Graphs

A Introduction to Graphs

Graphs are a fundamental data structure in computer science, used to represent relationships between objects. They consist of vertices (or nodes) and edges (connections between nodes). Graphs can be directed or undirected, weighted or unweighted, and can be used to model various real-world scenarios such as social networks, transportation systems, and more.

The terminology used in graphs includes:

- **Vertex (Node)**: A fundamental part of a graph, representing an entity.
- **Edge**: A connection between two vertices.
- **Degree (of a Vertex)**: The number of edges incident to a vertex. In directed graphs, we distinguish between *in-degree* (number of incoming edges) and *out-degree* (number of outgoing edges).
- **Path**: A sequence of edges connecting a sequence of vertices.
- **Cycle**: A path that starts and ends at the same vertex without repeating any edges.

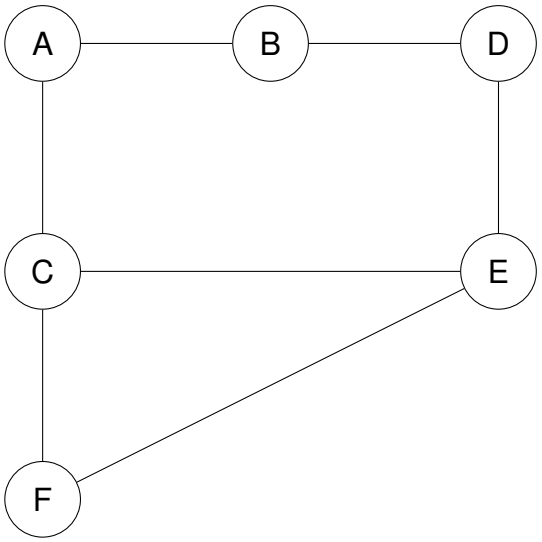


Figure 1: A sample undirected graph

Figure 1 illustrates the following graph concepts:

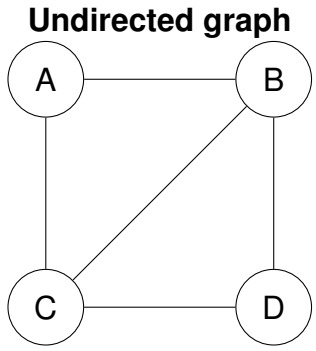
- **Vertices (Nodes):** The labeled circles A, B, C, D, E, and F represent the vertices of the graph.
- **Edges:** The lines connecting pairs of vertices, such as (A, B), (A, C), (B, D), (C, E), (D, E), and (C, F).
- **Degree:** Vertex C has degree 3 because it is connected to vertices A, E, and F.
- **Path:** A $\rightarrow$ B $\rightarrow$ D is a path from A to D.
- **Cycle:** The path C $\rightarrow$ E $\rightarrow$ F $\rightarrow$ C is a cycle.

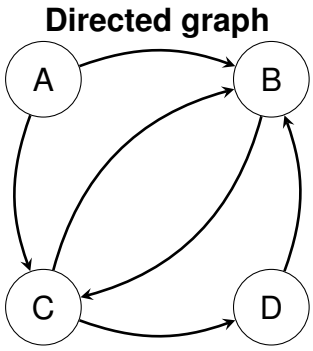
In this document, we will explore different types of graphs, discuss various ways to represent them (such as adjacency matrices and adjacency lists), and introduce some common algorithms used to process graphs, including traversal, shortest path, and connectivity algorithms.

B Directed vs. Undirected Graphs

Graphs can be classified into two main types based on the direction of their edges:

- **Directed Graphs (Digraphs):** In directed graphs, edges have a direction, meaning they go from one vertex to another. For example, an edge from A to B does not imply an edge from B to A.
- **Undirected Graphs:** In undirected graphs, edges do not have a direction. An edge between A and B implies a connection in both directions.





In the two examples above, the undirected graph shows connections between vertices without direction, while the directed graph has arrows indicating the direction of each edge. It is worth noticing that in the undirected graph, the path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D form a circle, but there is no such path in the directed graph, as the edges have a specific direction. Besides, in the directed graph, the path B $\rightarrow$ C $\rightarrow$ B is a cycle, while in the undirected graph such a path does not exist.

Edges in a graph can be represented as pairs of vertices. The way these pairs are defined depends on whether the graph is directed or undirected:

- **Undirected Graphs:** An edge is represented as an *unordered pair* of vertices, such as $\{u, v\}$. This means the edge connects u and v with no direction; $\{u, v\}$ is the same as $\{v, u\}$.
- **Directed Graphs:** An edge is represented as an *ordered pair* of vertices, such as (u, v). This indicates a direction from u (the source) to v (the destination); (u, v) is different from (v, u).

Using these representations, a graph G can be formally defined as $G = (V, E)$, where V is the set of vertices and E is the set of edges. For undirected graphs, E is a set of unordered pairs, while for directed graphs, E is a set of ordered pairs.

This distinction is important when modeling relationships: for example, a friendship network (undirected) uses unordered pairs, while a follower network (directed) uses ordered pairs. The choice of edge representation directly affects how we store, process, and analyze graphs in algorithms and data structures.

Practice Exercise:

Consider the following real-world systems. For each, discuss whether you would model the system using a directed or undirected graph, and explain your reasoning:

1. **Road Map:** Cities are vertices, and roads are edges.
2. **Twitter Follower Network:** Users are vertices, and a ”follows” relationship is an edge.
3. **Facebook Friendship Network:** Users are vertices, and a ”friendship” is an edge.
4. **Citation Network:** Research papers are vertices, and a citation from one paper to another is an edge.
5. **Airline Flight Routes:** Airports are vertices, and direct flights are edges.

For each system, justify your choice of directed or undirected edges, and discuss how this choice affects the types of questions you can answer about the system.

C Weighted vs. Unweighted Graphs

Graphs can also be classified based on whether their edges have weights or not:

- **Unweighted Graphs:** In unweighted graphs, all edges are considered equal, and there is no cost associated with traversing them. These graphs are often used to represent simple relationships where the presence or absence of an edge is more important than its weight.
- **Weighted Graphs:** In weighted graphs, each edge has a numerical value (weight) associated with it, representing the cost, distance, or capacity of the connection. Weighted graphs are useful for modeling more complex relationships where the strength or cost of a connection matters.

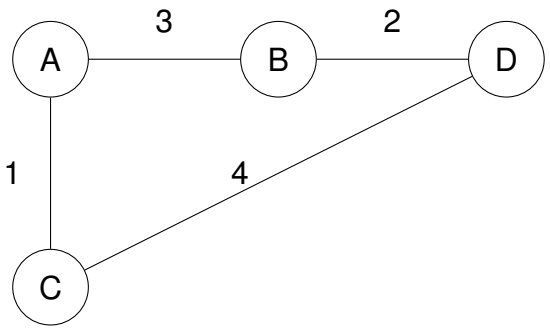


Figure 2: A sample weighted graph with edges labeled by their weights.

Figure 2 illustrates a weighted graph where each edge is labeled with its weight. For example, the edge between vertices A and B has a weight of 3, indicating the cost or distance associated with that connection. Weighted graph can be directed or undirected, just like unweighted graphs. In the example, the edges are undirected, meaning the connection between A and B is bidirectional.

Weighted graphs are particularly useful in scenarios where the relationships between entities have varying strengths or costs. For instance, in a transportation network, the weights could represent distances or travel times between locations. In social networks, weights might indicate the strength of relationships between users. Understanding the distinction between weighted and unweighted graphs is crucial for selecting the appropriate algorithms and data structures for graph-related problems. For instance, Figure 3 shows a directed

graph modeling bus lines and stops, where each edge is labeled with the bus line number. This representation allows for efficient routing and scheduling of bus services.

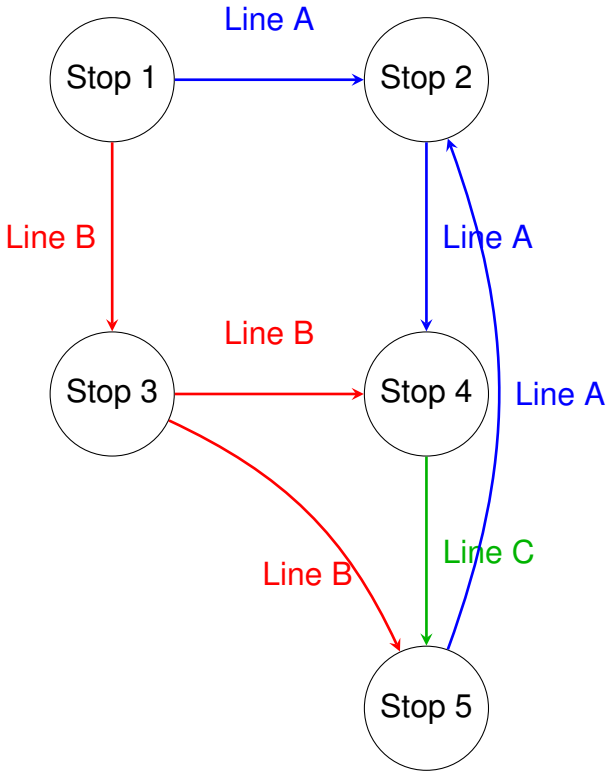


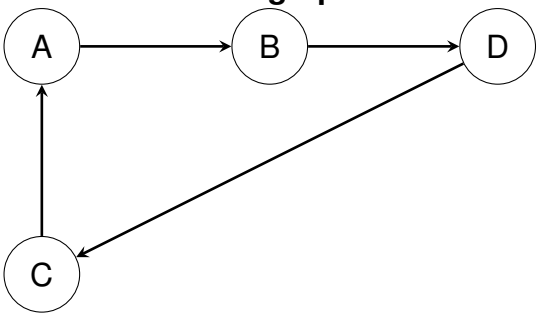
Figure 3: A directed graph modeling bus lines (edges) and bus stops (vertices), with each edge labeled by the bus line.

D Connected vs. Disconnected Graphs

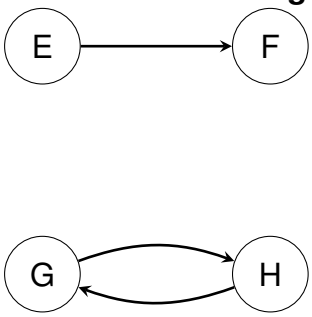
Graphs can also be classified based on their connectivity:

- **Connected Graphs:** A graph is connected if there is a path between every pair of vertices. In other words, all vertices are reachable from any starting vertex. Connected graphs are important in many applications, such as network design and communication.
- **Disconnected Graphs:** A graph is disconnected if there are at least two vertices in the graph that are not connected by a path. Disconnected graphs can be further classified into connected components, which are maximal connected subgraphs.

Connected directed graph

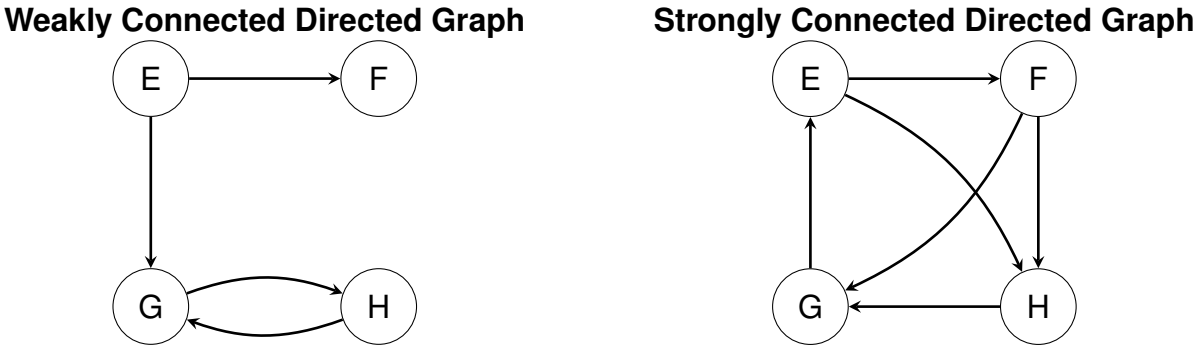


Disconnected directed graph



In the examples above, the first graph is a connected directed graph, meaning there is a path between every pair of vertices. The second graph is a disconnected directed graph, because there is no path connecting any vertex in the first component to any vertex in the

second component. For instance, there is no path from vertex G to vertex F, and vice versa, due to the direction of the edges.
In directed graphs, the concept of connectivity is more nuanced than in undirected graphs, as the direction of edges can create isolated vertices or components. We can have a **strongly connected graph**, where every vertex is reachable from every other vertex in the component, or a **weakly connected graph**, where the underlying undirected graph is connected. The left graph is strongly connected, while the right graph is weakly connected.



E Cyclic vs. Acyclic Graphs

Graphs can also be classified based on the presence of cycles:

- **Cyclic Graphs:** A graph is cyclic if it contains at least one cycle, which is a path that starts and ends at the same vertex without repeating any edges. Cyclic graphs can be directed or undirected.
- **Acyclic Graphs:** A graph is acyclic if it does not contain any cycles. Acyclic graphs can also be directed or undirected. Directed acyclic graphs (DAGs) are particularly important in many applications, such as scheduling and data processing.
- **Directed Acyclic Graphs (DAGs):** A directed graph that has no cycles. DAGs are often used to represent dependencies, such as task scheduling or version control systems.

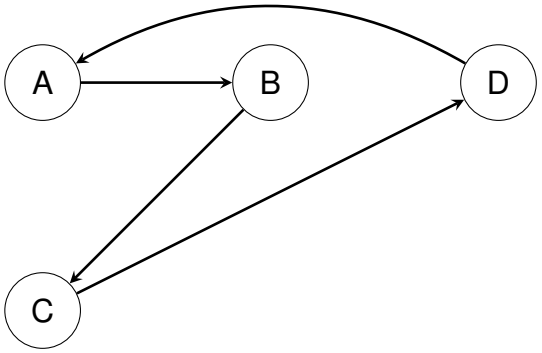


Figure 4: A cyclic directed graph with a cycle A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A.

Figure 4 illustrates a cyclic directed graph, where the cycle is formed by the path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A. This means that starting from vertex A, you can traverse through vertices B, C, and D, and return to A without repeating any edges.
Figure 5 illustrates an acyclic directed graph (DAG), where there are no cycles. In this graph, you can traverse from vertex A to B and C, and then to D, but you cannot return to A or create a cycle.
Note that an acyclic connected undirected graph is called a **tree**. A tree is a special type of graph.

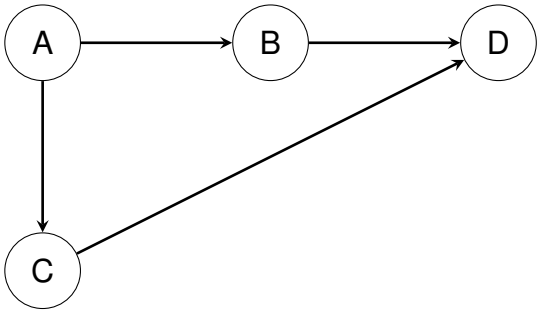


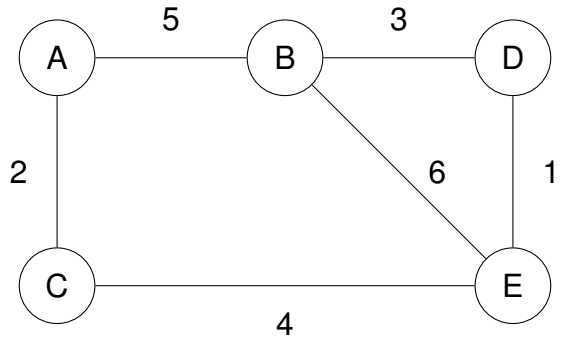
Figure 5: An acyclic directed graph (DAG) with no cycles.

F Graph Representation

Graphs can be represented in various ways, depending on the specific requirements of the application. The two most common representations are:

- **Adjacency Matrix:** A 2D array of size V x V, where V is the number of vertices. The element at row i and column j is 1 if there is an edge from vertex i to vertex j, and 0 otherwise. This representation is simple and allows for quick edge lookups, but it can be memory-intensive for sparse graphs.
- **Adjacency List:** An array of lists, where each list corresponds to a vertex and contains the neighbors of that vertex. This representation is more space-efficient for sparse graphs and allows for easy iteration over the neighbors of a vertex.
- **Edge List:** A list of all edges in the graph, where each edge is represented as a pair of vertices (u, v). This representation is simple and can be efficient for certain algorithms, but it may require more complex data structures to quickly access neighbors.

For example, if we consider the following weighted undirected graph:



The adjacency matrix representation of this graph would look like this:

	A	B	C	D	E
A	0	5	2	0	0
B	5	0	0	3	6
C	2	0	0	0	4
D	0	3	0	0	1
E	0	6	4	1	0

The adjacency list representation of the same graph would look like this:

Vertex	Neighbors (Weights)
A	B (5), C (2)
B	A (5), D (3), E (6)
C	A (2), E (4)
D	B (3), E (1)
E	B (6), C (4), D (1)

The edge list representation of the same graph would look like this:

Edge	Weight
(A, B)	5
(A, C)	2
(B, D)	3
(B, E)	6
(C, E)	4
(D, E)	1

Each representation has its advantages and disadvantages. The adjacency matrix is easy to implement and allows for quick edge lookups, but it can be inefficient in terms of space for sparse graphs. The adjacency list is more space-efficient and allows for easy iteration over neighbors, but it may require more complex code for edge lookups. The edge list is simple and can be efficient for certain algorithms, but it may not be as convenient for neighbor access.

Practice Exercise:

Implement a simple graph representation in C++ using both an adjacency matrix and an adjacency list. Your implementation should include the following features:

- A class for the graph that supports adding vertices and edges.
- Methods to display the graph using both representations.
- A method to check if an edge exists between two vertices.
- A method to find all neighbors of a given vertex.

Provide sample code demonstrating the usage of your graph class, including creating a graph, adding vertices and edges, and displaying the graph in both representations.

G Graph Traversal Algorithms

Graph traversal algorithms are used to explore the vertices and edges of a graph systematically. The two most common traversal algorithms are:

- **Depth-First Search (DFS):** This algorithm explores as far as possible along each branch before backtracking. It can be implemented using a stack (either explicitly or via recursion).
- **Breadth-First Search (BFS):** This algorithm explores all neighbors at the present depth prior to moving on to nodes at the next depth level. It is typically implemented using a queue.

In this section, we will discuss the implementation details of the graph traversal algorithms.

Depth-First Search (DFS) DFS can be implemented using a recursive approach or an explicit stack. The basic idea is to start from a source vertex, mark it as visited, and explore each unvisited neighbor recursively until all reachable vertices are visited. The algorithm can be used to find connected components, detect cycles, and perform topological sorting in directed graphs. Details of the algorithm can be found in the pseudocode below.

Listing 1: Depth-First Search (DFS) Pseudocode

```
DFS(graph, start):
    create an empty set visited
```

```
    call DFS-Visit(start, visited)

DFS-Visit(vertex, visited):
    mark vertex as visited
    for each neighbor in graph[vertex]:
        if neighbor not in visited:
            DFS-Visit(neighbor, visited)
```

Breadth-First Search (BFS) BFS is typically implemented using a queue. The algorithm starts from a source vertex, marks it as visited, and enqueues all its unvisited neighbors. The process continues until all reachable vertices are visited. The BFS algorithm is particularly useful for finding the shortest path in unweighted graphs and can be used to explore all vertices at the current depth before moving on to the next level. The pseudocode for BFS is shown below.

Listing 2: Breadth-First Search (BFS) Pseudocode

```
BFS(graph, start):
    create an empty set visited
    create an empty queue
    enqueue start into the queue
    mark start as visited

    while queue is not empty:
        vertex = dequeue from the queue
        for each neighbor in graph[vertex]:
            if neighbor not in visited:
                mark neighbor as visited
                enqueue neighbor into the queue
```

Practice Exercise:

Given the following directed graph represented by its adjacency matrix:

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

1. Represent this graph using both an adjacency list and an edge list.
2. On paper, perform both DFS and BFS traversals starting from vertex A. Show the order in which the vertices are visited for each traversal.
3. Reflect: How could you implement these traversal algorithms in a programming language such as C++ or Python? What data structures would you use, and what steps would your algorithm follow?