Learning Objectives

Learners will be able to…

- **Differentiate an adjacency list and an adjacency matrix**
- **Implement an adjacency list and an adjacency matrix**
- **Visualize each representation**
- **Implement and transpose a directed graph**

Introduction to Graph Representation

Representing a Graph

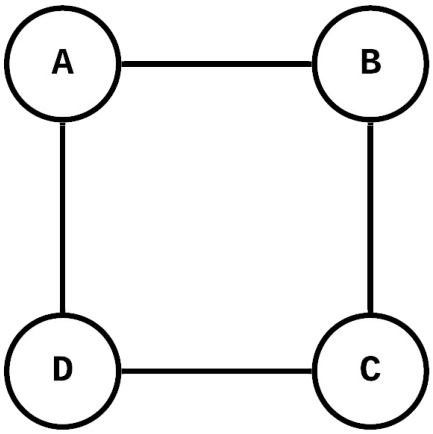
Graphs are versatile structures used to model complex relationships. The way we choose to represent these graphs in a program can significantly impact the efficiency and feasibility of graph operations. This assignment introduces two fundamental methods of representing graphs: adjacency matrices and adjacency lists.

Graph representation refers to the method of storing graph information in a computer's memory. This representation is crucial for performing any graph-related operations, such as traversals, pathfinding, and modifications. The choice of representation often depends on the specific requirements of the application, such as the need for efficient edge lookups or the frequency of modifications to the graph structure.

Adjacency Matrix

An adjacency matrix is a 2D array where each cell `(i, j)` indicates whether there is an edge from vertex `i` to vertex `j`. In this representation, the value in the cell can indicate the presence or absence of an edge, or in the case of weighted graphs, the weight of the edge.

Consider a simple undirected, unweighted graph:



Using an adjacency matrix, the example graph would look like the table below. If we look at the first row, we see that vertex `A` has a connection when the column contains a `1`. When you see a `0`, that means there is no connection. Since this is an unweighted graph, the possible values are `0` and `1`.

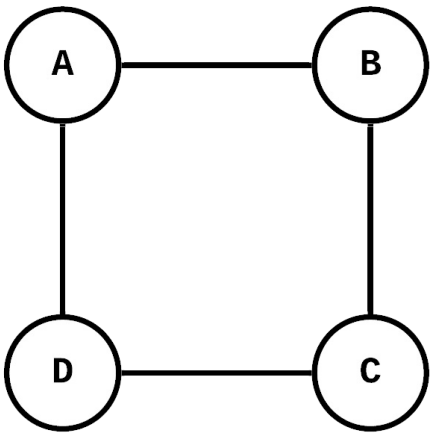
	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

An adjacency matrix is space-efficient for dense graphs (graphs where the number of edges is close to the maximum possible). If you want to look up if an edge exists between two vertices, the time complexity is $O(1)$ as this involves a simple array index access. Adding or removing edges is also an efficient operation. However, adding or removing vertices can be costly.

Adjacency List

An adjacency list represents a graph as a collection of lists. Each list describes the set of neighbors of a vertex in the graph. The index in the array represents a vertex, and each element in its list represents the other vertices that form an edge with the vertex.

We are going to use the same undirected, unweighted graph from above:



Using an adjacency list, the example graph would look like the list of lists below. The first entry shows that vertex `A` is connected to vertices `B` and `D`.

- A: [B, D]
- B: [A, C]
- C: [B, D]
- D: [A, C]

An adjacency list is space-efficient for sparse graphs (graphs with fewer edges). So if your graph does not have many edges, an adjacency list is more efficient than an adjacency matrix. It is also easier and more efficient to add vertices in an adjacency list as compared to an adjacency matrix.

Understanding these two methods of graph representation is essential for anyone looking to work with graphs in programming. On the following pages, we will explore each representation in detail, including their implementation in C++ and their practical applications. In addition, we will cover directed graphs.

Embarking on this exploration will equip you with the knowledge to select the most suitable graph representation for your specific use case, balancing factors such as performance, ease of implementation, and memory usage.

Adjacency List

Creating a Graph with an Adjacency List

we are going to use a `std::vector` of `std::vector<int>` to represent the adjacency list in C++.

Implementing the Graph

Start by creating the `AdjacencyListGraph` class that has a single attribute, `adjList`, which is a two-dimensional `std::vector` of type `int`. This implementation of a graph requires that both vertices and edges be represented as integers. In the constructor, initialize `adjList` to an empty `std::vector`.

```
#include <iostream>
#include <vector>

class AdjacencyListGraph {
private:
    std::vector<std::vector<int>> adjList;

public:
    AdjacencyListGraph() = default;
};
```

The `addVertex` method is a public method that does not have any parameters. This method adds a new `std::vector<int>` to `adjList`. The newly added `std::vector<int>` represents a vertex in the graph, and it will contain any other vertices to which it is connected.

```
    void addVertex() {
        adjList.push_back(std::vector<int>());
    }
```

The `addEdge` method is a public method that takes two integers representing two `vertices` in a graph. Instead of storing the retrieved `std::vector<int>` in local references, it directly retrieves and updates the adjacency list for each vertex. The method calls the `getOrCreateList` helper method to ensure that both vertices exist in the graph, and then adds each vertex to the other's adjacency list. This method handles the connections in an undirected graph by ensuring that `vertex1` is connected to `vertex2` and vice versa.

```
void addEdge(int vertex1, int vertex2) {
    // Ensure both vertices exist
    getOrCreateList(vertex1).push_back(vertex2);
    getOrCreateList(vertex2).push_back(vertex1);
}
```

The `getOrCreateList` method is a private helper method that takes an integer representing an index. We are going to use a while loop to see if the given index is greater than the size of the `std::vector`. If so, add an empty `std::vector<int>` to `adjList`. Then return the element at the given index.

```
    std::vector<int>& getOrCreateList(int index) {
        // Ensure adjList is large enough
        while (index >= static_cast<int>(adjList.size())) {
            adjList.push_back(std::vector<int>());
        }
        return adjList[index];
    }
```

The `printGraph` method is a public method that iterates over the outer `std::vector`, printing the index (`i` in the code below) followed by the list of connected vertices.

```
    void printGraph() const {
        for (int i = 0; i < static_cast<int>(adjList.size());
        i++) {
            std::cout << "Vertex " << i << " is connected to: ";
            for (int vertex : adjList[i]) {
                std::cout << vertex << " ";
            }
            std::cout << std::endl;
        }
    }
```

Testing the Code

To test our code, create the `main` method that instantiates an `AdjacencyListGraph` object. Add four vertices, then add edges between the vertices. Remember, we are using `std::vector` to manage our graph. That means we need to use an index to reference a vertex in the graph—pay close attention to this. Finally, print the graph.

```
int main() {
    AdjacencyListGraph graph;

    // Adding vertices
    graph.addVertex(); // vertex 0
    graph.addVertex(); // vertex 1
    graph.addVertex(); // vertex 2
    graph.addVertex(); // vertex 3

    // Adding edges
    graph.addEdge(0, 1); // edge 0-1
    graph.addEdge(1, 2); // edge 1-2
    graph.addEdge(2, 3); // edge 2-3
    graph.addEdge(3, 0); // edge 3-0

    // Print graph
    graph.printGraph();

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```
#include <iostream>
#include <vector>

class AdjacencyListGraph {
private:
    std::vector<std::vector<int>> adjList;

public:
    AdjacencyListGraph() = default;

    void addVertex() {
        adjList.push_back(std::vector<int>());
    }

    void addEdge(int vertex1, int vertex2) {
        // Ensure both vertices exist
        getOrCreateList(vertex1).push_back(vertex2);
        getOrCreateList(vertex2).push_back(vertex1);
    }

    std::vector<int>& getOrCreateList(int index) {
        // Ensure adjList is large enough
        while (index >= static_cast<int>(adjList.size())) {
            adjList.push_back(std::vector<int>());
        }
        return adjList[index];
    }

    void printGraph() const {
```

```
        for (int i = 0; i < static_cast<int>
        (adjList.size()); i++) {
            std::cout << "Vertex " << i << " is connected
        to: ";
            for (int vertex : adjList[i]) {
                std::cout << vertex << " ";
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    AdjacencyListGraph graph;

    // Adding vertices
    graph.addVertex(); // vertex 0
    graph.addVertex(); // vertex 1
    graph.addVertex(); // vertex 2
    graph.addVertex(); // vertex 3

    // Adding edges
    graph.addEdge(0, 1); // edge 0-1
    graph.addEdge(1, 2); // edge 1-2
    graph.addEdge(2, 3); // edge 2-3
    graph.addEdge(3, 0); // edge 3-0

    // Print graph
    graph.printGraph();

    return 0;
}
```

You should see the following output:

```
Vertex 0 is connected to: 1 3
Vertex 1 is connected to: 0 2
Vertex 2 is connected to: 1 3
Vertex 3 is connected to: 2 0
```

challenge

Try this variation

Update the main method so that you add one vertex only. Then add an edge between this vertex (index 0) and a vertex with index 4.

```
int main() {
    AdjacencyListGraph graph;

    graph.addVertex();
    graph.addEdge(0, 4);
    graph.printGraph();

    return 0;
}
```

▼ **What is happening?**

> Notice that the program did not throw an error even though we asked it to create an edge to a vertex that had not been added to the graph. The `getOrCreateList` method is the reason why. The index `4` is greater than `1`, which is the number of vertices in the graph. The `while` loop keeps adding empty `std::vector<int>` instances until `4` is equal to the number of vertices. The method then returns the fifth `std::vector<int>` and the edge is added.

Analysis

- **Space Complexity:** The space complexity for an adjacency list is $O(V + E)$, making it more efficient for sparse graphs.
- **Time Complexity for Edge Operations:** The time complexity to add an edge is $O(1)$. However, removing an edge has a time complexity of $O(E)$ in the worst case, as it may require traversing through all edges of a vertex to find the specific edge to remove.
- **Edge Lookup:** Checking if an edge exists between two vertices can take $O(V)$ time in the worst case, as it might require a linear search through the list of adjacent vertices.

The adjacency list representation is particularly beneficial for handling graphs with a large number of vertices but relatively fewer edges. It offers a more scalable and flexible approach, especially in scenarios where the graph is dynamically changing, with frequent additions and deletions of

vertices and edges. However, adjacency lists are less efficient when compared to an adjacency matrix. Adjacency lists can also consume more space if the graph is dense.

▼ **Adjacency Lists and Hash Maps**

> The example on this page used `std::vector` to represent an adjacency list instead of a hash map. This was done to show how graphs can be represented with other underlying data structures. However, using `std::vector` has limitations compared to hash maps. In future examples, we will use the `std::unordered_map` for better efficiency.

Adjacency Matrix

Creating a Graph with an Adjacency Matrix

In this section, we will translate the implementation of an adjacency matrix for graph representation in C++.

The adjacency matrix is particularly suitable for representing dense graphs, where the number of edges is close to the maximum possible. We will implement a simple undirected graph using an adjacency matrix in C++.

Implementing the Graph

The core of our implementation is a 2D array where each element represents a potential edge between two vertices. We will create the `AdjacencyMatrixGraph` class. The class has two attributes: a 2D array of boolean values and an integer representing the number of vertices in the graph.

```
#include <iostream>

class AdjacencyMatrixGraph {
private:
    bool** adjMatrix;
    int numVertices;

public:
    AdjacencyMatrixGraph(int numVertices);
    ~AdjacencyMatrixGraph();
    void addEdge(int i, int j);
    void removeEdge(int i, int j);
    void printMatrix();
};
```

The constructor takes an integer as a parameter. It sets the `numVertices` attribute to the given integer and allocates memory for the `2D` array. Since we are using dynamic memory allocation, we also implement a destructor to free the allocated memory.

```
AdjacencyMatrixGraph::AdjacencyMatrixGraph(int numVertices) {
    this->numVertices = numVertices;
    adjMatrix = new bool*[numVertices];
    for (int i = 0; i < numVertices; i++) {
        adjMatrix[i] = new bool[numVertices];
        for (int j = 0; j < numVertices; j++) {
            adjMatrix[i][j] = false; // Initialize all edges to
        false (no edges)
        }
    }
}

AdjacencyMatrixGraph::~AdjacencyMatrixGraph() {
    for (int i = 0; i < numVertices; i++) {
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;
}
```

The `addEdge` method is a public method that takes two integers representing two `vertices` in the graph. Since this is an undirected graph, the edge needs to be added in both directions. The `removeEdge` method performs the inverse operation, removing the edge in both directions.

```
void AdjacencyMatrixGraph::addEdge(int i, int j) {
    adjMatrix[i][j] = true;
    adjMatrix[j][i] = true; // For undirected graph
}

void AdjacencyMatrixGraph::removeEdge(int i, int j) {
    adjMatrix[i][j] = false;
    adjMatrix[j][i] = false; // For undirected graph
}
```

Finally, we create the `printMatrix` method to visualize the graph. We will iterate over the `2D` array with nested loops and represent an edge with `1` or the absence of an edge with `0`.

```
void AdjacencyMatrixGraph::printMatrix() {
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            std::cout << (adjMatrix[i][j] ? 1 : 0) << " ";
        }
        std::cout << std::endl;
    }
}
```

Testing the Code

We will now write the `main` method to test our code. Start by creating an `AdjacencyMatrixGraph` object with four vertices. Add edges between the vertices, print the graph, remove an edge, and print the graph again.

```
int main() {
    AdjacencyMatrixGraph graph(4);

    // Adding edges
    graph.addEdge(0, 1); // Edge A-B
    graph.addEdge(0, 2); // Edge A-C
    graph.addEdge(1, 3); // Edge B-D
    graph.addEdge(2, 3); // Edge C-D

    std::cout << "Initial Graph:" << std::endl;
    graph.printMatrix();

    std::cout << "\nGraph after removing edge B-D:" <<
        std::endl;
    graph.removeEdge(1, 3); // Removing Edge B-D
    graph.printMatrix();

    return 0;
}
```

▼ **Code**

Your code should look like this:

```
#include <iostream>

class AdjacencyMatrixGraph {
private:
    bool** adjMatrix;
    int numVertices;

public:
    AdjacencyMatrixGraph(int numVertices);
    ~AdjacencyMatrixGraph();
    void addEdge(int i, int j);
    void removeEdge(int i, int j);
    void printMatrix();
};

AdjacencyMatrixGraph::AdjacencyMatrixGraph(int numVertices)
        {
    this->numVertices = numVertices;
    adjMatrix = new bool*[numVertices];
    for (int i = 0; i < numVertices; i++) {
        adjMatrix[i] = new bool[numVertices];
        for (int j = 0; j < numVertices; j++) {
            adjMatrix[i][j] = false;
        }
    }
```

```
}

AdjacencyMatrixGraph::~AdjacencyMatrixGraph() {
    for (int i = 0; i < numVertices; i++) {
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;
}

void AdjacencyMatrixGraph::addEdge(int i, int j) {
    adjMatrix[i][j] = true;
    adjMatrix[j][i] = true;
}

void AdjacencyMatrixGraph::removeEdge(int i, int j) {
    adjMatrix[i][j] = false;
    adjMatrix[j][i] = false;
}

void AdjacencyMatrixGraph::printMatrix() {
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            std::cout << (adjMatrix[i][j] ? 1 : 0) << " ";
        }
        std::cout << std::endl;
    }
}

int main() {
    AdjacencyMatrixGraph graph(4);

    // Adding edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 3);

    std::cout << "Initial Graph:" << std::endl;
    graph.printMatrix();

    std::cout << "\nGraph after removing edge B-D:" <<
        std::endl;
    graph.removeEdge(1, 3);
    graph.printMatrix();

    return 0;
}
```

You should see the following output:

```
Initial Graph:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0

Graph after removing edge B-D:
0 1 1 0
1 0 0 0
1 0 0 1
0 0 1 0
```

Analysis

- **Space Complexity** : The space complexity of the adjacency matrix representation is $O(V^2)$, where V is the number of vertices. This is because we maintain a two-dimensional array regardless of the number of edges.
- **Time Complexity for Edge Operations** : Adding or removing an edge in an adjacency matrix takes $O(1)$ time since it involves setting or unsetting a single element in the matrix.
- **Edge Lookup** : Checking if an edge exists between two vertices is also $O(1)$, as it involves a direct lookup in the matrix.

An adjacency matrix is a powerful and intuitive way to represent graphs, particularly when the graph is dense or when we need to frequently check for the presence or absence of specific edges. However, it can be inefficient in terms of space for sparse graphs, and resizing the matrix to add or remove vertices is more complex.

Directed Graph

Creating a Directed Graph

Having explored the basic implementations of graphs using both adjacency matrices and adjacency lists, we are now equipped to delve into directed graphs. We are first going to work with a directed, weighted graph based on an adjacency list (using the `std::map` and `std::vector` from the STL). The bulk of the code is already in the editor. We are going to focus on adding and removing edges.

Find the section of code that indicates where to write your code. Start by adding the `addEdge` method, a public method that takes two strings representing `vertices` and an `integer` representing the weight. Because this is a directed graph, the connection should only go from one vertex to another vertex. The reciprocal connection does not exist. Get the key-value pair for the first vertex and add a new Pair object for the second vertex and the associated weight

```
    void addEdge(const std::string& fromVertex, const
        std::string& toVertex, int weight) {
        adjList[fromVertex].emplace_back(toVertex, weight);
    }
```

Create the `removeEdge` method to remove an edge in a directed graph. This public method takes two strings representing vertices. First, fetch the list of connected `Pair` objects from the first vertex. If the list is not `nullptr`, remove the Pair object for the second vertex if it exists. Again, this is a one-way operation since the graph is directed.

```
    void removeEdge(const std::string& fromVertex, const
        std::string& toVertex) {
        auto& list = adjList[fromVertex];

        list.erase(std::remove_if(list.begin(), list.end(),
            [&](const Pair& pair) { return pair.vertex ==
        toVertex; }), list.end());
    }
```

Code already exists in the `main` method, so there is nothing for us to add. However, note that there is a connection from `A` to `B` with a weight of `4`, and there is a connection from `B` to `A` with a weight of `5`. These are valid connections in a directed graph. Importantly, removing edge `AB` should not affect edge `BA`.

▼ Code

> Your code should look like this:
>
> ```
> void addEdge(const std::string& fromVertex, const
>     std::string& toVertex, int weight) {
>     adjList[fromVertex].emplace_back(toVertex, weight);
> }
>
> void removeEdge(const std::string& fromVertex, const
>     std::string& toVertex) {
>     auto& list = adjList[fromVertex];
>     list.erase(std::remove(list.begin(), list.end(),
>     Pair(toVertex, 0)), list.end());
> }
> ```

You should see the following output:

```
Graph before removing edge:
Vertex A is connected to: B (4) C (2)
Vertex B is connected to: C (3) A (5)
Vertex C is connected to:

Graph after removing the edge between A and B:
Vertex A is connected to: C (2)
Vertex B is connected to: C (3) A (5)
Vertex C is connected to:
```

This demonstrates how directed edges work in the graph, where removing the edge from A to B does not affect the edge from B to A.

Transpose of a Graph

For certain applications, such as finding strongly connected components, you may need the transpose of a graph (reversing the direction of all edges). Implementing this in an adjacency matrix involves creating a new matrix where rows and columns are swapped. Click the link below to open the `Transpose.cpp` file.

This time, we are going to represent a directed graph with an adjacency matrix. Most of the code is already in the editor. However, we are going to add methods to manage edges and transpose the graph. Start by creating the `addEdge` method. This public method takes two integers representing the vertices of the edge. Find the element in the 2D array and set its value to true. This is a directed graph, so we only need a one-way connection.

```
void addEdge(int fromVertex, int toVertex) {
    adjMatrix[fromVertex][toVertex] = true;
}
```

Next, create the `removeEdge` method. This public method takes two integers representing vertices. Find the element in the `2D` array and set its value to `false`. Again, this is a directed graph, so we do not have to worry about the reciprocal connection.

```
    void removeEdge(int fromVertex, int toVertex) {
        adjMatrix[fromVertex][toVertex] = false;
    }
```

To calculate the transpose of a graph, create the `transpose` method. This public method returns a `Transpos` object. Start by instantiating a Transpose object of the same size as the original graph. Use a nested loop to iterate over the `2D` array. If there is an edge between two vertices (`i` and `j` in the code below), create an edge with the opposite direction in the new graph. After the loops terminate, return the new graph.

```
public:
    Transpose transpose() {
        Transpose transposedGraph(numVertices);
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                if (adjMatrix[i][j]) {
                    transposedGraph.addEdge(j, i);
                }
            }
        }
        return transposedGraph;
    }
```

Again, the `main` method already has code in it. The method creates a graph with four `vertices`. It then adds the edges `AB`, `AC`, `BD`, and `CD`. The method prints the graph and creates a new, transposed graph which it also prints. You should see how the second graph has the edges `BA`, `CA`, `DB`, and `DC`.

▼ Code

> Your code should look like this:

```
    void addEdge(int fromVertex, int toVertex) {
        adjMatrix[fromVertex][toVertex] = true;
    }

    void removeEdge(int fromVertex, int toVertex) {
        adjMatrix[fromVertex][toVertex] = false;
    }

    Transpose transpose() {
        Transpose transposedGraph(numVertices);
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                if (adjMatrix[i][j]) {
                    transposedGraph.addEdge(j, i);
                }
            }
        }
        return transposedGraph;
    }
```

You should see the following output:

```
Initial Graph:
0 1 1 0
0 0 0 1
0 0 0 1
0 0 0 0

Transpose of Graph:
0 0 0 0
1 0 0 0
1 0 0 0
0 1 1 0
```

We have seen how to create weighted, unweighted, directed, and undirected graphs. We also represented these graphs with either an adjacency list or an adjacency matrix. This sets the foundation for the basic understanding of a graph. From here, we are going to see how we can use this knowledge to traverse a graph, find the shortest path, test connectivity, cycle detection, etc. These topics will be covered in future assignments.

Formative Assessment 1

Formative Assessment 2