Directed graph with applications

Usman Usman

Gina cody school of Engineering

Concordia University

Montreal, Canada

U_sman@live.concordia.ca

Jean Cedrick Dorelas

Gina cody school of Engineering

Concordia University

Montreal, Canada

jcedrik100@gmail.com

CHAPTER 1

Introduction:

1.1 What is a graph?

A graph is a diagram showing the relation between variable quantities, typically of two variables.

Every graph has a vertex and edge and there are two types of graphs namely;

- Directed graph
- Undirected graph

We decided to write a C++ program to constructs a directed graph for the YUL airport.

1.2 LICENSE

ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO

THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,

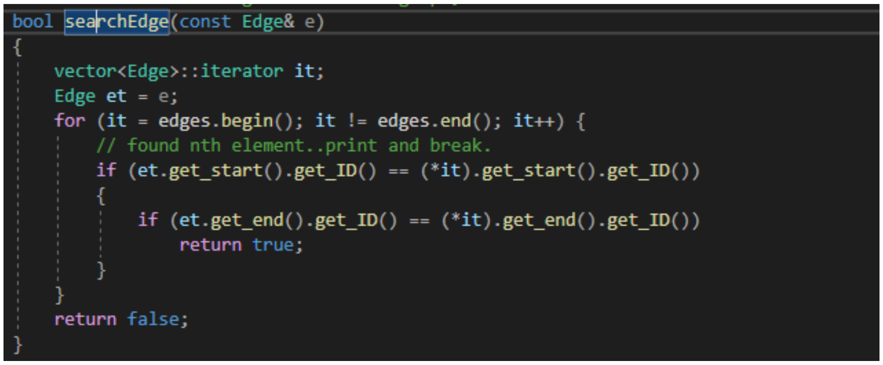
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 2

DESIGN DESCRIPTION

2.1 Search edge

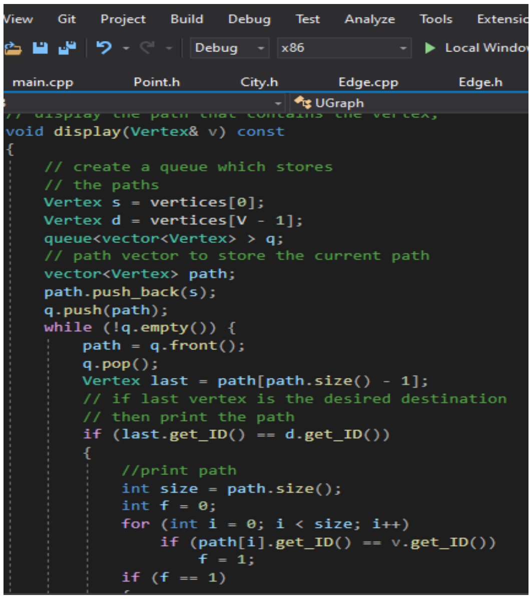
This function searches for an edge in a graph. If found it returns true and if the edge isn’t found, it returns false



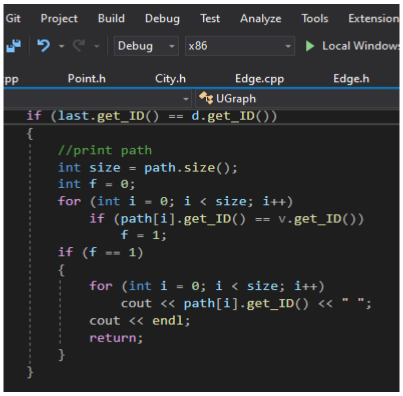
```
bool searchEdge(const Edge& e)
{
    vector<Edge>::iterator it;
    Edge et = e;
    for (it = edges.begin(); it != edges.end(); it++) {
        // found nth element..print and break.
        if (et.get_start().get_ID() == (*it).get_start().get_ID())
        {
            if (et.get_end().get_ID() == (*it).get_end().get_ID())
                return true;
        }
    }
    return false;
}
```

2.2 Display Function

This function displays all the vertices in the graph. The function uses a queue(create a queue which stores the path), vector(to store the current path) and the get_neighbours member function to print the vertices contained in the class.

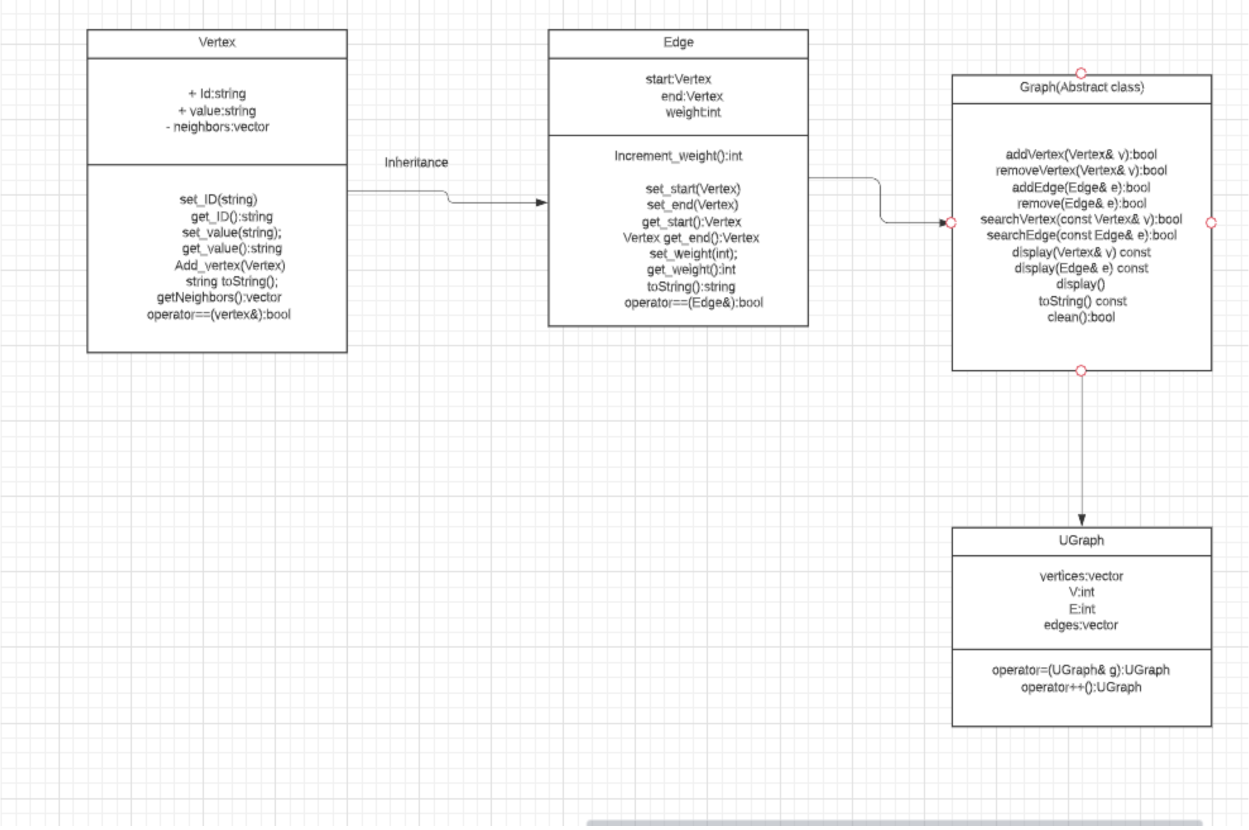


```
// display the path that contains the vertex;
void display(Vertex& v) const
{
    // create a queue which stores
    // the paths
    Vertex s = vertices[0];
    Vertex d = vertices[V - 1];
    queue<vector<Vertex> > q;
    // path vector to store the current path
    vector<Vertex> path;
    path.push_back(s);
    q.push(path);
    while (!q.empty()) {
        path = q.front();
        q.pop();
        Vertex last = path[path.size() - 1];
        // if last vertex is the desired destination
        // then print the path
        if (last.get_ID() == d.get_ID())
        {
            //print path
            int size = path.size();
            int f = 0;
            for (int i = 0; i < size; i++)
                if (path[i].get_ID() == v.get_ID())
                    f = 1;
            if (f == 1)
```



```
if (last.get_ID() == d.get_ID())
{
    //print path
    int size = path.size();
    int f = 0;
    for (int i = 0; i < size; i++)
        if (path[i].get_ID() == v.get_ID())
            f = 1;
    if (f == 1)
    {
        for (int i = 0; i < size; i++)
            cout << path[i].get_ID() << " ";
        cout << endl;
        return;
    }
}
```

2.3 Class Diagram



CHAPTER 3

USAGE

3.1 Operator Overloading

We also used operator overloading techniques in some of the methods we created such as;

- bool Vertex::operator==(Vertex& v): This function is used to know if two vertices have the same id.
- bool Edge::operator==(Edge& e): This function is used to know if two edges have the same starting and ending vertices and weight.

3.2 Inheritance

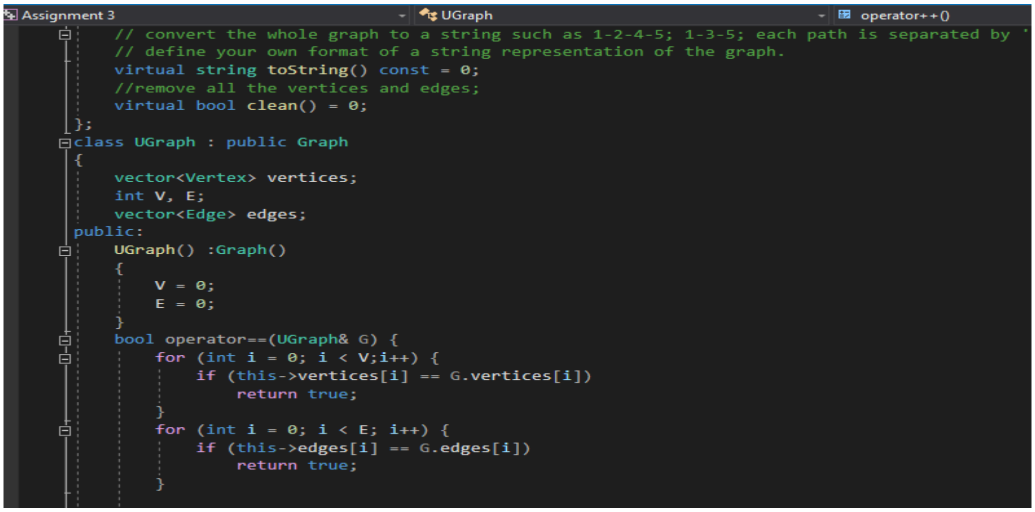
Inheritance technique was used in writing the code. The Edge class inherits the public methods and attributes from the vertex class which is the parent class. The graph class also inherits from the Edge class and the Ugraph class inherits from the Graph class. A visual representation of the inheritance is shown in the class diagram in chapter 2.

3.3 Templates

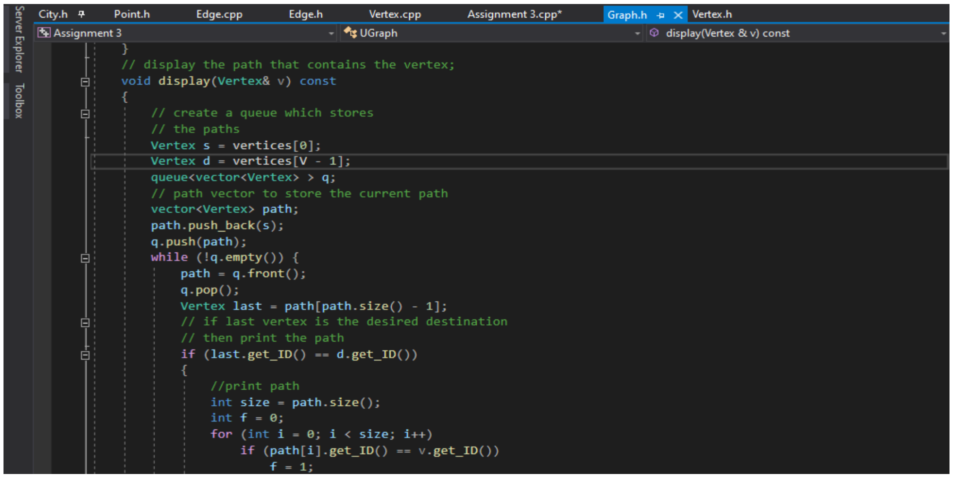
We used some containers from the standard template library such as;

- Queue
- Vector

We used queue and vector in some of the methods in the Ugraph class. We used it to store and display the vertex and edge objects in the undirected graph.



```
    // convert the whole graph to a string such as 1-2-4-5; 1-3-5; each path is separated by '
    // define your own format of a string representation of the graph.
    virtual string toString() const = 0;
    //remove all the vertices and edges;
    virtual bool clean() = 0;
};
class UGraph : public Graph
{
    vector<Vertex> vertices;
    int V, E;
    vector<Edge> edges;
public:
    UGraph() :Graph()
    {
        V = 0;
        E = 0;
    }
    bool operator==(UGraph& G) {
        for (int i = 0; i < V;i++) {
            if (this->vertices[i] == G.vertices[i])
                return true;
        }
        for (int i = 0; i < E; i++) {
            if (this->edges[i] == G.edges[i])
                return true;
        }
```



```cpp
    }
    // display the path that contains the vertex;
    void display(Vertex& v) const
    {
        // create a queue which stores
        // the paths
        Vertex s = vertices[0];
        Vertex d = vertices[V - 1];
        queue<vector<Vertex> > q;
        // path vector to store the current path
        vector<Vertex> path;
        path.push_back(s);
        q.push(path);
        while (!q.empty()) {
            path = q.front();
            q.pop();
            Vertex last = path[path.size() - 1];
            // if last vertex is the desired destination
            // then print the path
            if (last.get_ID() == d.get_ID())
            {
                //print path
                int size = path.size();
                int f = 0;
                for (int i = 0; i < size; i++)
                    if (path[i].get_ID() == v.get_ID())
                        f = 1;
```