Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Lab Report
On

“Implementing Non-linear Data Structures - Graph”

[Code No.: COMP 202]

Submitted by

Hridayanshu Raj Acharya (1)

Sankalp Acharya (2)

UNG-CE (II/I)

Submitted to

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date: 2024/07/13

Lab Report 5

Graph

Graphs are non-linear data structures consisting of a finite set of vertices (or nodes) and a set of edges connecting these vertices. Each edge connects a pair of vertices, which can represent a wide variety of relationships in different domains such as social networks, transportation systems, and more. This report aims to demonstrate the functionality and applications of graph data structures through code demonstrations.

Operations

The major operations performed in a graph data structure are as follows:

1. **isEmpty()**: Returns true if the graph is empty, and false otherwise.
2. **isDirected()**: Returns true if the graph is directed, and false otherwise.
3. **addVertex(newVertex)**: Inserts a new vertex to the graph.
4. **addEdge(vertex1, vertex2)**: Adds an edge from vertex1 to vertex2.
5. **removeVertex(vertexToRemove)**: Removes a vertex from the graph.
6. **removeEdge(vertex1, vertex2)**: Removes an edge from the graph.
7. **numVertices()**: Returns the number of vertices in the graph.
8. **numEdges()**: Returns the number of edges in the graph.
9. **indegree(vertex)**: Returns the indegree of a vertex.
10. **outdegree(vertex)**: Returns the outdegree of a vertex.
11. **degree(vertex)**: Returns the degree of a vertex.
12. **neighbours(vertex)**: Returns the neighbours of a vertex.
13. **neighbour(vertex1, vertex2)**: Returns true if vertex2 is a neighbour of vertex1.

Code

Github Link: https://github.com/hridayanshu236/CE2022_Lab_1_2

Functions:

Functions used in the following program are:-

Class Graph

```
class Graph {
private:
    std::vector<std::vector<int>> adjMatrix;
    int vertexCount;
    int edgeCount;
    bool directed;

public:
    Graph(bool isDirected = false);
    bool isEmpty() const;
    bool isDirected() const;
    void addVertex();
    void addEdge(int vertex1, int vertex2);
    void removeVertex(int vertex);
    void removeEdge(int vertex1, int vertex2);
    int numVertices() const;
    int numEdges() const;
    int indegree(int vertex) const;
    int outdegree(int vertex) const;
    int degree(int vertex) const;
    std::vector<int> neighbours(int vertex) const;
    bool neighbour(int vertex1, int vertex2) const;
};
```

1. **isEmpty():**

```
bool Graph::isEmpty() const {
    return vertexCount == 0;
}
```

2. **isDirected():**

```
      bool Graph::isDirected() const {
    return directed;
}
```

3. **addVertex():**

```cpp
void Graph::addVertex() {
    vertexCount++;
    for (auto& row : adjMatrix) {
        row.push_back(0);
    }
    adjMatrix.push_back(std::vector<int>(vertexCount, 0));
}
```

4. **addEdge(int vertex1, int vertex2):**

```cpp
void Graph::addEdge(int vertex1, int vertex2) {
    if (vertex1 >= vertexCount || vertex2 >= vertexCount || vertex1 < 0 ||
vertex2 < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return;
    }
    if (adjMatrix[vertex1][vertex2] == 0) {
        adjMatrix[vertex1][vertex2] = 1;
        edgeCount++;
        if (!directed) {
            adjMatrix[vertex2][vertex1] = 1;
        }
    }
}
```

5. **removeVertex(int vertex):**

```cpp
void Graph::removeVertex(int vertex) {
    if (vertex >= vertexCount || vertex < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return;
    }
    adjMatrix.erase(adjMatrix.begin() + vertex);
    for (auto& row : adjMatrix) {
        row.erase(row.begin() + vertex);
    }
    vertexCount--;
}
```

6. **removeEdge(int vertex1, int vertex2):**

```cpp
void Graph::removeEdge(int vertex1, int vertex2) {
    if (vertex1 >= vertexCount || vertex2 >= vertexCount || vertex1 < 0 ||
vertex2 < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return;
    }
    if (adjMatrix[vertex1][vertex2] == 1) {
        adjMatrix[vertex1][vertex2] = 0;
        edgeCount--;
        if (!directed) {
            adjMatrix[vertex2][vertex1] = 0;
        }
    }
}
```

7. **numVertices():**

```
int Graph::numVertices() const {
    return vertexCount;
}
```

8. **numEdges():**

```
int Graph::numEdges() const {
    return edgeCount;
}
```

9. **indegree(int vertex):**

```
int Graph::indegree(int vertex) const {
    if (vertex >= vertexCount || vertex < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return -1;
    }
    int inDeg = 0;
    for (int i = 0; i < vertexCount; ++i) {
        inDeg += adjMatrix[i][vertex];
    }
    return inDeg;
}
```

10. **outdegree(int vertex):**

```
int Graph::outdegree(int vertex) const {
    if (vertex >= vertexCount || vertex < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return -1;
    }
    int outDeg = 0;
    for (int j = 0; j < vertexCount; ++j) {
        outDeg += adjMatrix[vertex][j];
    }
    return outDeg;
}
```

11. degree(int vertex):

```cpp
int Graph::degree(int vertex) const {
    if (directed) {
        return indegree(vertex) + outdegree(vertex);
    } else {
        return outdegree(vertex); // Same as indegree in an undirected graph
    }
}
```

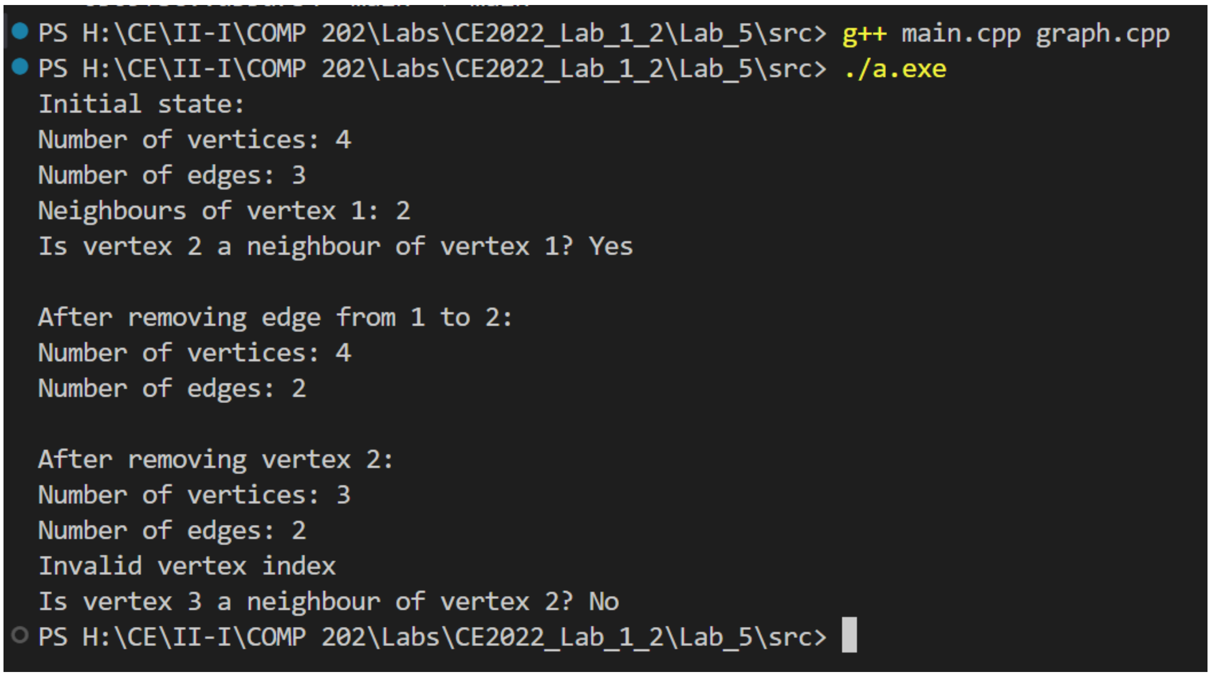
12. neighbours(int vertex):

```cpp
std::vector<int> Graph::neighbours(int vertex) const {
    if (vertex >= vertexCount || vertex < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return std::vector<int>();
    }
    std::vector<int> neighbors;
    for (int i = 0; i < vertexCount; ++i) {
        if (adjMatrix[vertex][i] == 1) {
            neighbors.push_back(i);
        }
    }
    return neighbors;
}
```

13. neighbour(int vertex1, int vertex2):

```cpp
bool Graph::neighbour(int vertex1, int vertex2) const {
    if (vertex1 >= vertexCount || vertex2 >= vertexCount || vertex1 < 0 ||
vertex2 < 0) {
        std::cerr << "Invalid vertex index" << std::endl;
        return false;
    }
    return adjMatrix[vertex1][vertex2] == 1;
}
```

Output:



```
PS H:\CE\II-I\COMP 202\Labs\CE2022_Lab_1_2\Lab_5\src> g++ main.cpp graph.cpp
PS H:\CE\II-I\COMP 202\Labs\CE2022_Lab_1_2\Lab_5\src> ./a.exe
Initial state:
Number of vertices: 4
Number of edges: 3
Neighbours of vertex 1: 2
Is vertex 2 a neighbour of vertex 1? Yes

After removing edge from 1 to 2:
Number of vertices: 4
Number of edges: 2

After removing vertex 2:
Number of vertices: 3
Number of edges: 2
Invalid vertex index
Is vertex 3 a neighbour of vertex 2? No
PS H:\CE\II-I\COMP 202\Labs\CE2022_Lab_1_2\Lab_5\src>
```

Conclusion:

The graph data structure is a versatile and powerful tool for representing relationships between entities. It consists of vertices (nodes) and edges (connections) that can be directed or undirected. Key operations include adding and removing vertices and edges, checking for neighbors, and determining degrees of vertices. Graphs are widely used in various fields, such as computer science, networking, social networks, and more, to model complex systems and solve problems related to connectivity and traversal. The provided implementation and main function illustrate these fundamental operations and their practical applications.