Learning Objectives

Learners will be able to...

- **Implement DFS and BFS in C++**
- **Retrieve one vertex or all vertices**
- **Check for the presence of a vertex**
- **Analyze traversal efficiency and applications**

Introduction to Graph Traversal and Operations

Graph Traversal: An Overview

Welcome to our next phase in exploring graph theory: **Graph Traversal and Operations**. Graph traversal is a cornerstone concept in computer science, essential for navigating and manipulating complex data structures represented as graphs. This unit will provide a comprehensive understanding of fundamental traversal algorithms and key graph operations.

Graph traversal refers to the process of visiting, examining, and/or updating each vertex in a graph. It's a critical operation for numerous applications, from network analysis to algorithm design. The two primary traversal techniques we will focus on are:

- **Depth-First Search (DFS):** DFS explores as far as possible along each branch before backtracking. It's akin to exploring a maze by going down one path until you can't go any further, then backtracking to explore different paths.
- **Breadth-First Search (BFS):** BFS explores all neighbors at the present depth before moving on to the nodes at the next depth level. This is similar to exploring all paths that diverge from a single point before moving further.

Key Graph Operations

In addition to traversal, we will learn essential operations that allow interaction and manipulation of graphs:

- **Get Vertex (`getVertex`):** Retrieve specific vertex details from the graph using a unique identifier or key.
- **List All Vertices (`getVertices`):** Return a comprehensive list of all the vertices in the graph, providing an overview of the graph's structure.
- **Vertex Presence Check (`isInGraph`):** A method to verify if a certain vertex exists within the graph, crucial for validating inputs and conditions within graph algorithms.

Practical Implementation

We will not only discuss the theoretical aspects of these traversal methods and operations but also implement them in C++. This practical approach helps in solidifying the understanding of graph traversal and manipulation techniques.

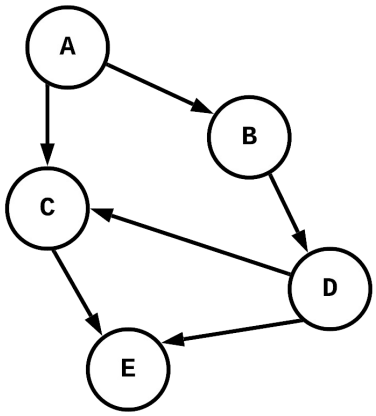
- **DFS and BFS Implementation:** We'll code these traversal algorithms in C++, demonstrating how they operate on a graph data structure.
- **Graph Operations:** Develop method in C++ to perform operations like retrieving and listing vertices, enhancing your ability to work with and understand complex graph structures.

By the end of this assignment, you will have a solid grasp of graph traversal techniques and be able to implement essential graph operations. These skills are vital for further exploration into advanced graph algorithms and their real-world applications, such as route planning, social network analysis, and optimization problems.

Implementing Depth-First Search

DFS

We will implement the Depth-First Search (DFS) algorithm in C++, starting with creating a simple graph structure. We'll then apply DFS to traverse the graph. Our example graph for this exercise is as follows:



Implementing a Graph with DFS

We will start by creating the `Graph` class that uses an `unordered_map` for an adjacency list. The `adjList` attribute represents the key-value pairs of vertices (the key) and a list of their connections (value). In the constructor, initialize an empty unordered_map.

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <unordered_set>
#include <string>

class Graph {
private:
    std::unordered_map<std::string, std::vector<std::string>>
        adjList;

public:
    Graph() {
        adjList = std::unordered_map<std::string,
        std::vector<std::string>>();
    }
};
```

As we have previously seen, we are going to need an `addVertex` method that takes a string representing a vertex. If this string is not in the `unordered_map`, add it to the map along with an empty vector.

```
    void addVertex(const std::string& label) {
        adjList.emplace(label, std::vector<std::string>());
    }
```

We also need to be able to add an edge to the graph. Create the `addEdge` method which takes two strings representing `vertices`. Access the list of neighbors and add the vertex to the list. This is a directed graph, so we only need to perform this operation once.

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

Depth-first search explores as far as possible along each branch before backtracking. We will do this recursively. Start by creating the public `dfs` method that takes a string representing the starting vertex. Create an `unordered_set` data structure that will contain all of the vertices visited during traversal.

```
    void dfs(const std::string& start) {
        std::unordered_set<std::string> visited;
        dfs(start, visited);
    }
```

Next, overload the `dfs` method to be a private, recursive method. This method takes a string representing a vertex as well as the set of visited vertices. Start by checking if the given vertex is already in the set. If so, exit the method. This serves as our base case.

```
private:
    void dfs(const std::string& vertex,
        std::unordered_set<std::string>& visited) {
        if (visited.find(vertex) != visited.end()) {
            return;
        }
    }
```

Otherwise, add the given vertex to the set. Then print the vertex followed by a space.

```
    void dfs(const std::string& vertex,
        std::unordered_set<std::string>& visited) {
        if (visited.find(vertex) != visited.end()) {
            return;
        }

        visited.insert(vertex);
        std::cout << vertex << " ";
    }
```

Generate a list of neighbors for the given vertex. If the list is not empty, iterate over it. Call the recursive `dfs` method and pass it the next neighbor and the set of visited vertices.

```
    void dfs(const std::string& vertex,
        std::unordered_set<std::string>& visited) {
        if (visited.find(vertex) != visited.end()) {
            return;
        }

        visited.insert(vertex);
        std::cout << vertex << " ";

        const std::vector<std::string>& neighbors =
        adjList[vertex];
        for (const std::string& next : neighbors) {
            dfs(next, visited);
        }
    }
```

Testing the Code

To test the code, create the main method. Instantiate a Graph object and then add the vertices `A`, `B`, `C`, `D`, and `E`. Next, add the edges `AC`, `AB`, `CE`, `BD`, `DC`, and `DE`. Finally, start at vertex `A` and do a depth-first search on the graph. Running this will output the vertices in the order they are visited using `DFS`, starting from vertex `A`. The exact order may vary depending on how the `vertices` are stored in the adjacency list.

```
int main() {
    Graph graph;
    std::string vertices[] = {"A", "B", "C", "D", "E"};

    for (const std::string& vertex : vertices) {
        graph.addVertex(vertex);
    }

    graph.addEdge("A", "C");
    graph.addEdge("A", "B");
    graph.addEdge("C", "E");
    graph.addEdge("B", "D");
    graph.addEdge("D", "C");
    graph.addEdge("D", "E");

    std::cout << "DFS Traversal starting from vertex A:" <<
        std::endl;
    graph.dfs("A");

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <unordered_set>
#include <string>

class Graph {
private:
    std::unordered_map<std::string,
        std::vector<std::string>> adjList;

public:
    Graph() {
        adjList = std::unordered_map<std::string,
        std::vector<std::string>>();
    }

    void addVertex(const std::string& label) {
        adjList.emplace(label, std::vector<std::string>());
    }

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

    void dfs(const std::string& start) {
        std::unordered_set<std::string> visited;
        dfs(start, visited);
```

```
    }

private:
    void dfs(const std::string& vertex,
        std::unordered_set<std::string>& visited) {
        if (visited.find(vertex) != visited.end()) {
            return;
        }

        visited.insert(vertex);
        std::cout << vertex << " ";

        const std::vector<std::string>& neighbors =
        adjList[vertex];
        for (const std::string& next : neighbors) {
            dfs(next, visited);
        }
    }
};

int main() {
    Graph graph;
    std::string vertices[] = {"A", "B", "C", "D", "E"};

    for (const std::string& vertex : vertices) {
        graph.addVertex(vertex);
    }

    graph.addEdge("A", "C");
    graph.addEdge("A", "B");
    graph.addEdge("C", "E");
    graph.addEdge("B", "D");
    graph.addEdge("D", "C");
    graph.addEdge("D", "E");

    std::cout << "DFS Traversal starting from vertex A:" <<
        std::endl;
    graph.dfs("A");

    return 0;
}
```

You should see the following output (remember the exact order may be different):

```
DFS Traversal starting from vertex A:
A C E B D
```

challenge

Try this variation

Change the addEdge method so the graph becomes undirected. Run the program again and notice how the output changes.

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

▼ **What is happening?**

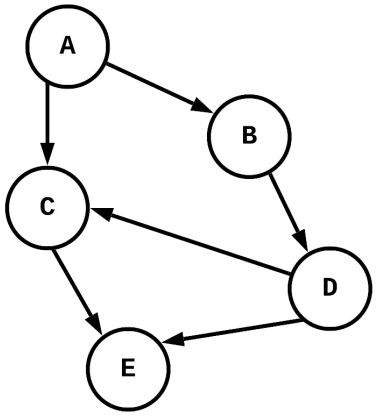
> The traversal path goes from `A C E B D` to `A C E D B`. In an undirected graph, vertex `D` is now a neighbor to vertex `E`, so the traversal goes to `D` before `B`. In a directed graph, you must traverse `B` before you get to `D`.

This implementation of DFS provides a foundation for traversing graphs and can be extended or modified for various applications, such as searching for a specific element, pathfinding, or exploring connected components. DFS is a versatile tool and forms the basis of many more complex algorithms in graph theory.

Implementing Breadth-First Search

BFS

Now we will focus on implementing the breadth-first search (BFS) algorithm in C++. Similar to the previous DFS implementation, we will start by creating a simple graph structure. Our example graph for BFS will be the same as before:



Implementing a Graph with BFS

We will start off with an undirected graph represented by an adjacency list using `unordered_map` from the C++ STL. The graph includes methods to add vertices and edges. By now, you should be familiar with this basic structure.

Graph Class Definition

```cpp
#include <iostream>
#include <unordered_map>
#include <list>
#include <queue>
#include <unordered_set>
#include <string>

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

public:
    void addVertex(const std::string& label) {
        adjList[label]; // Initialize an empty list for this
        vertex.
    }

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

Breadth-First Search explores each level of a graph before moving to the next level. It utilizes a queue to keep track of the next set of `vertices` to visit.

Start by creating the `bfs` method. We will not use a recursive approach for this traversal. This method takes a string representing the starting node. Just as with `DFS`, create a set to keep track of the visited nodes. We also need to create a queue. Add the starting vertex to the set and queue.

```
void bfs(const std::string& start) {
    std::unordered_set<std::string> visited;
    std::queue<std::string> queue;

    queue.push(start);
    visited.insert(start);

    while (!queue.empty()) {
        std::string vertex = queue.front();
        queue.pop();
        std::cout << vertex << " ";

        // Get the neighbors of the current vertex
        const auto& neighbors = adjList[vertex];
        for (const auto& next : neighbors) {
            if (visited.find(next) == visited.end()) {
                visited.insert(next);
                queue.push(next);
            }
        }
    }
}
```

Testing the Code

Now, let's test our `BFS` implementation in the main method. Instantiate a new Graph object and add the vertices `A`, `B`, `C`, `D`, and `E`. Then add the edges `AC`, `AB`, `CE`, `BD`, `DC`, and `DE`. Then print the breadth-first traversal starting with `A`. The output should reflect the level-by-level traversal characteristic of `BFS`.

```
int main() {
    Graph graph;
    std::string vertices[] = {"A", "B", "C", "D", "E"};

    for (const auto& vertex : vertices) {
        graph.addVertex(vertex);
    }

    graph.addEdge("A", "C");
    graph.addEdge("A", "B");
    graph.addEdge("C", "E");
    graph.addEdge("B", "D");
    graph.addEdge("D", "C");
    graph.addEdge("D", "E");

    std::cout << "BFS Traversal starting from vertex A:" <<
        std::endl;
    graph.bfs("A");

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <queue>
#include <unordered_set>
#include <string>

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

public:
    void addVertex(const std::string& label) {
        adjList[label]; // Initialize an empty list for
        this vertex.
    }

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

    void bfs(const std::string& start) {
        std::unordered_set<std::string> visited;
        std::queue<std::string> queue;

        queue.push(start);
        visited.insert(start);

        while (!queue.empty()) {
            std::string vertex = queue.front();
            queue.pop();
            std::cout << vertex << " ";

            // Get the neighbors of the current vertex
            const auto& neighbors = adjList[vertex];
            for (const auto& next : neighbors) {
                if (visited.find(next) == visited.end()) {
                    visited.insert(next);
                    queue.push(next);
                }
            }
        }
    }
};


int main() {
    Graph graph;
    std::string vertices[] = {"A", "B", "C", "D", "E"};

    for (const auto& vertex : vertices) {
```

```
        graph.addVertex(vertex);
    }

    graph.addEdge("A", "C");
    graph.addEdge("A", "B");
    graph.addEdge("C", "E");
    graph.addEdge("B", "D");
    graph.addEdge("D", "C");
    graph.addEdge("D", "E");

    std::cout << "BFS Traversal starting from vertex A:" <<
        std::endl;
    graph.bfs("A");

    return 0;
}
```

You should see the following output:

```
BFS Traversal starting from vertex A:
A C B E D
```

challenge

Try this variation

Change the direction of the edge BD to become DB. Run the program again and notice how the output changes.

```
graph.addEdge("A", "C");
graph.addEdge("A", "B");
graph.addEdge("C", "E");
graph.addEdge("D", "B");
graph.addEdge("D", "C");
graph.addEdge("D", "E");
```

▼ What is happening?

> The traversal path goes from `A C B E D` to `A C B E`. Vertex `D` is not visited because all edges point from vertex `D` to other vertices. There are no vertices that point to vertex `D`.

BFS is an essential algorithm for traversing graphs. It is especially useful in scenarios requiring the exploration of vertices in the order of their proximity or level. This method is widely used in shortest-path calculations, network broadcasting, and many other practical applications in graph theory.

Vertex Retrieval and Presence Check

Additional Operations

We are going to expand upon our graph knowledge by adding operations to get a single vertex, get all the vertices, and check if a vertex is present in the graph. These operations are fundamental for many graph algorithms and applications.

Instead of a simple string representing a vertex, we are going to use a user-defined class to represent the vertex. Vertices in a graph tend to be complex, containing information and any associated methods.

Coding the Graph

Create the `Graph` class. We are also going to need the `Airport` class, which has two string attributes `code` and `name`. The constructor takes two strings representing the airport code and official name and sets them to their respective attributes. We also need to override the `toString` method (in C++ it will be the `operator<<`) so that printing an `Airport` object results in the airport code, a dash, and the official name.

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>

class Graph {
public:
    // Make the Airport class public
    class Airport {
    public:
        std::string code;
        std::string name;

        Airport(const std::string& code, const std::string&
        name) : code(code), name(name) {}

        friend std::ostream& operator<<(std::ostream& os, const
        Airport& arpt) {
            os << arpt.code << " - " << arpt.name;
            return os;
        }
    };


};
```

We are going to create two hash maps for this graph. The first graph is going to be an adjacency list that will manage all of the connections in the graph. The key is going to be a string (the airport code) and the value will be a list of strings (connected airport codes). The second hash map manages the Airport objects. The key for this hash map is a string (the airport code) and the value is an Airport object. We are also going to add the default constructor for the `Graph` class.

▼ **Why two hash maps?**

> We are using two different hash maps so we can separate responsibilities. The `adjList` hash map only has to worry about connections between vertices, while the `airports` hash map only has to worry about storing `Airport` objects. Combining everything into a single hash map adds complexity, which can make your code harder to understand. Keeping two distinct things separate is a common design technique.

```
private:
    // Adjacency list to manage connections
    std::unordered_map<std::string, std::vector<std::string>>
        adjList;
    // Hashmap to store Airport objects
    std::unordered_map<std::string, Airport> airports;

public:
    Graph() = default;
```

Create the `addVertex` method. This public method takes two strings representing the airport code and official name. Instantiate an `Airport` object with the given strings. Add the airport code and `Airport` object as a key-value pair to the airports hash map. Then add the airport code and empty vector to the `adjList` hash map.

```
    // Add vertex
    void addVertex(const std::string& code, const std::string&
        name) {
        airports.emplace(code, Airport(code, name));
        adjList.emplace(code, std::vector<std::string>());
    }
```

Working with Vertices

Now that the Graph class has a basic working structure, we are now going to add methods for manipulating `vertices`. Start with the `getVertex` method. This public method takes a string (the airport code) and returns a pointer to an `Airport` object associated with that code. If the airport code is not in the graph, the method will return `nullptr`.

```
    // Get a vertex (airport) by its code
    const Airport* getVertex(const std::string& code) const {
        auto it = airports.find(code);
        if (it != airports.end()) {
            return &it->second;
        }
        return nullptr;
    }
```

If we want all of the `vertices` in the graph, then we need to create the `getVertices` method. This public method has no parameters, and it returns a vector of `Airport` objects. Return a vector made up of the values in the airports hash map.

```
    // Get all vertices (airports)
    std::vector<Airport> getVertices() const {
        std::vector<Airport> vertices;
        for (const auto& pair : airports) {
            vertices.push_back(pair.second);
        }
        return vertices;
    }
```

Finally, we want to be able to check if a vertex is in the graph. Create the `isInGraph` method. This public method takes a string representing an airport code. Return true if this vertex is in the graph, otherwise return false. Because we are using the `unordered_map` container, we can use the built-in find method.

```
    // Check if vertex is in the graph
    bool isInGraph(const std::string& code) const {
        return airports.find(code) != airports.end();
    }
```

Testing the Code

To test these vertex operations, create the main function which instantiates a Graph object. Add `vertices` (airports) to the graph for Istanbul, London, Paris, Frankfurt, and Boston. Use the `isInGraph` method to search for airports with the `"CDG"` and `"DFW"` codes. Then use the getVertex method to retrieve the Airport objects for airports with the codes `"BOS"` and `"LAX"`. Lastly, print out all of the `vertices` in the graph.

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

    // Adding vertices
    graph.addVertex("IST", "Istanbul Airport");
    graph.addVertex("LHR", "London Heathrow Airport");
    graph.addVertex("CDG", "Charles de Gaulle Airport");
    graph.addVertex("FRA", "Frankfurt Airport");
    graph.addVertex("BOS", "Logan International Airport");

    // Checking if vertices exist in the graph
    std::string searchCode = "CDG";
    std::cout << searchCode << " in the graph: " <<
        (graph.isInGraph(searchCode) ? "true" : "false") <<
        std::endl;

    searchCode = "DFW";
    std::cout << searchCode << " in the graph: " <<
        (graph.isInGraph(searchCode) ? "true" : "false") <<
        std::endl;

    // Retrieve and print a specific vertex
    const Graph::Airport* bosAirport = graph.getVertex("BOS");
    if (bosAirport) {
        std::cout << *bosAirport << std::endl;
    } else {
        std::cout << "null" << std::endl;
    }

    const Graph::Airport* laxAirport = graph.getVertex("LAX");
    if (laxAirport) {
        std::cout << *laxAirport << std::endl;
    } else {
        std::cout << "null" << std::endl;
    }

    // Print all vertices
    std::vector<Graph::Airport> vertices = graph.getVertices();
    std::cout << "[";
    bool first = true;
    for (const auto& arpt : vertices) {
        if (!first) {
            std::cout << ", ";
        }
        std::cout << arpt;
        first = false;
    }
    std::cout << "]" << std::endl;

    return 0;

}
```

▼ Code

> Your code should look like this:

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>

class Graph {
public:
    // Make the Airport class public
    class Airport {
    public:
        std::string code;
        std::string name;

        Airport(const std::string& code, const std::string&
        name) : code(code), name(name) {}

        friend std::ostream& operator<<(std::ostream& os,
        const Airport& arpt) {
            os << arpt.code << " - " << arpt.name;
            return os;
        }
    };

private:
    // Adjacency list to manage connections
    std::unordered_map<std::string,
        std::vector<std::string>> adjList;
    // Hashmap to store Airport objects
    std::unordered_map<std::string, Airport> airports;

public:
    Graph() = default;

    // Add vertex
    void addVertex(const std::string& code, const
        std::string& name) {
        airports.emplace(code, Airport(code, name));
        adjList.emplace(code, std::vector<std::string>());
    }

    // Get a vertex (airport) by its code
    const Airport* getVertex(const std::string& code) const
        {
        auto it = airports.find(code);
        if (it != airports.end()) {
            return &it->second;
        }
        return nullptr;
    }

    // Get all vertices (airports)
    std::vector<Airport> getVertices() const {
        std::vector<Airport> vertices;
        for (const auto& pair : airports) {
            vertices.push_back(pair.second);
        }
        return vertices;
    }
```

```
    // Check if vertex is in the graph
    bool isInGraph(const std::string& code) const {
        return airports.find(code) != airports.end();
    }

};

int main() {
    Graph graph;

    // Adding vertices
    graph.addVertex("IST", "Istanbul Airport");
    graph.addVertex("LHR", "London Heathrow Airport");
    graph.addVertex("CDG", "Charles de Gaulle Airport");
    graph.addVertex("FRA", "Frankfurt Airport");
    graph.addVertex("BOS", "Logan International Airport");

    // Checking if vertices exist in the graph
    std::string searchCode = "CDG";
    std::cout << searchCode << " in the graph: " <<
        (graph.isInGraph(searchCode) ? "true" : "false") <<
        std::endl;

    searchCode = "DFW";
    std::cout << searchCode << " in the graph: " <<
        (graph.isInGraph(searchCode) ? "true" : "false") <<
        std::endl;

    // Retrieve and print a specific vertex
    const Graph::Airport* bosAirport =
        graph.getVertex("BOS");
    if (bosAirport) {
        std::cout << *bosAirport << std::endl;
    } else {
        std::cout << "null" << std::endl;
    }

    const Graph::Airport* laxAirport =
        graph.getVertex("LAX");
    if (laxAirport) {
        std::cout << *laxAirport << std::endl;
    } else {
        std::cout << "null" << std::endl;
    }

    // Print all vertices
    std::vector<Graph::Airport> vertices =
        graph.getVertices();
    std::cout << "[";
    bool first = true;
    for (const auto& arpt : vertices) {
        if (!first) {
            std::cout << ", ";
        }
        std::cout << arpt;
        first = false;
    }
    std::cout << "]" << std::endl;
```

```
        return 0;

    }
```

You should see the following output:

```
CDG in the graph: true
DFW in the graph: false
BOS - Logan International Airport
null
[BOS - Logan International Airport, FRA - Frankfurt Airport, CDG
        - Charles de Gaulle Airport, LHR - London Heathrow
        Airport, IST - Istanbul Airport]
```

These additional methods enhance the functionality of our graph, allowing for more detailed interrogation and manipulation of the structure. Being able to retrieve a single vertex or all vertices and check for the existence of a specific vertex are operations that form the basis for more complex graph algorithms and applications, such as network analysis, graph coloring, and finding subgraphs.

Complexity Analysis in Graph Operations

Analysis of Graph Operations

When working with graphs, understanding the complexity of various operations is crucial. Complexity analysis provides insight into the efficiency and scalability of graph algorithms, helping to evaluate their performance in different scenarios.

Graphs can vary significantly in size and structure, from a few nodes and edges to millions of them. The efficiency of operations on these structures largely depends on the number of vertices (V) and edges (E) in the graph.

Time Complexity

Time complexity refers to the amount of time an algorithm takes to complete based on the size of the input graph.

- **Traversal Algorithms (DFS and BFS):** The time complexity for both DFS and BFS is $O(V + E)$ in the case of an adjacency list representation. This is because every vertex and every edge will be explored in the worst-case scenario.
- **Adding/Removing Vertices and Edges:** For an adjacency list, adding a vertex is $O(1)$, while adding an edge is $O(1)$ for undirected graphs and $O(E)$ for checking duplicates. Removing a vertex is $O(V + E)$ since it involves searching and updating edges, and removing an edge is $O(E)$.

Space Complexity

Space complexity considers the amount of memory an algorithm uses in relation to the size of the input graph.

- **Graph Representation:** In an adjacency list, the space complexity is $O(V + E)$ since we store a list of edges for each vertex. For adjacency matrices, it's $O(V^2)$, as we maintain a two-dimensional array for vertices.
- **Traversal Algorithms:** Both DFS and BFS have a space complexity of $O(V)$, as they may store all vertices in the worst case (for visited nodes, stack in DFS, or queue in BFS).

Factors Influencing Complexity

- **Graph Representation:** Choosing between an adjacency list and an adjacency matrix can significantly affect complexity. Adjacency lists are generally more space-efficient for sparse graphs, while adjacency matrices can provide quicker access for dense graphs.
- **Graph Type:** Directed vs. undirected graphs or weighted vs. unweighted graphs can influence complexity, especially in edge-related operations.
- **Operation Nature:** Certain operations, like searching for an edge in a graph, can vary in complexity based on the representation and the graph's structure.

Understanding complexity is vital for:

- Selecting the right graph representation and algorithm based on the specific requirements and size of the data.
- Optimizing graph algorithms for performance, especially for large-scale applications like social networks or transportation systems.
- Balancing between time and space efficiency, crucial in resource-constrained environments like embedded systems or real-time applications.

By comprehensively analyzing and understanding the complexity of graph operations, we can make informed decisions that lead to efficient and scalable solutions. In our next assignments, we will continue to explore advanced graph algorithms and their applications, keeping complexity considerations in mind.

Formative Assessment 1

Formative Assessment 2