COEN-244 Final Project

Jaskirat Kaur
Concordia University
Gina Cody School of Engineering
Montreal, Canada
jaskiratkaur1906@gmail.com

Simren Matharoo
Concordia University
Gina Cody School of Engineering
Montreal, Canada
simren.matharoo@gmail.com

***Abstract*—This document contains the description of the final project for COEN-244 which is the second object-oriented programming class in the course sequence for the Computer Engineering Degree. This report explains the detail of the Family Tree Application that was developed for this project.**

I. Introduction

Graphs are used in many different fields to link together many different types of information. For example, a network of friends can be represented by a graph, a course sequence can be represented by a graph, etc. In this project, a family tree graph was made. This graph was implemented using notions of object-oriented programming. This program was designed to run as an application to depict the different relationships in any given family. While the application itself was not extensive, the driver may be further developed to fit the needs of anyone who wishes to use it. The program will be able to run and compute all the implemented functionalities.

II. Function Descriptions

For this project, eight different functions were needed to be implemented. These functions will be described in this section.

A. A Graph Can Be Empty

The function isEmpty() makes it possible for a graph to be without any edges or vertices. It will verify if that is the case and return a Boolean value of 1 if the graph is empty. This function does so by verifying if the array of vertices and the array of edges are empty. The following figure shows the black box testing of the function.

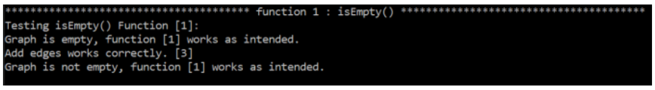

Fig. 1. Function isEmpty() Black Box testing

B. A Graph Can Be Directional or Undirectional

The code of this project was written in such a way that a graph can be directional or unidirectional. In fact, the function isDirectional() was implemented to check if the given graph is directional. If it is, the function will return a Boolean value of 1. The directionality of the graph is verified by checking if every edge that goes from a source vertex to a destination vertex also has an edge that leads from the destination vertex to the source vertex. The following figure shows the black box testing of the function.

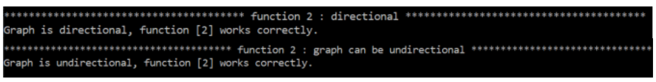

Fig. 2. Function isDirectional() Black Box testing

C. A Graph Can Be Added in Vertices and Edges

This function was implemented as two different functions in the project code. The first one is addVertex(Vertex), which adds a vertex to the array of vertices. Then there is the addEdge(Edge) function which adds an edge to an array of edges. Both of these functions return a Boolean value of 1 if the adds are successful. The following figure shows the black box testing of these two functions.

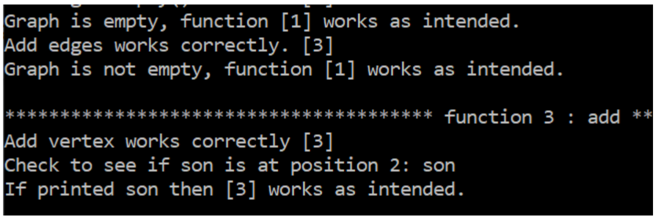

Fig. 3. Functions addEdge and addVertex Black Box testing

D. A Vertex Can Contain Values of Any Type

This function was not implemented as a separate function in the program. In fact, the value of a vertex is given by a string, which can take any value.

E. A Graph Can Be Displayed By Listing All Paths

The function displayGraph() makes it possible to display every path from the top of the graph to the bottom of the graph. The following figure shows the black box testing of the function.

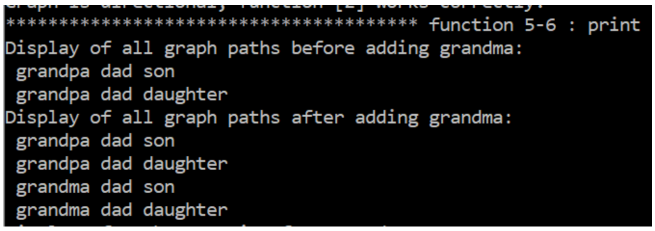

Fig. 4. Function dispayGraph() Black Box testing

F. A Graph Can Be Queried By A Starting Vertex

The function printFromVertex(Vertex) makes it possible to display every path from one vertex to the end of the graph. The following figure shows the black box testing of the function.

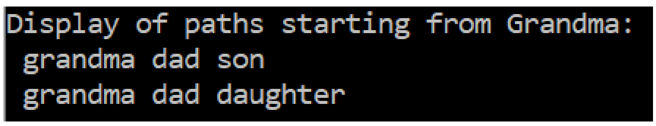

Fig. 5. Function printFromVertex(Vertex) Black Box testing

G. A Graph Can Be Queried By An Edge

The function searchEdge(Edge) makes it possible to verify if a given edge exists in the graph. If it does, the function returns a Boolean value of 1. The following figure shows the black box testing of the function.

```
************************************** function 7 : searchEdge() ****
Looking for edge between son and dad:
dad-son exists, function [7] works as intended.

Looking for edge that does not exist:
random edge does not exist in graph, function [7] works as intended.
```

Fig. 6. Function searchEdge(Edge) Black Box testing

H. A Graph Can Be Queried By A Value

The function searchByVertexValue(string) verifies if a vertex with a given name or value exists in the graph. If it does, the function returns a Boolean value of 1. The following figure shows the black box testing of the function.

```
************************************ function 8 : searchByVertexValue() **
Search function [8] functions correctly, finds value of vertex.
Search function [8] functions correctly, does not find random value.
```

Fig. 7. Function searchByVertexValue(string) Black Box testing

III. Design Description

The following figure shows the UML diagram for the project. It demonstrates the links between each class.

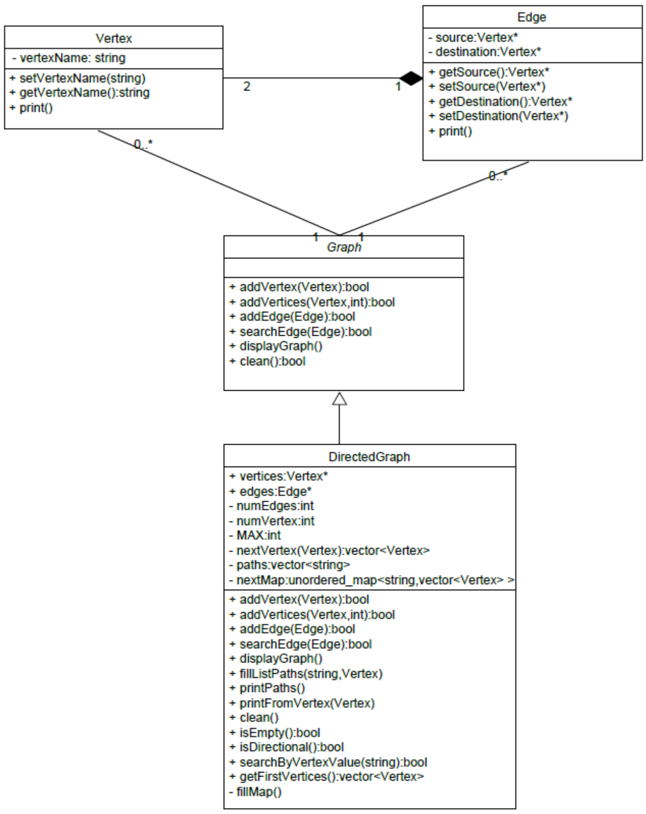

Fig. 8. UML Diagram of Project

IV. Techniques and Methods Description

A. Techniques Used

In this project, four specific techniques were used: inheritance, polymorphism, operator overloading, and exception handling.

i. The first technique utilised in this project is inheritance. This is shown in the UML diagram of all the classes shown in fig. 8. In fact, the class DirectedGraph inherits from the class Graph. This means that every method and attribute in the class Graph is available in the class DirectedGraph. Also, since Graph has access to the public functions of classes Vertex and Edge

ii. The second technique used in this project is polymorphism. This technique is used widely in the program. In fact, all of the functions in the Graph class are pure virtual, meaning that they must be overloaded in the child class DirectedGraph. Also, in both of the classes Edge and Vertex, the method function print() is used. Making print() a virtual function allows it to be used by both class objects. This was also done to facilitate the addition of more code. Should another graph type be needed, it could be implemented without much difficulty because of the polymorphism technique utilised.

iii. The third technique used in this project is operator overloading. In fact, the operator == was overloaded to be able to equalise two edges. As shown in fig 9, if the source and destination of an edge are equal to that of another edge, the operator returns true.

```
bool Edge::operator==(const Edge& e) const
{
    if ((e.source == source) && (e.destination == destination))
        return true;
    else
        return false;
}
```

Fig. 9. Operator== Overload Implementation

iv. The fourth technique used in this project is exception handling. This technique is used at more than one location in the code. It was mostly used to make sure that there were no arrays that were being accessed out of bounds like in the addVertex(Vertex) method shown in fig 10. If an array was accessed out of bounds, an exception would be thrown, and the method would return false.

```cpp
bool DirectedGraph::addVertex(Vertex& v)
{
    // adds a vertex to the array of vert
    try {
        vertices[numVertex] = v;
        numVertex++;
    }
    catch (...) { return false; }

    return true;
}
```

Fig. 10. Exception Handling Example

B. *Methods Description*

Two non-trivial methods that were implied in the project will be explained in this report. These methods are the getFirstVertices() method and the fillListPaths(string,Vertex) method.

i. The first method that will be discussed is the getFirstVertices() method. The goal of this method is to create and return a vector of Vertex objects that are at the top of the graph. In simpler terms, it returns a list of every vertex that does not have a parent.

The function begins by creating a vector of type Vertex. It will then iterate through every vertex in the graph and check if there is an edge that has the current vertex as a destination. If there is, the counter will increase by one. If after iterating through every edge, if the counter is still at 0, the current vertex will be added to the vector.

```cpp
vector<Vertex> DirectedGraph::getFirstVertices()
{
    //get the top of the graph, the vertices that no other vertice has this vertice as a next, and puts it in a vector
    vector<Vertex> tempVertexList;
    for (int i = 0; i < numVertex; i++) {
        int counterOfInstances = 0;
        for (int j = 0; j < numEdges; j++) {
            if (edges[j].getDestination()->getVertexName() == vertices[i].getVertexName()) {
                counterOfInstances++;
            }
        }
        if (counterOfInstances == 0) {
            tempVertexList.push_back(vertices[i]);
        }
    }

    return tempVertexList;
}
```

Fig. 11. getFirstVertices() Function Implementation

ii. The second method that will be discussed in this report is the fillListPaths(string,Vertex) method. This function checks the map to see if there are child vertices to the current vertex and makes a recursion to find all the paths.

The function begins by creating a string variable to hold the names of all the vertices in a given path. The string passed in the parameter will be added to this string, so when the function is first called, an empty string must be put in the parameter. The second parameter determines from which vertex the path will begin at. The first thing that is verified is that there are adjacent vertices to the current vertex. If there are not any, the path will be added to the vector of type string called paths. Otherwise, there will be a recursion that puts the next vertex into the string that will become the path. Therefore, in the end, the vector of paths will be filled with every single path that can be taken from the starting vertex that was inputted in the parameter of the function.

```cpp
void DirectedGraph::fillListPaths(string str, Vertex v)
{
    //string that holds one path at a time
    string tempStr = str;
    tempStr += " " + v.getVertexName();

    //Recursion to get all the paths
    //if the vertice has no next vertice, push string of paths into list of paths
    if (nextMap[v.getVertexName()].empty()) {
        paths.push_back(tempStr);
        return;
    }
    else {
        //if theres a next vertice, recursion to find paths
        int size = nextMap[v.getVertexName()].size();
        for (int i = 0; i < size; i++) {
            fillListPaths(tempStr, nextMap[v.getVertexName()][i]);
        }
    }
}
```

Fig. 12. fillListPaths(string,Vertex) Function Implementation

ACKNOWLEDGMENT

We would like to acknowledge and thank the professor as well as the TAs for being very helpful and informative durin the entirety of the semester. We would also like to take this opportunity to thank the third, unofficial member of our team, Mau. Mau, the cat, has provided countless hours of emotional support during the difficult parts of this project.

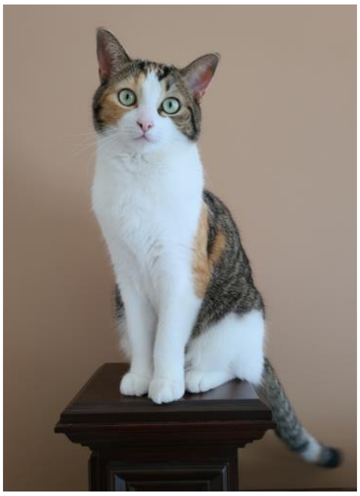

Fig. 13. Mau the Third, Unoffical Member of Our Team