Learning Objectives

Learners will be able to…

- **Define Graph ADT**
- **Create undirected weighted and unweighted graphs**
- **Implement a graph using an adjacency list**

Introduction to Graphs

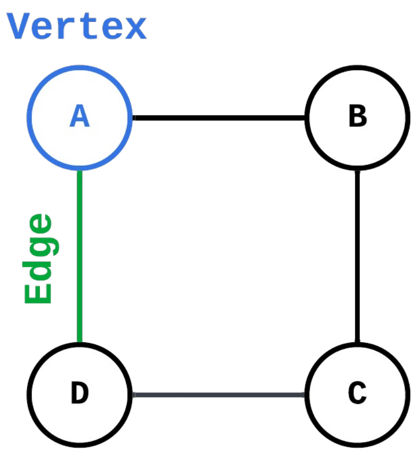
Graphs

Graphs are incredibly versatile and powerful structures used to model various real-world scenarios. This introductory assignment aims to demystify the concept of graphs by breaking down their definition and explaining their significance in a simple, approachable manner.

What is a Graph?

In its most basic form, a graph is a set of items connected by links. These items are known as **vertices** (singular: vertex) or nodes, and the links that connect them are called **edges**. Think of a graph as a network where vertices represent points in the network, and edges represent the connections between these points.

- **Vertices (Nodes):** The fundamental units of a graph. They can represent objects like cities, computers, or even concepts.
- **Edges:** The connections between vertices. An edge can symbolize a relationship or a pathway between two vertices.



In the example above, **A**, **B**, **C**, and **D** are **vertices**(nodes). The lines connecting them (**AB**, **BC**, **CD**, and **AD**) are **edges**.

Why are Graphs Important?

Graphs are a critical tool in computer science and related fields. They are used to model various types of relationships and interactions in a network, such as social networks, computer networks, biological networks, and many more. Understanding graphs helps in solving complex problems like:

- Finding the shortest path for travel or data transfer.
- Analyzing social networks for connections and influences.
- Optimizing routes in transportation and logistics.

Graphs are a mathematical and abstract representation of a collection of vertices (nodes) and edges that connect these nodes. Graphs can be incredibly complex, with hundreds, thousands, or even millions of vertices and edges that model relationships or connections in data.

Types of Graphs

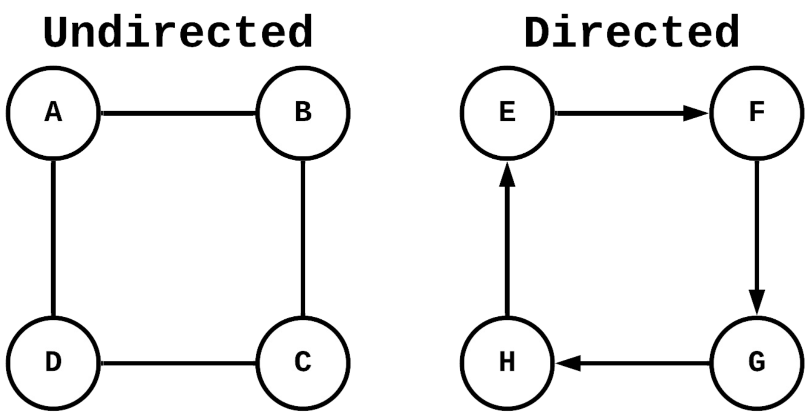
Different Graph Types

After grasping the basic concept of graphs, it’s important to delve into the different types of graphs you will encounter. Each type of graph has distinct properties and uses, making them suitable for various applications.

Undirected vs Directed Graphs

In an **undirected graph**, the edges have no direction. The edges simply connect two vertices, without indicating a specific direction from one vertex to the other. In this graph, an edge like `A --- B` means there is a connection between A and B, but it does not specify a direction from A to B or B to A.

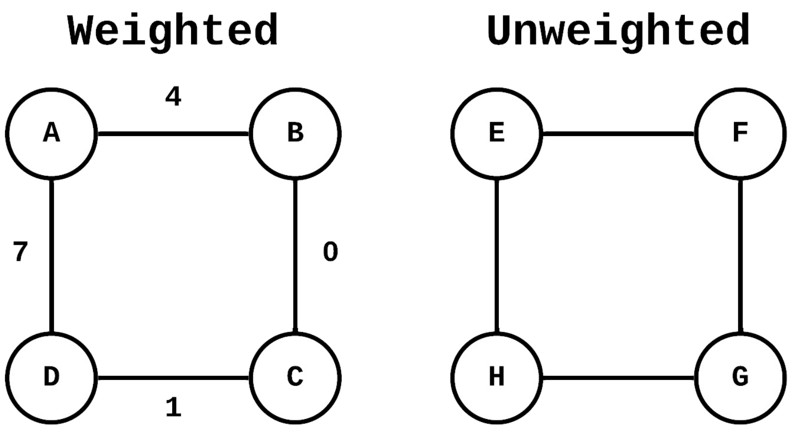
A **directed graph**, or **digraph**, features edges with a specific direction. Each edge points from one vertex to another. In this graph, an edge like `E --> F` means there is a connection between E and F, and the direction goes from E to F. These directional edges define the flow or relationship direction between nodes.



Weighted vs Unweighted Graphs

Weighted graphs assign a weight, cost, or value to each edge. These weights are crucial in algorithms that find the shortest path or the least expensive route in a graph. The numbers represent the weights of the edges. For instance, the edge `A --4-- B` has a weight of 4.

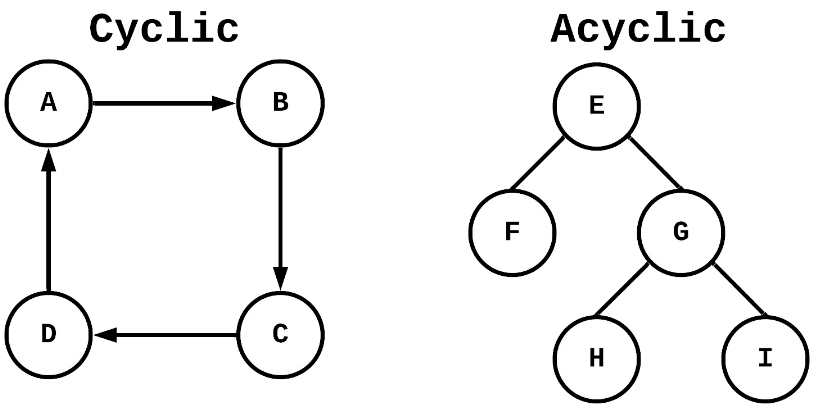
In contrast to weighted graphs, **unweighted graphs** do not have values associated with their edges. The edges simply indicate a connection or relationship.



Cyclic vs. Acyclic Graphs

Cyclic Graphs have at least one cycle. A cycle occurs when you can start at a vertex and return to it by traveling along a sequence of edges. In the example below, you can start at any vertex and arrive back at the starting point.

Acyclic Graphs do not have any cycles. A common example of an acyclic graph is a tree in computer science. Trees flow from the parent to the child. There is no “looping back” to a node higher up in the tree. Therefore trees are acyclic.

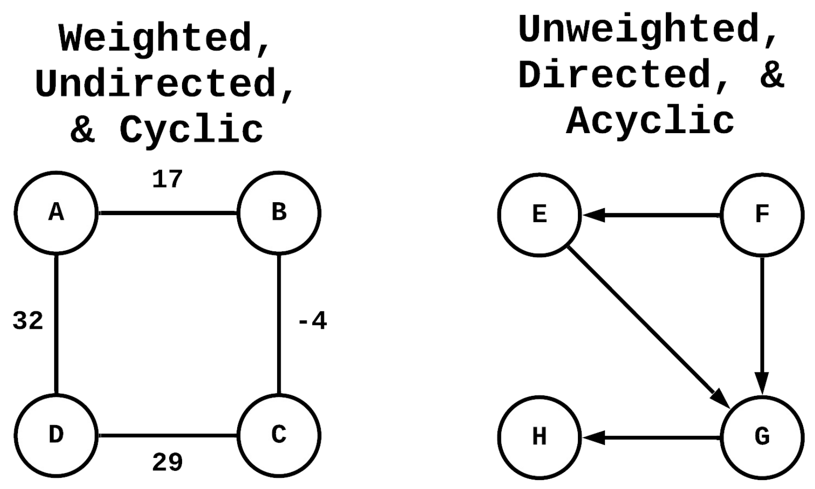


▾ **Trees as graphs**

> A tree is a special type of graph, which is an acyclic connected graph. Every tree is a graph, but not every graph is a tree.

Combining Graph Types

These types of graphs are not mutually exclusive; graphs can have two or more of the types listed above. Combining these different characteristics gives graphs more nuance and offers a better description of the relationships they represent.


Weighted, Undirected, & Cyclic
A
B
C
D
17
32
-4
29
Unweighted, Directed, & Acyclic
E
F
G
H

Graph Representation in Programming

Graphs can be represented in programming languages using different data structures, the most common being:

- **Adjacency List**: A list where each element represents a node and stores a list of its neighbors.
- **Adjacency Matrix**: A 2D array where the cell at the `ith` row and `jth` column is `true` if there is an edge from vertex `i` to vertex `j`.

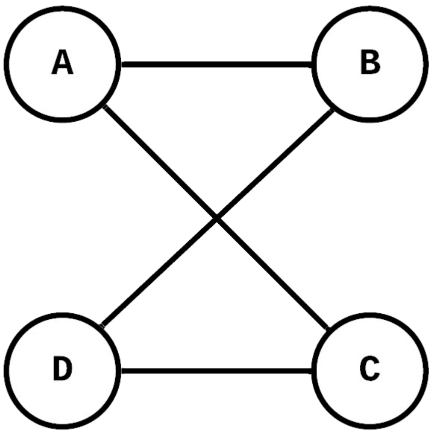
In the upcoming pages, we'll explore how to implement these graph types in C++, along with operations like adding and deleting vertices and edges. Understanding these concepts will be crucial in tackling more complex graph-related problems and algorithms.

Undirected, Unweighted Graph

Implementing a Simple Graph

In this section, we will create a simple undirected and unweighted graph structure. To keep track of the vertices and edges, we will use an adjacency list. This is an efficient way of representing graphs in terms of space, especially when the graph is sparse (few edges relative to nodes).

We will use the `map` from the C++ Standard Library for our adjacency list. The key will be a string (the name of the vertex), and the value will be a list (or vector) of strings representing edges associated with the vertex.



Coding the Graph

Start by creating the `Graph` class. It has only one private attribute, `adjList`, which will be our adjacency list. In the constructor, initialize `adjList` as an empty `map`.

```
#include <iostream>
#include <map>
#include <list>
#include <string>

class Graph {
private:
     std::map<std::string, std::list<std::string>> adjList;

public:
    Graph() {
        // Constructor initializes an empty adjacency list
    }


};
```

The addVertex method is a public method that adds a vertex to the graph. It takes a string, representing the name of the vertex, and does not return a value. The vertex is added to the adjacency list as a key with an empty list of edges.

```
    void addVertex(const std::string& label) {
        adjList[label] = std::list<std::string>();
    }
```

The addEdge method is a public method that adds an edge between two vertices. It takes two strings, representing the names of the vertices, and does not return a value. Since this is an undirected graph, the edge is added in both directions.

```
    void addEdge(const std::string& vertex1, const std::string&
        vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
```

Finally, create the printGraph method. This public method has no parameters and does not return a value. It iterates over the keys of the adjacency list and prints the connected vertices for each key.

```
    void printGraph() const {
        for (const auto& vertex : adjList) {
            std::cout << "Vertex " << vertex.first << " is
        connected to: ";
            for (const auto& edge : vertex.second) {
                std::cout << edge << " ";
            }
            std::cout << std::endl;
        }
    }
```

Testing the Code

The main method in C++ is the entry point for execution. First, instantiate a Graph object. Then, add the vertices `A`, `B`, `C`, and `D` to the graph. Add edges so that A is connected to `B`, `A` is connected to `C`, `B` is connected to `D`, and `C` is connected to `D`. Finally, use the `printGraph` method to print a representation of the graph.

```cpp
int main() {
    Graph graph;

    // Adding vertices
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");

    // Adding edges
    graph.addEdge("A", "B");
    graph.addEdge("A", "C");
    graph.addEdge("B", "D");
    graph.addEdge("C", "D");

    // Print the graph
    graph.printGraph();

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```cpp
#include <iostream>
#include <map>
#include <list>
#include <string>

class Graph {
private:
    std::map<std::string, std::list<std::string>> adjList;

public:
    Graph() {
        // Constructor initializes an empty adjacency list
    }

    void addVertex(const std::string& label) {
        adjList[label] = std::list<std::string>();
    }

    void addEdge(const std::string& vertex1, const
        std::string& vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }

    void printGraph() const {
        for (const auto& vertex : adjList) {
            std::cout << "Vertex " << vertex.first << " is
        connected to: ";
            for (const auto& edge : vertex.second) {
                std::cout << edge << " ";
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    Graph graph;

    // Adding vertices
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");

    // Adding edges
    graph.addEdge("A", "B");
    graph.addEdge("A", "C");
    graph.addEdge("B", "D");
    graph.addEdge("C", "D");

    // Print the graph
    graph.printGraph();

    return 0;
}
```

You should see the following output:

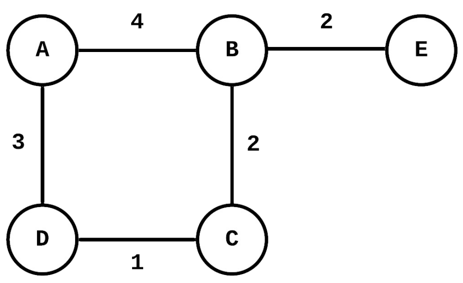
```
Vertex A is connected to: B C
Vertex B is connected to: A D
Vertex C is connected to: A D
Vertex D is connected to: B C
```

This output represents the connections in the graph: `A` is connected to `B` and `C`, `B` is connected to `A` and `D`, and so on. This simple visualization helps understand the structure of the graph and how the vertices are interconnected. By creating and visualizing this graph, we have laid the groundwork for further exploration into graph algorithms and applications.

Undirected, Weighted Graph

Implementing a Weighted Graph

After working with an undirected, unweighted graph, let’s turn our attention to an undirected, weighted graph. Each edge not only connects two vertices but also carries a weight. The weight can represent key details about the relationship between vertices (like distance or cost).



Coding the Graph

For a weighted graph, we will still use an adjacency list. In addition, we need a way to store the weights along with the vertices. We can achieve this by creating a `Pair` structure which stores a string for the vertex and an integer for the weight.

Start by creating the `WeightedGraph` class. This will represent the weighted graph. We also need the `Pair` structure. This structure has two attributes—a string to represent the vertex and an integer to represent the weight. The constructor for the `Pair` structure takes a string and an integer, and initializes these attributes. Finally, implement a `toString` function so that it prints the vertex followed by the weight in parentheses.

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>

class WeightedGraph {
private:
    struct Pair {
        std::string vertex;
        int weight;

        Pair(const std::string& vertex, int weight) :
        vertex(vertex), weight(weight) {}

        std::string toString() const {
            return vertex + " (" + std::to_string(weight) + ")";
        }
    };
```

The `WeightedGraph` class has the `adjList` attribute used to represent an adjacency list. Again, we are going to use the `Map` interface and the `std::map` class. Use the default constructor for the `WeightedGraph` class.

```cpp
    std::map<std::string, std::vector<Pair>> adjList;

public:
    WeightedGraph() = default;
```

Just as before, we are going to have methods to add a vertex. When adding a vertex, create a new `Pair` structure. The `addEdge` method now takes two strings (the two vertices) and an integer (the weight). Get the first vertex from the map and add the vertex and the weight to the `Pair` structure. Since this is an undirected graph, we need to get the second vertex from the map and add the first vertex and the weight to the `Pair` structure.

```cpp
    void addVertex(const std::string& label) {
        adjList[label] = std::vector<Pair>();
    }

    void addEdge(const std::string& vertex1, const std::string&
        vertex2, int weight) {
        adjList[vertex1].emplace_back(vertex2, weight);
        adjList[vertex2].emplace_back(vertex1, weight); // Since
        the graph is undirected
    }
```

The `removeEdge` method takes two strings representing vertices in teh graph. It does not return anything. Find each vertex in the adjacency list. Use two conditionals to remove each vertex from the graph. Removal is done with the erase-remove idiom.

```
    void removeEdge(const std::string& vertex1, const
        std::string& vertex2) {
        auto it1 = adjList.find(vertex1);
        auto it2 = adjList.find(vertex2);

        if (it1 != adjList.end()) {
            auto& list1 = it1->second;
            list1.erase(std::remove_if(list1.begin(),
        list1.end(),
                                        [&vertex2](const Pair&
        p) { return p.vertex == vertex2; }),
                        list1.end());
        }

        if (it2 != adjList.end()) {
            auto& list2 = it2->second;
            list2.erase(std::remove_if(list2.begin(),
        list2.end(),
                                        [&vertex1](const Pair&
        p) { return p.vertex == vertex1; }),
                        list2.end());
        }

    }
```

The `printGraph` method iterates over the adjacency list and prints each vertex (with context about connecting vertices and weights) in the graph.

```
    void printGraph() const {
        for (const auto& [vertex, neighbors] : adjList) {
            std::cout << "Vertex " << vertex << " is connected
        to: ";
            for (const auto& pair : neighbors) {
                std::cout << pair.toString() << " ";
            }
            std::cout << std::endl;
        }
    }
};
```

Testing the Code

This C++ code implements an undirected weighted graph with functionalities to add and remove edges and vertices, as well as print the graph. The output will show the connections between vertices along with their weights.

```
int main() {
    WeightedGraph graph;

    // Adding vertices
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");
    graph.addVertex("E");

    // Adding weighted edges
    graph.addEdge("A", "B", 4);
    graph.addEdge("B", "E", 2);
    graph.addEdge("A", "C", 3);
    graph.addEdge("B", "D", 2);
    graph.addEdge("C", "D", 1);

    std::cout << "Graph before removing edge:" << std::endl;
    graph.printGraph();

    // Removing an edge
    graph.removeEdge("A", "B");

    std::cout << "\nGraph after removing the edge between A and
        B:" << std::endl;
    graph.printGraph();

    return 0;
}
```

▼ **Code**

Your code should look like this:

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>

class WeightedGraph {
private:
    struct Pair {
        std::string vertex;
        int weight;

        Pair(const std::string& vertex, int weight) :
        vertex(vertex), weight(weight) {}

        std::string toString() const {
            return vertex + " (" + std::to_string(weight) +
        ")";
        }
    };

std::map<std::string, std::vector<Pair>> adjList;
```

```
public:
    WeightedGraph() = default;

    void addVertex(const std::string& label) {
        adjList[label] = std::vector<Pair>();
    }

    void addEdge(const std::string& vertex1, const
        std::string& vertex2, int weight) {
        adjList[vertex1].emplace_back(vertex2, weight);
        adjList[vertex2].emplace_back(vertex1, weight); //
        Since the graph is undirected
    }

    void removeEdge(const std::string& vertex1, const
        std::string& vertex2) {
        auto it1 = adjList.find(vertex1);
        auto it2 = adjList.find(vertex2);

        if (it1 != adjList.end()) {
            auto& list1 = it1->second;
            list1.erase(std::remove_if(list1.begin(),
        list1.end(),
                                        [&vertex2](const
        Pair& p) { return p.vertex == vertex2; }),
                        list1.end());
        }

        if (it2 != adjList.end()) {
            auto& list2 = it2->second;
            list2.erase(std::remove_if(list2.begin(),
        list2.end(),
                                        [&vertex1](const
        Pair& p) { return p.vertex == vertex1; }),
                        list2.end());
        }

    }

    void printGraph() const {
        for (const auto& [vertex, neighbors] : adjList) {
            std::cout << "Vertex " << vertex << " is
        connected to: ";
            for (const auto& pair : neighbors) {
                std::cout << pair.toString() << " ";
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    WeightedGraph graph;

    // Adding vertices
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");
    graph.addVertex("E");

    // Adding weighted edges
    graph.addEdge("A", "B", 4);
    graph.addEdge("B", "E", 2);
    graph.addEdge("A", "C", 3);
    graph.addEdge("B", "D", 2);
    graph.addEdge("C", "D", 1);

    std::cout << "Graph before removing edge:" <<
        std::endl;
    graph.printGraph();

    // Removing an edge
    graph.removeEdge("A", "B");

    std::cout << "\nGraph after removing the edge between A
        and B:" << std::endl;
    graph.printGraph();

    return 0;
}
```

You should see the following output:

```
Graph before removing edge:
Vertex A is connected to: B (4) C (3)
Vertex B is connected to: A (4) E (2) D (2)
Vertex C is connected to: A (3) D (1)
Vertex D is connected to: B (2) C (1)
Vertex E is connected to: B (2)

Graph after removing the edge between A and B:
Vertex A is connected to: C (3)
Vertex B is connected to: E (2) D (2)
Vertex C is connected to: A (3) D (1)
Vertex D is connected to: B (2) C (1)
Vertex E is connected to: B (2)
```

This output shows the graph's structure both before and after removing the edge between A and B, providing a comprehensive view of the weighted graph.

Recap

Graph Summary

As we conclude our introductory assignment on graphs, let's take a moment to review the key concepts and operations we have covered. This foundation sets the stage for more advanced topics in graph theory and their applications.

Key Concepts Covered

- **Graphs**
    - Graphs are a collection of vertices (nodes) connected by edges.
    - These structures are used to model complex relationships and networks.
- **Types of Graphs**
    - Different types of graphs were introduced, including undirected and directed graphs, weighted and unweighted graphs, and special forms like trees and cyclic/acyclic graphs.
    - These distinctions are important for understanding the suitability of various graphs in different scenarios.
- **Graph Representation**
    - We explored how graphs can be represented in programming, particularly focusing on adjacency lists and adjacency matrices.
    - This understanding is vital for implementing and manipulating graphs in real-world applications.
- **Implementing Graphs in Java**
    - Practical implementation began with creating an undirected, unweighted in Java and then expanded to include weighted graphs.
    - The concepts of vertices and edges were solidified through code, demonstrating how they form the backbone of a graph's structure.
- **Basic Graph Operations**
    - Core operations like adding and removing vertices and edges were implemented, showcasing how graphs can be dynamically altered.
    - For weighted graphs, we not only added edges with weights but also developed a method to remove edges, highlighting the dynamic nature of graphs.
- **Visualization and Understanding:**

- Methods to visualize and print graph structures were provided, aiding in understanding the abstract concept of graphs and making debugging easier.
- These visualizations are particularly helpful in grasping the complex relationships and connections within a graph.

Looking Ahead

In the assignments ahead, we will cover implementing graphs with an adjacency matrix, directed graphs, cyclic graphs, traversal, and common graph algorithms for finding the shortest path (Dijkstra's and Bellman-Ford algorithms) and spanning trees (Prim's and Kruskal's algorithms).

Formative Assessment 1

Formative Assessment 2