Final Project

Anthony Fratipietro: 40210231
Lauren Lastoria: 40209449

COEN 244
Bahareh Goodarzi
April 2022

Modelling a Graph Application in C++

I. Design Description

Based on the class diagram shown in figure 1, our program deals with **inheritance** between the abstract *Graph* class and the *Directed* and *Undirected* classes. The *Graph* class is the base class as shown by the arrows directing upward from the *Undirected* and *Directed* classes. Furthermore, our program also involves **polymorphism** due to the use of purely virtual functions in the *Graph* class, as indicated by the italicized functions from figure 1. Finally, the last technique included in our program was **operator overloading**. The equality, post and pre-increment, assignment, relational, stream insertion and addition operators were overloaded to facilitate the use of typical operators with the attributes of *Directed* and *Undirected* graph classes.

In addition, the *Vertex* class is a 'part' of the 'aggregate' *Graph* class as demonstrated in the class diagram. It is an aggregation since a graph is composed of several vertices, demonstrating a 'part-whole' relationship. In addition, the vector of *Vertex* objects is an attribute of the *Graph* class, meaning it contains a dynamic array of objects from the *Vertex* class. Also, the *Edge* class is associated with the *Vertex* class via composition. If the *Vertex* object is destroyed, then the *Edge* object is inevitably destroyed. Since the two classes are more closely bound to one another, a composition relationship was considered in our design. Every *Edge* object is composed of a pointer to an ending vertex, while every Vertex object contains a vector of edges (edgeList). Having an edge list for each vertex eliminated the need for a pointer to a starting vertex in the edge class, as the starting vertex can be represented by the current vertex object (pointed to by 'this' pointer). Moreover, an edge list was necessary for an adjacency list representation of a graph, which allows for ease of access to the adjacent vertices of a vertex.

.

II. Member Functions

Add Vertex Function

To add a vertex to the list of vertices, an object of the *Vertex* class must be passed to the function. The program will loop through the list of vertices and search to see if the vertex passed already exists in the list. If the vertex already exists, then a statement will be printed, and the function will return false. Otherwise, the vertex will be added to the end of the list of vertices by using the *push_back()* functions of vectors, which will add the given vertex at the end of the vector. Once this occurs, the statement will be printed that the addition of the vertex was done successfully, and the function will return true. The code for this function is shown in figure 2.1.

Add Edge Function

Similar to the *addVertex(Vertex&)* function, an initial search to verify if the edge already exists is necessary for the "addEdge" function. The user will input two vertices, one for the start of the edge and one for the end of the edge, and these two vertices must be searched within the list of vertices. Furthermore, the edge must be searched as well using the *searchEdge(Vertex&,Vertex&)* function. If the list of vertices contains both given vertices and the edge exists already, then the function will return false, and the error statement defined in the function will be printed. However, if the search functions for both vertices were true and the search edge was false, then the end can be added to the graph. Using vectors, the edge can be added to the list of edges with the *addEdgetoEdgeList(Vertex*,int)* function and then finally incrementing the total weight of the edge list. The key distinction between the addition of an undirected and a directed edge is that an undirected edge must be added in both directions while a directed edge requires only one edge from 'start' to

‘end’. These distinctions can be seen upon comparison of figures 2.2 and 2.3

III. Black Box Testing

When the program is run, the initial menu is displayed to the user where they can input the value corresponding to the functions listed. To create the graph, vertices must be added to the vector list to then create edges from that list of vertices. As shown in figure 3.1, the functions work properly, and the statements are printed onto the terminal when the vertices or edges are added correctly.

For the application of our program, the undirected graph would represent a map of Canada in which we would be able to display paths between Montreal, Toronto, or Ottawa for example. It would represent the paths of the edges added to the graph between different cities and provinces. For directed graphs on the other hand, it would represent followers on a social media platform. For example, Bert follows Sara, who follows another user and so on. It would display the paths of users following other users.

In the screenshot of Figure 3.2, the user is creating the undirected graph for a map between Montreal, Toronto, and Ottawa. As depicted in the image, the program generates the correct output for the valid inputs given by the user. The user can successfully add vertices and edges without the program crashing or disrupting.

IV. Figures

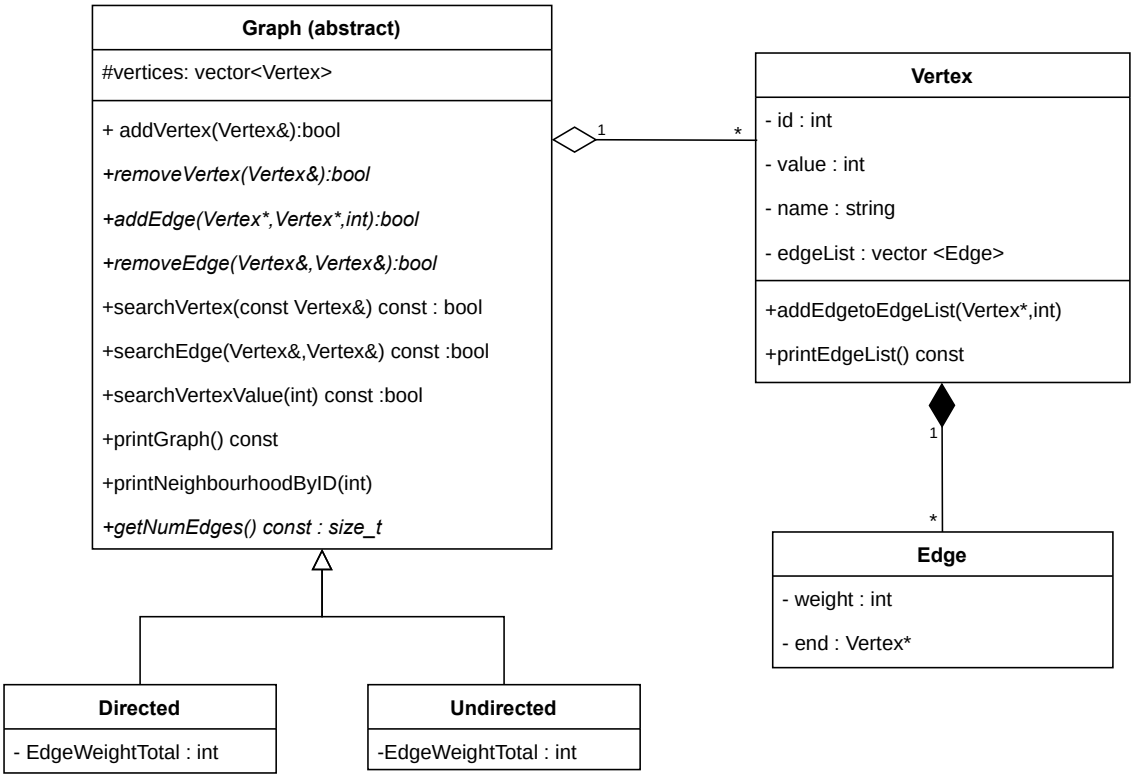

Fig. 1. Class Diagram following UML conventions

```
bool Graph::addVertex(Vertex& v) {
    // check if the vertex already exists in the graph
    bool check = searchVertex(v);
    if (check == true) {
        cout << "Vertex with this ID already exists!" << endl;
        return false;
    }
    // if vertex is not already in graph
    // add vertex to list of vertices
    vertices.push_back(v);
    cout << "New vertex added successfully" << endl;
    return true;
}
```

Fig. 2.1. Code for function to add a vertex to a graph

```
bool Directed::addEdge(Vertex* start,Vertex* end,int weight) {
    // check if vertices are in the graph
    bool check1 = searchVertex(*start);
    bool check2 = searchVertex(*end);
    // check if edge already exists in the graph
    bool check3 = searchEdge(*start,*end);

    if (check1 == true && check2 == true) { // vertices are in the graph
        if (check3 == true) { // edge also exists
            cout << "Edge between " << start->getName() << "(" << start->getID() << ") and
                "
                 << end->getName() << "(" << end->getID() << ") already exist" << endl;
            return false;
        }
        else { // edge does not exist
            for (int i = 0; i < vertices.size(); i++) {
                // add directed edge
                if (vertices.at(i).getName() == start->getName()) {
                    vertices.at(i).addEdgetoEdgeList(end,weight);
                }
            }
            EdgeWeightTotal += weight; // add to total edge weight of graph
            cout << "Edge addition complete" << endl;
            return true;
        }
    }
    else { // one or both vertices are not in the graph
        cout << "Invalid vertices entered" << endl;
        return false;
    }
}
```

Fig. 2.2. Code for function to add an edge in a directed graph

```cpp
bool Undirected::addEdge(Vertex* start, Vertex* end, int weight) {
    // check if vertices are in graph
    bool check1 = searchVertex(*start);
    bool check2 = searchVertex(*end);
    // check if edge already exists in the graph
    bool check3 = searchEdge(*start,*end);

    if (check1 == true && check2 == true) { // vertices are in the graph
        if (check3 == true) { // edge also exists
            cout << "Edge between " << start->getName() << "(" << start->getID() << ") and "
                 << end->getName() << "(" << end->getID() << ") already exist" << endl;
            return false;
        }
        else { // edge does not exist
            for (int i = 0; i < vertices.size(); i++) {
                // add undirected edge
                if (vertices.at(i).getID() == start->getID()) {
                    vertices.at(i).addEdgetoEdgeList(end,weight);
                }
                else if (vertices.at(i).getID() == end->getID()) {
                    vertices.at(i).addEdgetoEdgeList(start,weight);
                }
            }
            EdgeWeightTotal += weight; // add to total edge weight of graph
            cout << "Edge addition complete" << endl;
            return true;
        }
    }
    else { // one or both vertices are not in the graph
        cout << "Invalid vertices entered" << endl;
        return false;
    }
}
```

Fig. 2.3. Code for function to add an edge in an undirected graph

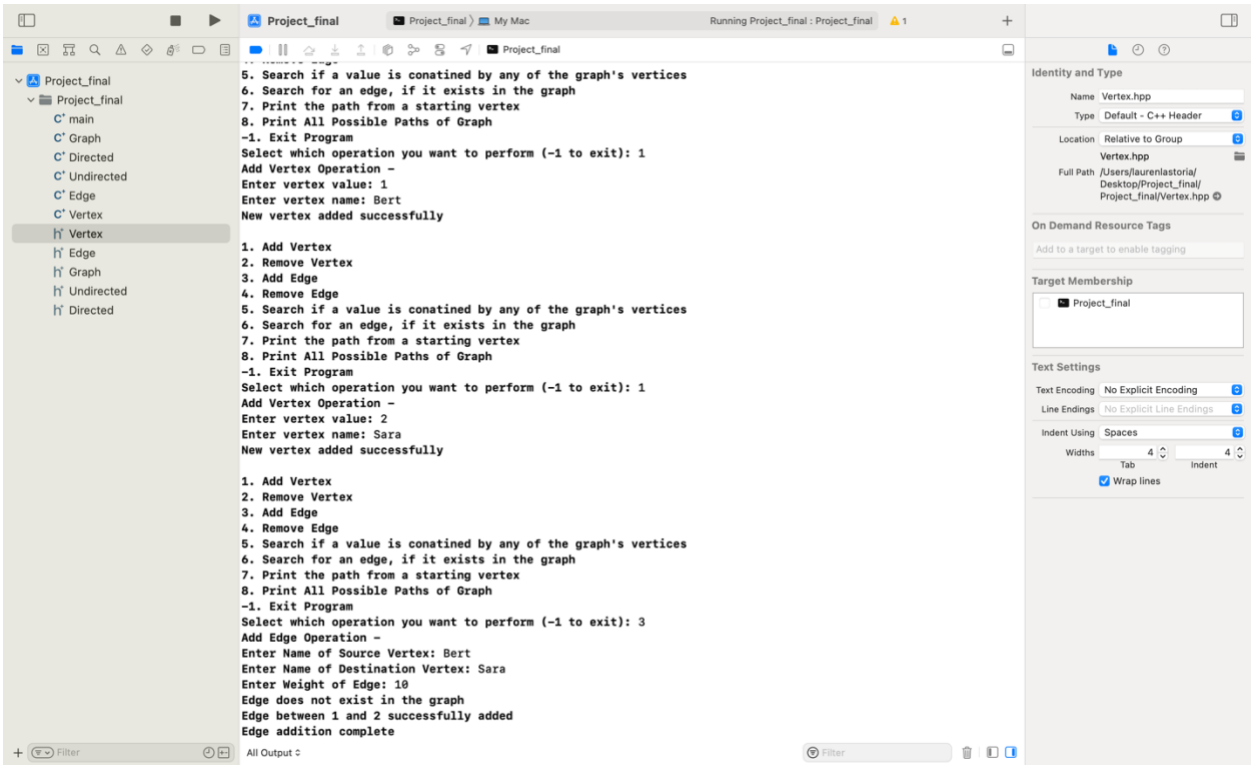

Fig. 3.1. Directed Graph for social media

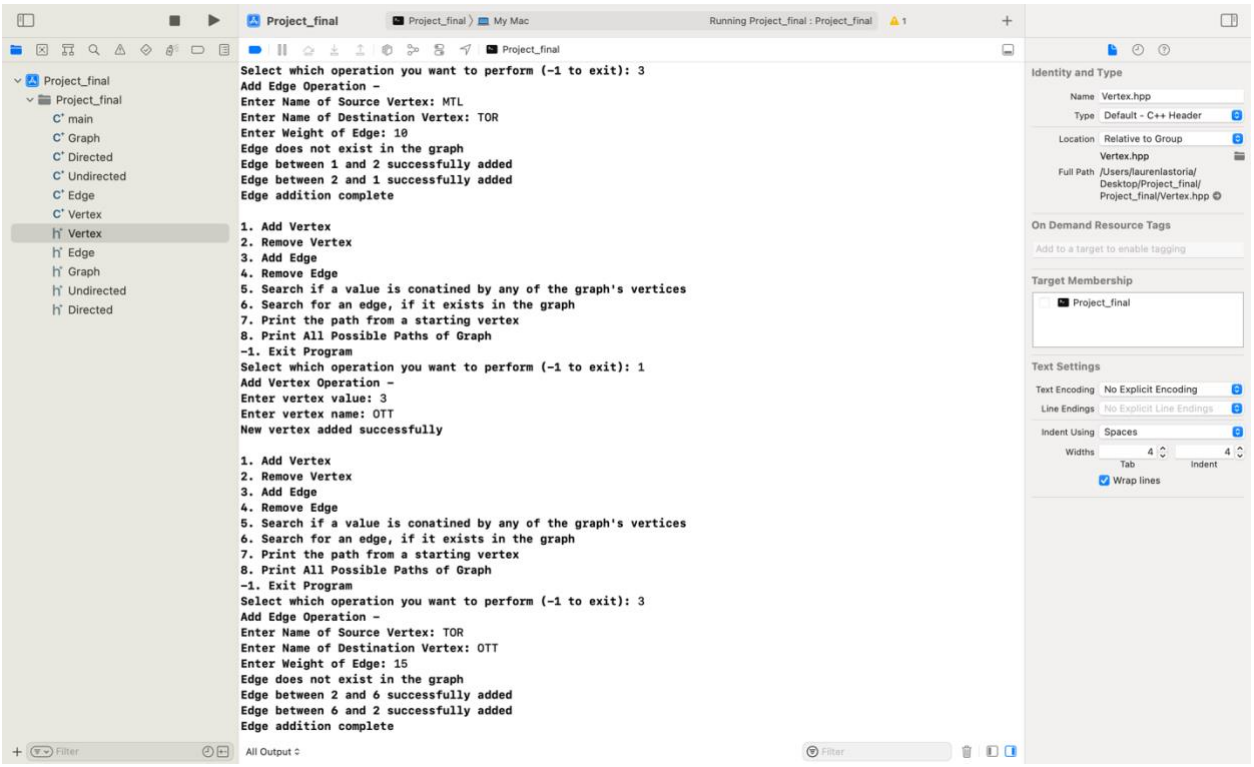

Fig. 3.2. Undirected Graph for Map of Canada