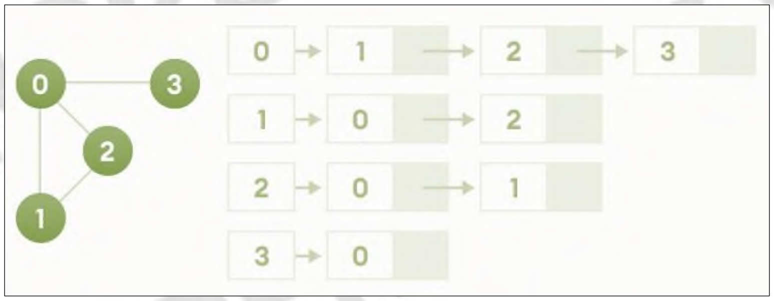
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Since it is an undirected graph, for edge (0, 2), the edge (2,0) is also marked, making the adjacency matrix symmetric about the diagonal.

Edge lookup, a way of checking if an edge exists between vertex A and vertex B, is extremely fast in adjacency matrix representation. Still, a space is reserved for every possible link between all vertices (V x V), so it requires more space.

Adjacency List
An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex, and each element in its linked list represents the other vertices that form an edge with the vertex.

For example:



An adjacency list is efficient in terms of storage because only the values for the edges are stored. For a graph with millions of vertices, this is a lot of space saved.

The program below shows a **minimal Java program** with edges among four (4) vertices:

```
import java.util.*;

public class Main {
    // Inner Graph class using an adjacency list representation
    static class Graph {
        // Each vertex maps to a list of its neighbors
        private final Map<String, List<String>> adj = new HashMap<>();
        // Ensure a vertex exists in the adjacency list
        public void addVertex(String v) {
            adj.computeIfAbsent(v, k -> new ArrayList<>());
            // if v is not present, create an empty list
        }

        // Add an undirected edge between u and v
        public void addEdge(String u, String v) {
            addVertex(u);  // ensure u exists
            addVertex(v);  // ensure v exists
            adj.get(u).add(v); // add v to u’s adjacency list
            adj.get(v).add(u); // add u to v’s adjacency list (because undirected)
        }

        // Print adjacency list for all vertices
        public void printAdjacency() {
            for (var entry : adj.entrySet()) {
                System.out.println(entry.getKey() + " -> " + entry.getValue());
            }
        }
    }

    public static void main(String[] args) {
        Graph g = new Graph();  // Create a new Graph object

        // Define edges of the graph (4 vertices: A, B, C, D)
        g.addEdge("A", "B");  // connect A and B
        g.addEdge("A", "C");  // connect A and C
        g.addEdge("B", "D");  // connect B and D
        g.addEdge("C", "D");  // connect C and D

        // Print adjacency list to visualize the graph
        System.out.println("Adjacency list:");
        g.printAdjacency();
    }
}
```

The output:

```
Adjacency list:
A -> [B, C]
B -> [A, D]
C -> [A, D]
D -> [B, C]
```

Graph Traversal Techniques

Graph traversal algorithms are used to visit all the vertices and edges in a graph. The two (2) most common traversal methods are Depth-First Search (DFS) and Breadth-First Search (BFS), with each method having its own algorithm and applications.

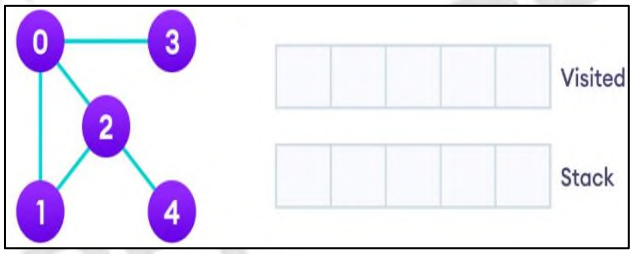
Depth-First Search (DFS)
Or Depth-first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

A standard DFS implementation puts each vertex of the graph into one of two (2) categories: Visited and Not Visited.

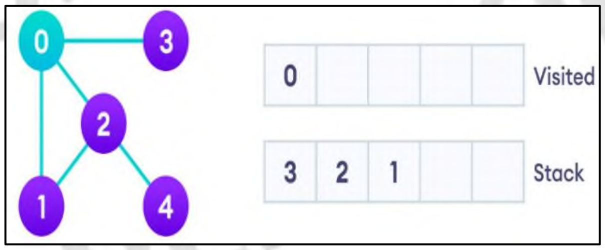
The DFS algorithm works as follows:

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 that are not in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

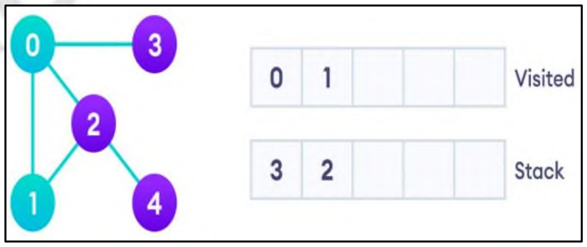
For example, using an undirected graph with five (5) vertices:



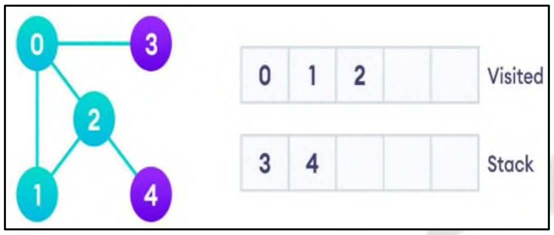
Starting from vertex 0, the DFS algorithm begins by putting it in the Visited list and putting all its adjacent vertices in the stack.



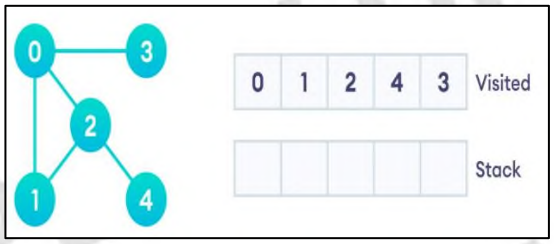
Next, visit the element at the top of the stack, i.e., 1, and go to its adjacent nodes. Since 0 has already been visited, visit 2 instead.



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



After visiting the last element 3, it does not have any unvisited adjacent nodes, so the Depth First Traversal of the graph is completed.



The program below shows a simple undirected graph with **4 vertices (A, B, C, D)**.

```
import java.util.*;
public class DFSExample {
    static class Graph {
        private final Map<String, List<String>> adj = new HashMap<>();
        // Add an undirected edge between u and v
        void addEdge(String u, String v) {
            adj.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
            adj.computeIfAbsent(v, k -> new ArrayList<>()).add(u);
        }
        // Recursive DFS starting from 'u'
        void dfs(String u, Set<String> visited) {
            visited.add(u);                    // mark current node as visited
            System.out.print(u + " ");         // process the node (print it)
            for (String v : adj.getOrDefault(u, List.of())) {
                if (!visited.contains(v)) {
                    dfs(v, visited);           // visit unvisited neighbors
                }
            }
        }
    }
    public static void main(String[] args) {
        Graph g = new Graph();
        // Build a small graph: A-B, A-C, B-D, C-D
        g.addEdge("A", "B");
        g.addEdge("A", "C");
        g.addEdge("B", "D");
        g.addEdge("C", "D");

        System.out.println("DFS Traversal starting from A:");
        g.dfs("A", new HashSet<>());  // call DFS with empty visited set
    }
}
```

The output:

```
DFS Traversal starting from A:
A B D C //The exact order can vary (e.g., A C D B) depending on the adjacency list order.
```

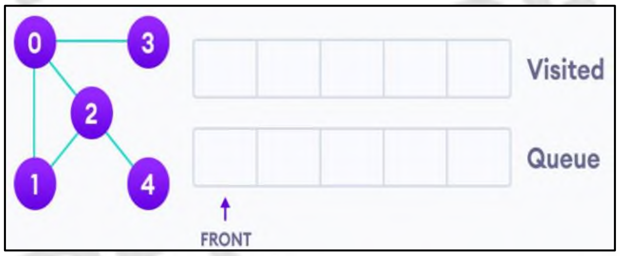
Breadth-First Search (BSF)
or Breadth First Search is typically implemented using an iterative approach with a queue. It explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. This also puts each vertex of the graph into one of two (2) categories: Visited and Not Visited.

The BFS algorithm works as follows:

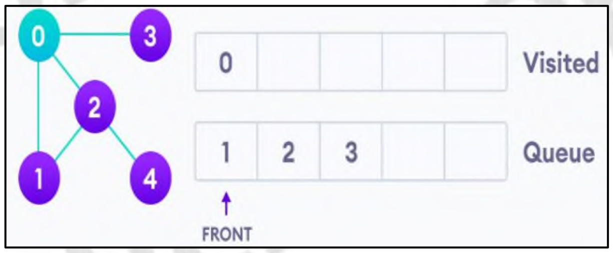
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones that are not in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two (2) different disconnected parts. To make sure that every vertex is covered, run the BFS algorithm on every node.

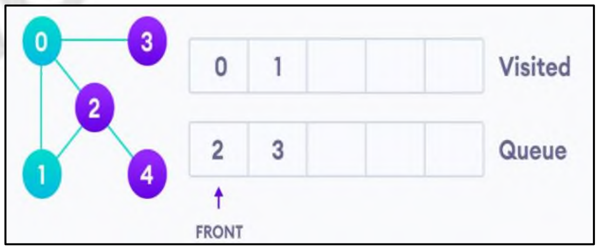
For example, using an undirected graph with five (5) vertices:



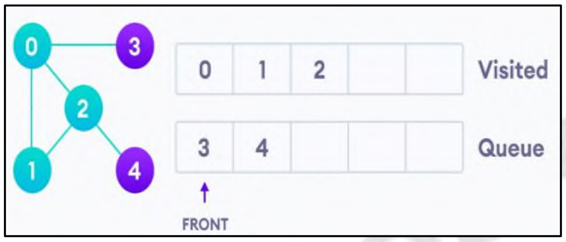
Starting from vertex 0, the BFS algorithm begins by putting it in the Visited list and putting all its adjacent vertices in the queue.



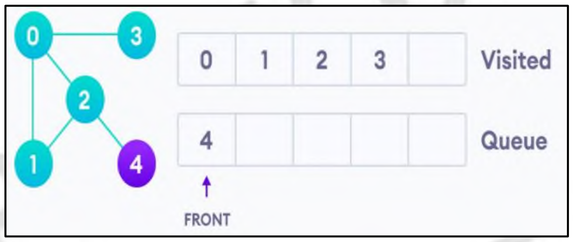
Next, visit the element at the from of the stack i.e., 1, and go to its adjacent nodes. Since 0 has already been visited, visit 2 instead.



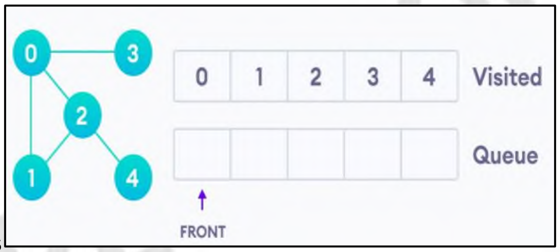
Vertex 2 has an unvisited adjacent vertex in 4, so add that to the back of the queue and visit 3, which is at the front of the queue.



Only 4 remain in the queue since the only adjacent node of 3 i.e. 0, is already visited. Visit it.



Since the queue is empty, the Breadth First Traversal of the graph is now completed.



s

The program below displays a Java program that demonstrates BFS on a simple undirected graph with four (4) vertices (A, B, C, D).

```
import java.util.*;

public class BFSExample {
    static class Graph {
        private final Map<String, List<String>> adj = new HashMap<>();

        // Add an undirected edge between u and v
        void addEdge(String u, String v) {
            adj.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
            adj.computeIfAbsent(v, k -> new ArrayList<>()).add(u);
        }

        // Breadth-First Search starting from 's'
        void bfs(String s) {
            Set<String> visited = new HashSet<>();    // track visited nodes
            Queue<String> q = new ArrayDeque<>();     // queue for BFS

            visited.add(s);       // mark start as visited
            q.add(s);             // enqueue start
            while (!q.isEmpty()) {
                String u = q.poll();       // remove from queue
                System.out.print(u + " "); // process (print) the node

                for (String v : adj.getOrDefault(u, List.of())) {
                    if (!visited.contains(v)) {
                        visited.add(v);    // mark neighbor as visited
                        q.add(v);          // enqueue neighbor
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        Graph g = new Graph();

        // Build a small graph: A-B, A-C, B-D, C-D
        g.addEdge("A", "B");
        g.addEdge("A", "C");
        g.addEdge("B", "D");
        g.addEdge("C", "D");

        System.out.println("BFS Traversal starting from A:");
        g.bfs("A");
    }
}
```

The output:

```
BFS Traversal starting from A:
A B C D //BFS explores level by level, so from A, it visits B and C, then finally D.
```

Application of Graphs

Here are some applications of the graph data structure.

Computer Science and Networking
In networking, **routers, servers, and computers** are represented as vertices, while **communication links** are modeled as edges. Routing algorithms, such as Dijkstra's or Bellman-Ford, are implemented to determine the most efficient path for data packets to travel from one point to another.

Similarly, the entire structure of the **World Wide Web** can be represented as a graph where web pages are vertices and hyperlinks are edges. This representation enables search engines to rank and retrieve relevant pages using graph algorithms like **PageRank**. Additionally, in peer-to-peer systems and distributed file sharing, nodes are treated as graph vertices to efficiently locate and recover resources.

Artificial Intelligence and Machine Learning
In artificial intelligence (AI), graphs are indispensable for solving problems that involve connectivity and structured relationships. **Pathfinding algorithms**, for example, are applied in video games and robotics, where a character or robot must find the shortest or safest path to a destination.

In machine learning, **knowledge graphs** represent entities (such as people, places, or objects) and their relationships, which improves semantic search and natural language understanding. **Graph Neural Networks (GNNs)** extend the power of deep learning by allowing models to learn directly from graph-structured data, making them useful in predicting chemical properties, analyzing social networks, or detecting fraud.

Operations Research and Logistics
Graphs are widely applied in operations research to optimize resources, costs, and time. Transportation systems can be modeled as graphs where cities, stations, or airports serve as vertices and roads, railways, or flight routes form the edges. This enables efficient route planning, as seen in GPS applications like Google Maps and Waze, which compute the shortest or fastest path between two points.

Graphs are also essential in **supply chain management**, where suppliers, warehouses, and retailers form a network that can be analyzed to minimize costs and maximize efficiency. In project management, tools such as **PERT (Program Evaluation Review Technique)** and **CPM (Critical Path Method)** represent project tasks as vertices and task dependencies as edges. These graph-based methods allow managers to calculate project timelines, identify bottlenecks, and allocate resources effectively.

Biology and Chemistry
In the sciences, graphs provide a natural way to model biological and chemical systems. For example, **protein-protein interaction networks** represent proteins as vertices and interactions as edges, allowing researchers to identify key proteins responsible for cellular functions or diseases.

Gene regulatory networks use directed graphs to show how genes influence one another's expression, which aids in understanding genetic disorders and biological pathways. In chemistry, molecules are represented as graphs where atoms are vertices and chemical bonds are edges. This modeling enables computational chemists to compare molecular structures through graph isomorphism algorithms, which helps in drug discovery, chemical informatics, and predicting molecular behavior.

Software Engineering
Graph theory is central to software design and analysis. **Control flow graphs** are used in compilers and program analysis to represent the flow of execution within a program, helping detect unreachable code or optimize performance.

Dependency graphs track the relationships between software modules, ensuring correct compilation order and supporting automated build tools. In modern version control systems such as **Git**, the history of commits is represented as a **directed acyclic graph (DAG)**, where each commit points to its parent(s). This makes merging, branching, and rollback operations more efficient and reliable. Software testing also employs graphs, particularly state transition graphs, to ensure all possible states and transitions of a system are tested.

Mathematics and Theoretical Applications
Within mathematics and theoretical computer science, graphs form the basis for many problem-solving approaches. **Graph coloring** is used to assign resources under constraints, such as ensuring no two adjacent exams in a timetable share the same time slot, or managing register allocation in compilers.

Network flow algorithms are applied to problems like maximizing bandwidth in communication networks, optimizing traffic flow, or solving bipartite matching in job assignment scenarios. Graph isomorphism, which tests whether two graphs are structurally identical, has applications in chemistry for comparing molecular structures and in computer science for plagiarism detection or image recognition. These mathematical applications demonstrate the abstract power of graph theory in diverse computational settings.

Everyday Applications

Graph data structures extend beyond specialized fields and are visible in everyday technologies. **Navigation systems** like GPS and ride-hailing apps rely on graphs to find optimal routes through road networks. **Electric circuits** can be modeled as graphs where junctions are vertices and wires are edges, allowing for systematic circuit analysis.

Recommendation engines in e-commerce platforms such as Amazon and Netflix build user-item graphs to suggest products or movies based on user preferences and shared patterns. Epidemiology uses graphs to track the **spread of diseases**, with individuals as vertices and possible transmissions as edges; this helps predict outbreaks and plan interventions. Organizational hierarchies and file systems also adopt tree structures (a type of graph), making it easier to represent and manage relationships among entities.

References:

Oracle. (n.d.a.). *Graph*. Retrieved on September 18, 2025, from https://docs.oracle.com/en/database/oracle/oracle-database/23/nfcoa/graph.html#GUID-graph3

Programiz. (2025). *Graph data structure*. Retrieved on September 18, 2025, from https://www.programiz.com/dsa/graph

WSCube Tech. (2025). *Graph Data Structure: Types, Uses, Examples, Algorithms*. Retrieved on September 17, 2025, from https://www.wscubetech.com/resources/dsa/graph-data-structure