Data Structures Course – Graphs Module

Authors: Refat Othman and Diaeddin Rimawi

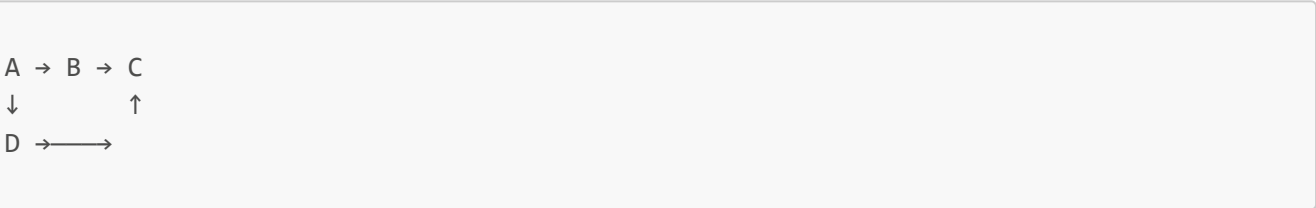
Lecture 1: Introduction to Graphs (2 hours)

1. Definition and Terminology

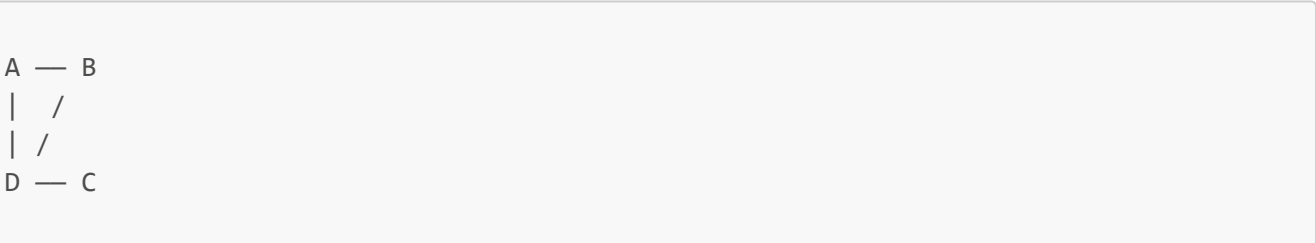
- **Graph (G)**: A collection of vertices (nodes) and edges (connections between nodes). Formally, a graph G = (V, E), where V is the set of vertices and E is the set of edges.
- **Vertex (plural: vertices)**: Fundamental units or points in a graph. Each vertex represents an entity.
- **Edge**: A link between two vertices. Represents the relationship or connection between the vertices.

2. Types of Graphs

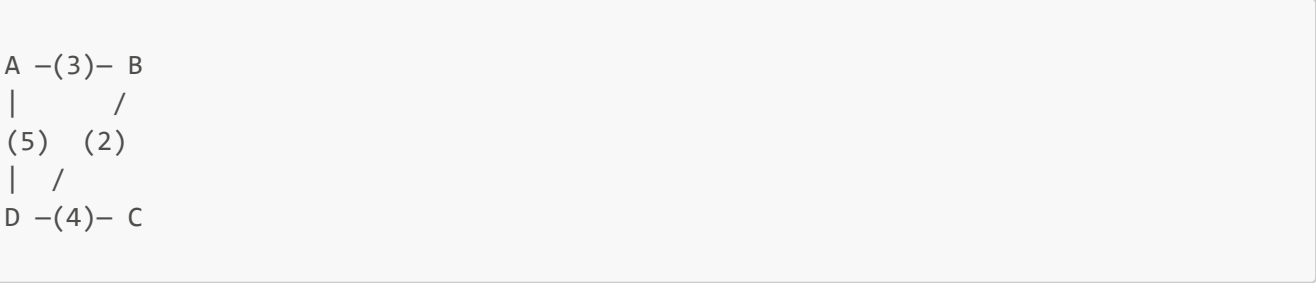
- **Directed Graph (Digraph)**: Edges have a direction, indicating a one-way relationship between two vertices.



- **Undirected Graph**: Edges do not have direction; the connection is mutual between vertices.



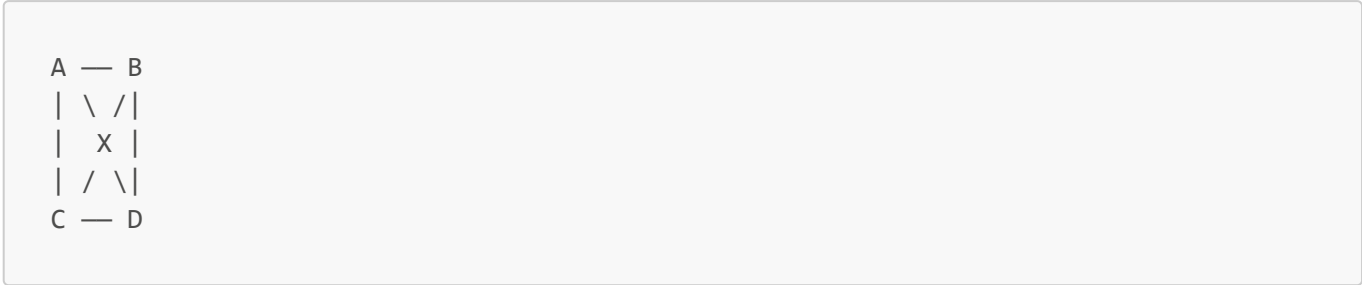
- **Weighted Graph**: Each edge has an associated weight or cost.



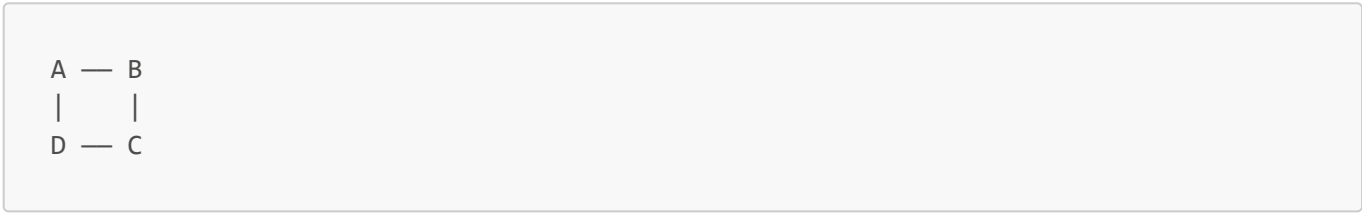
- **Unweighted Graph**: Edges have no associated weights; all connections are treated equally.

3. Graph Connectivity Types

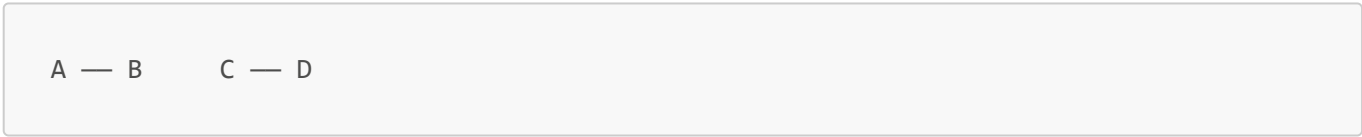
- **Complete Graph**: Every pair of distinct vertices is connected by a unique edge.



- **Fully Connected Graph**: In a directed graph with three vertices (A, B, C), every vertex has a directed path to every other vertex. (e.g., A → B, A → C, B→ A, B→C, C→A, C→B)
- **Connected Graph**: There exists a path between every pair of vertices.

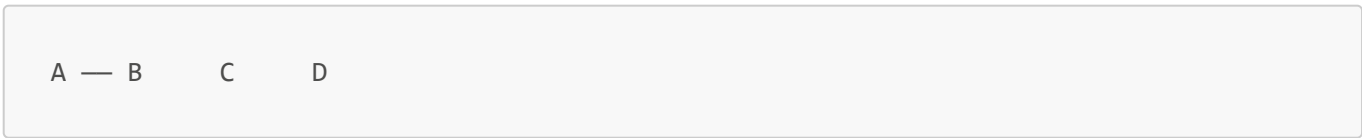


- **Disconnected Graph**: The graph has at least one vertex or group of vertices not reachable from the others. For example, if vertex D has no path to or from the rest of the graph, the graph is considered disconnected.



- **Sparse vs. Dense Graphs**:
  - **Sparse Graph**: A sparse graph is one in which the number of edges is much lower than the maximum number of possible edges. In a graph with `V` vertices, the theoretical maximum number of edges is `V*(V-1)/2` in undirected graphs and `V*(V-1)` in directed graphs. Sparse graphs are useful for representing real-world systems like road maps or communication networks where each node connects to only a few others.

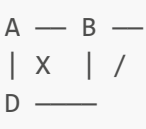
Example:



In this graph, with four vertices and only one connection (A to B), the graph is sparse because most nodes are not directly connected.

- **Dense Graph**: A dense graph is one where the number of edges is close to the maximum number of possible edges. For `V` vertices, this means having nearly `V*(V-1)/2` edges in an undirected graph, or `V*(V-1)` in a directed graph. Dense graphs are common in systems with high interconnectivity, such as computer networks, where each device communicates with many others.

Example:



This graph shows four vertices (A, B, C, D) where almost every pair is directly connected. The graph includes multiple crossing edges, reflecting high connectivity and minimal sparsity.

To clarify the difference, consider this:

- A graph with 4 vertices has a maximum of 6 edges in an undirected setting (since 4*(4-1)/2 = 6).
- If the graph contains only 2 edges, it is sparse.
- If it contains 5 or 6 edges, it is dense.
- If the graph contains 3 or 4 edges, it falls in a gray area. It may not be strictly considered sparse or dense. In such cases, the classification may depend on the specific application or context, though generally:
    - Closer to 3: tends toward sparse.
    - Closer to 4 or more: tends toward dense.

4. Graph Representations in Code

- **Adjacency List**: A dictionary where each key is a vertex and the value is a list of tuples representing connected vertices and edge weights.
- **Adjacency Matrix**: A 2D array where the cell at row i and column j represents the edge between vertex i and vertex j.

5. Implementation Example

```
class Edge:
    def __init__(self, target, weight):
        self.target = target  # Reference to target Vertex
        self.weight = weight

class Vertex:
    def __init__(self, label):
        self.label = label
        self.edges = []  # List of Edge instances

    def add_edge(self, target_vertex, weight):
        self.edges.append(Edge(target_vertex, weight))

    def remove_edge(self, target_vertex):
        self.edges = [e for e in self.edges if e.target != target_vertex]

class Graph:
```

```
    def __init__(self):
        self.vertices = []  # List of Vertex instances

    def add_vertex(self, label):
        if not any(v.label == label for v in self.vertices):
            self.vertices.append(Vertex(label))

    def find_vertex(self, label):
        for v in self.vertices:
            if v.label == label:
                return v
        return None

    def add_edge(self, from_label, to_label, weight):
        from_vertex = self.find_vertex(from_label)
        to_vertex = self.find_vertex(to_label)
        if from_vertex and to_vertex:
            from_vertex.add_edge(to_vertex, weight)

    def remove_edge(self, from_label, to_label):
        from_vertex = self.find_vertex(from_label)
        to_vertex = self.find_vertex(to_label)
        if from_vertex and to_vertex:
            from_vertex.remove_edge(to_vertex)

    def remove_vertex(self, label):
        target_vertex = self.find_vertex(label)
        if target_vertex:
            self.vertices = [v for v in self.vertices if v != target_vertex]
            for v in self.vertices:
                v.remove_edge(target_vertex)
```

This object-oriented implementation uses a list of `Vertex` objects instead of a dictionary. Each vertex maintains its own outgoing weighted edges.

6. Graph Traversal Techniques

- **Breadth-First Search (BFS)**: Ideal for finding the shortest path (in unweighted graphs). The following method can be added to the `Graph` class:

```
def bfs(self, start_label):
    start_vertex = self.find_vertex(start_label)
    if not start_vertex:
        return
    visited = set()
    queue = [start_vertex]

    while queue:
        vertex = queue.pop(0)
        if vertex.label not in visited:
```

```
            print(vertex.label)
            visited.add(vertex.label)
            for edge in vertex.edges:
                if edge.target.label not in visited:
                    queue.append(edge.target)

# Add this method to the Graph class as:
Graph.bfs = bfs
```

- **Depth-First Search (DFS)**: Useful for exploring all paths. The following method can be added to the `Graph` class using an explicit stack (list):

```
def dfs(self, start_label):
    start_vertex = self.find_vertex(start_label)
    if not start_vertex:
        return
    visited = set()
    stack = [start_vertex]

    while stack:
        vertex = stack.pop()
        if vertex.label not in visited:
            print(vertex.label)
            visited.add(vertex.label)
            for edge in reversed(vertex.edges):  # reverse to maintain consistent
order
                if edge.target.label not in visited:
                    stack.append(edge.target)

# Add this method to the Graph class as:
Graph.dfs = dfs
```

Lecture 2: Exercises on Graphs (2 hours)

Shortest Path Using Dijkstra's Algorithm

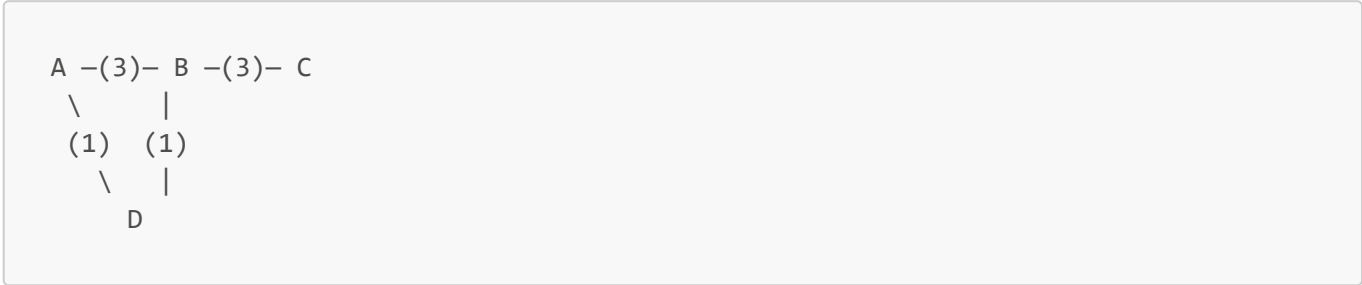
Dijkstra's algorithm is a classic algorithm used to compute the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It maintains a set of nodes whose minimum distance from the source is known and repeatedly selects the node with the smallest tentative distance.

Steps:

1. Assign a tentative distance value to every vertex (zero for the start vertex and infinity for all others).
2. Set the start vertex as current and mark all others as unvisited.

3. For the current vertex, consider all unvisited neighbors and calculate their tentative distances through the current vertex.
4. Update the neighbor's distance if the new distance is smaller.
5. After checking all neighbors, mark the current vertex as visited. A visited vertex will not be checked again.
6. Select the unvisited vertex with the smallest tentative distance and repeat from step 3.

Example: Graph:



If we start at A:

- Distance to A = 0
- Distance to D = 1 (via A)
- Distance to B = 2 (via A → D)
- Distance to C = 5 (via A → D → B → C)

Dijkstra's algorithm guarantees the shortest path from A to every reachable vertex.

Exercise 1: *Path Existence* Given a directed graph, determine if there exists a path from node `x` to node `y`.

Exercise 2: *Cycle Detection in a Directed Graph* Given a graph, write a function that detects whether a cycle exists using DFS.

Exercise 3: *Topological Sorting* Given a directed acyclic graph (DAG), return a valid topological order using Kahn's algorithm or DFS.

A topological sort is a linear ordering of the vertices in a directed acyclic graph such that for every directed edge `u → v`, vertex `u` comes before `v` in the ordering. This is useful in applications like task scheduling, compilation order in build systems, or resolving dependencies among software modules.

Example: Graph:

A valid topological sort for this graph could be: A, D, B, C

Task: Implement a method inside your Graph class that returns a list of vertex labels in valid topological order. Handle edge cases like cycles by detecting them and raising an error or returning an appropriate message.

Assignment: Graph Application

Title: "Modeling City Map with Graphs"

Problem Statement: Model a simplified map of a city using a directed weighted graph. Each vertex represents a location (e.g., hospital, school, store), and each edge represents a one-way road with a given travel time.

Tasks:

1. Build a class-based graph implementation.
2. Populate the graph with at least 6 locations and 10 roads.
3. Implement:
    - A function to add/remove locations and roads.
    - A function to compute the shortest path from one location to another (use Dijkstra's algorithm).
    - A function to list all reachable locations from a starting point (use BFS).

Submission Requirements:

- Python code file (.py)
- PDF report describing:
    - The graph design
    - Example use cases
    - Screenshots of the output

Deadline: [Insert Date Here]