Programming Methodology II - Course Project

Report

presented to

Bahareh Goodarzi

For the class:

COEN 244

By

Justin GOI

40171109

AND

Luat-Dinh NGUYEN

40174891

April, 20th 2021

Concordia University

The main goal of this project is to build a road network for both pedestrians and vehicles. This is done by creating a directed graph and an undirected graph. The content of the graphs are vertices which represent intersections and edges which represent roads. The directed graph is for the vehicles since it indicates the direction of the roads while the undirected graph is aimed at pedestrians. The program uses five different classes to implement this concept.

1- UML Diagram

The UML diagram describing the relationship between the classes is shown in Figure 1. As shown, the Graph class is abstract and it is used as a template for the directed and undirected graphs. The relationship is a one-to-many since all directed and undirected graphs follow that specific graph model. In contrast, each graph, whether directed or undirected, has zero or many vertices and edges.

2- Important Functions

There are four important functions that form the backbone of this road network. The four functions are used for direction. The first function is called findpaths and its purpose is to record all paths from a starting intersection to a destination. The second is called findAllPaths which will output all the possible paths from a particular intersection to a particular destination. There is also another variant of the previous function called printAP. However, this function will output all the possible paths for every permutation of a starting intersection to a destination. Finally, the function printlead will print all the possible paths from a starting intersection that is entered by the user to every possible destination.

2.1- Printpaths Function

To start, the most challenging function to implement was the findpaths((int u, int d, int path[], int& path_index) since its implementation had to be a recursive one. This function is a private function of the directed and undirected classes. The parameters are the starting vertex ID, the ending vertex ID, a vector of type bool which determines whether a vertex has been visited or not, an array of integers named path which holds the paths

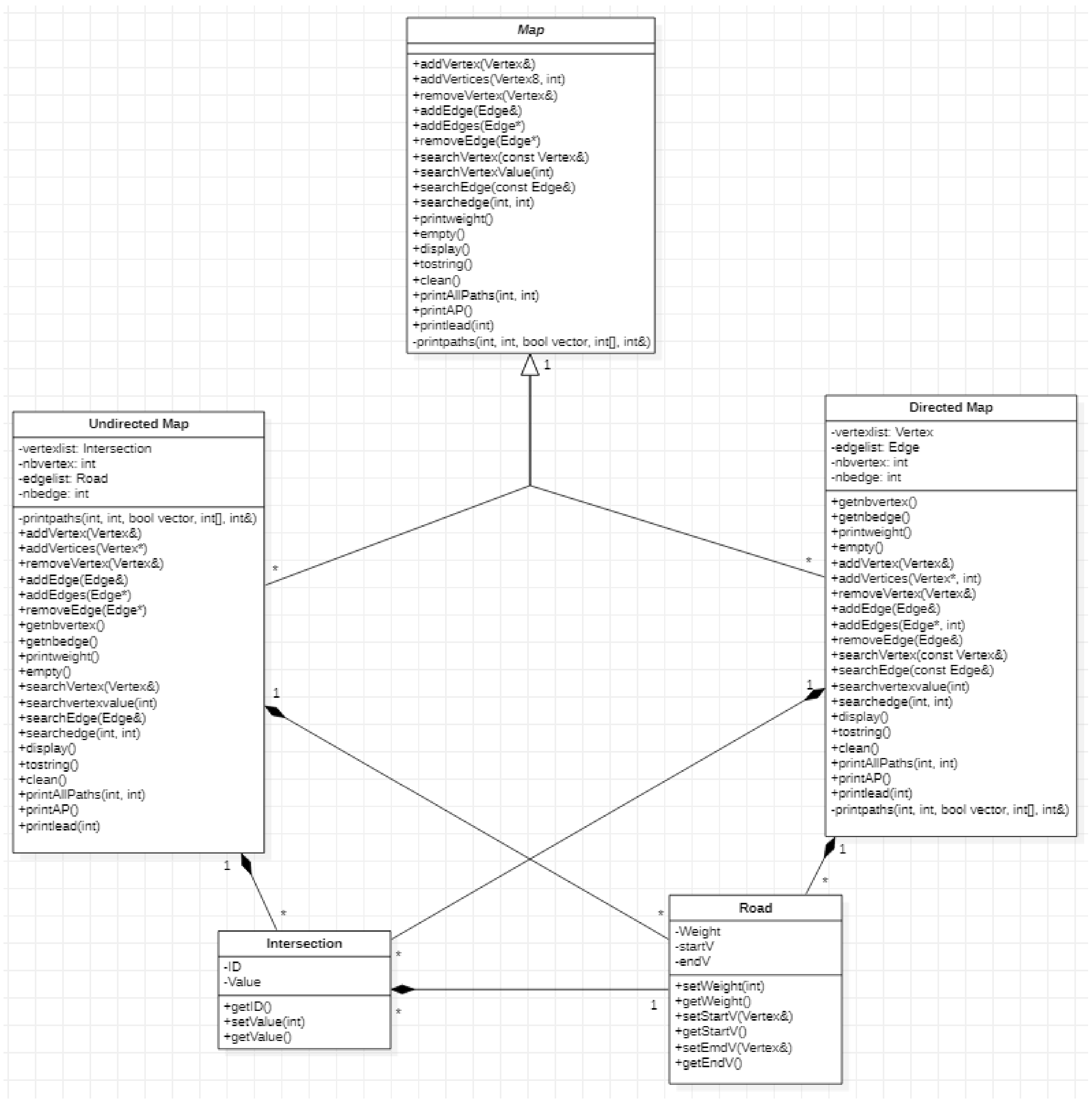
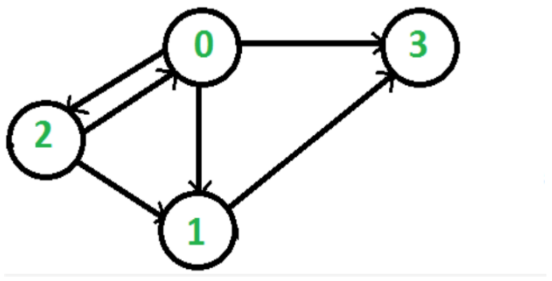


Figure 1: UML Diagram of the Road Network

from an intersection to another and a reference to path_index which is the index of the path array. The implementation of this function for an undirected graph is different to that of a directed graph.

In the case of the undirected graph, the implementation starts by assigning the starting vertex ID to the index of the path array named path_index which was initialized to zero. Consequently, that starting vertex is marked as visited. The path_index is then incremented to allow space for the next path to be stored. From here, the implementation breaks into two sections. The first section considers the case where the path from the starting vertex arrives at the ending vertex. When that occurs, the function will print the path from the starting vertex to the end vertex as well as all the visited vertices along the way. The second case is when the path has not reached the end. When this case occurs, it is divided into two sub-cases. The first one compares the starting vertex ID of all edges contained in the vector edgelist with the starting vertex ID parameter. If they are equal and the vector has not been visited, the function will call itself back but the first parameter will be changed to the ID of the ending vertex of that particular edge. On the other hand, if the ID of the starting vertex matches with the ID of the ending vertex of a particular edge and it has not been visited, the function will call itself but the first parameter becomes the ID of the end vertex of that edge. Whenever a vertex is visited, that vertex is registered as visited in the visited vector.

For a directed graph, the first few steps are the same as in the undirected graph. The first case is identical but the second case is not. The second case only has one sub-case since the edge has a direction. The ID of the starting vertex of all edges contained in edgelist is compared with the ID of the intersection entered as a parameter. If they are equal and the vertex has not been visited, the function calls itself back but the first parameter is changed to the ending vertex of that particular edge.



Source: Available at [1]

Figure 2: Directed Graph

2.2- Example of the Process

Given the directed graph of Figure 3, the sequence of steps to finding a path from vertex 2 to vertex 3 is as follows: the function is called with 2 and 3 as the first and second parameters respectively. The rest of the parameters are as indicated above. Firstly, vertex 2 is compared with vertex 3. Since they are not equal, the function breaks into its second case. Suppose the first edge in the edgelist is the one from vertex 2 to vertex 3, the program will compare the starting vertex ID and the end vertex ID of that edge with vertex 2. Since vertex 2 is the starting vertex, the function will call itself back but the first parameter becomes the ending vertex, in this case, vertex 0. The function is then called recursively until the destination is met. As the path is formed, the vertices become visited one by one. With this, the program will avoid passing through the same vertex more than once. After, it will output all the vertices visited along its path. When the function ends, the array of paths will contain all the possible paths from vertex 2 to vertex 3.

2.3- printAllPaths Function

Secondly, the implementation of the printAllPaths(int s, int d) function uses the previous private function to print all the paths from vertex s to vertex d. It starts by dynamically creating an array called path. The size of the array is one unit larger than the list of vertices. After, it creates and initializes a vector of type boolean to false. This vector will record the state of a vertex along a path. It can either be visited (1) or not (0). This function calls the function findpath mentioned above to fulfill its purpose.

2.4- PrintAP Function

Moving on, the printAP function is similar to the printAllPaths function except it does not take any parameters. Indeed, its purpose is to output all the paths for every permutation of starting and ending intersections. To do so, it uses a nested for loop. The outer loop specifies the ID of the starting intersection while the inner loop

describes the ID of the destination. The conditions for the two loops are from i = 1 and n = 1 to the size of the list of vertices. Inside of the nested loop is the printAllPaths function whose parameters are i and n.

Finally, the printlead function allows the user to find all the paths from his current location to all of the intersections present in the network. The function takes one parameter which is the current intersection of the user. The implementation of this function is similar to the previous one: it will call the printAllPath function in a for loop whose condition is n equal to 1 to n equal to the size of the list of vertices. The parameters of the printAllPath function are the current intersection of the user and n.

3- Techniques Used

The three techniques used in this project were inheritance, polymorphism, operator overloading and exception handling. The use of inheritance and polymorphism is apparent in the relationship of Graph() with Undirected() and Directed(). Indeed, these two classes were derived from Graph(). Furthermore, the Graph class is an abstract class since it does not need to be instantiated. As such, all of the member functions of Graph() are pure virtual functions as indicated by the keyword virtual and =0. Therefore, the implementation of the functions will differ depending on whether it is a directed or undirected graph. For example, as mentioned above, the implementation of the findpath function will be different for an undirected graph since the edges are bidirectional. Moving on, the use of operator overloading is reserved exclusively for graph manipulations and outputting important information related to the road network. The = operator is used to assign a specific graph to another, in other words, copying the vertex and edge lists of a graph to another. The == operator is used to compare two graphs. This comparison is done through the vertex and edge lists. To make this possible, the vertex class and edge class both have their own implementation of the == operator. For the vertex class, the == operator compares the ID and the value of the vertices while the one for the edge class compares the weight, starting vertex, and ending vertex of two edges. The + operator combines two graphs together to create a larger one. The resulting graph is composed of all the vertices and edges of the two graphs. Moving on, the increment operator increments the weight of each edge in a graph. The > operator is used to determine if the sum of the weights of the graph on the left of the operator is bigger than the other. Finally, the stream insertion operator (<<) is used to output the content of the graphs. The graph is displayed by its vertex set and its edge set. Furthermore, the use of exception handling is exclusively reserved for verifying the validity of the user inputs. For example, the program prompts the user to enter the starting and ending intersections of a road. Exception handling was used to ensure that the inputs of these two attributes are correct and if not, the program would display an error message as well as prompt the user to enter the correct values. An example of an incorrect input would be entering an intersection that does not exist.

4- Conclusion

To conclude, this program can help transport planners understand the layout of a city as well as plan their itinerary depending on the best path computed by this program. Nonetheless, the code for adding roads to the network could be more user-friendly since entering a nonexistent road loops the programs back to the start and not back to where an error has been made.

Annex 1

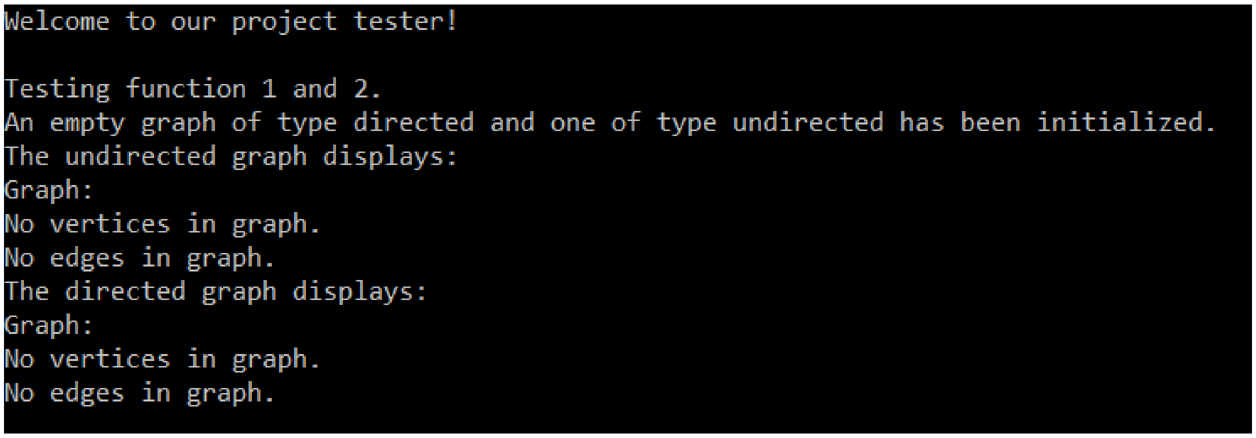

```
Welcome to our project tester!

Testing function 1 and 2.
An empty graph of type directed and one of type undirected has been initialized.
The undirected graph displays:
Graph:
No vertices in graph.
No edges in graph.
The directed graph displays:
Graph:
No vertices in graph.
No edges in graph.
```

Figure 3: Graphs can be empty.

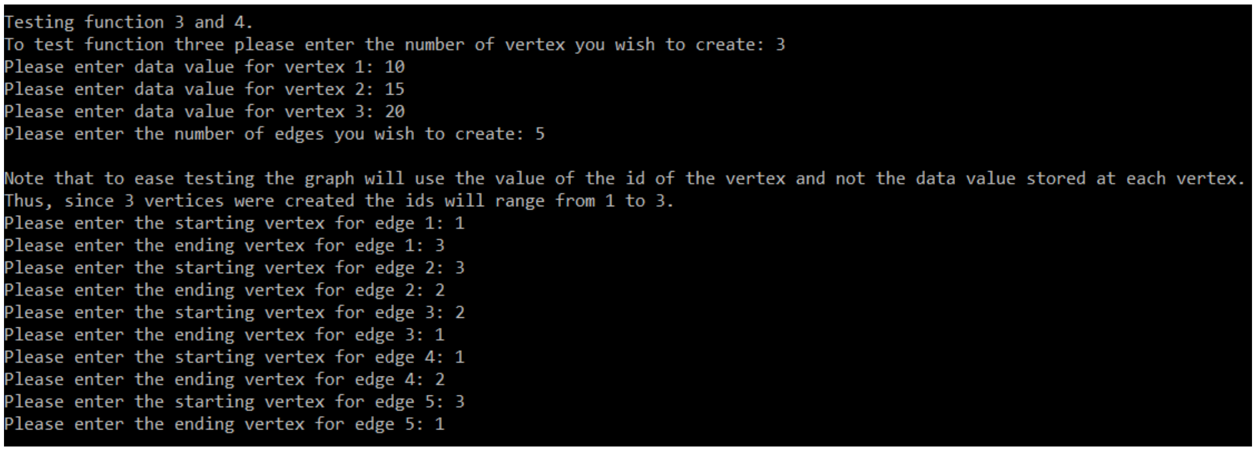
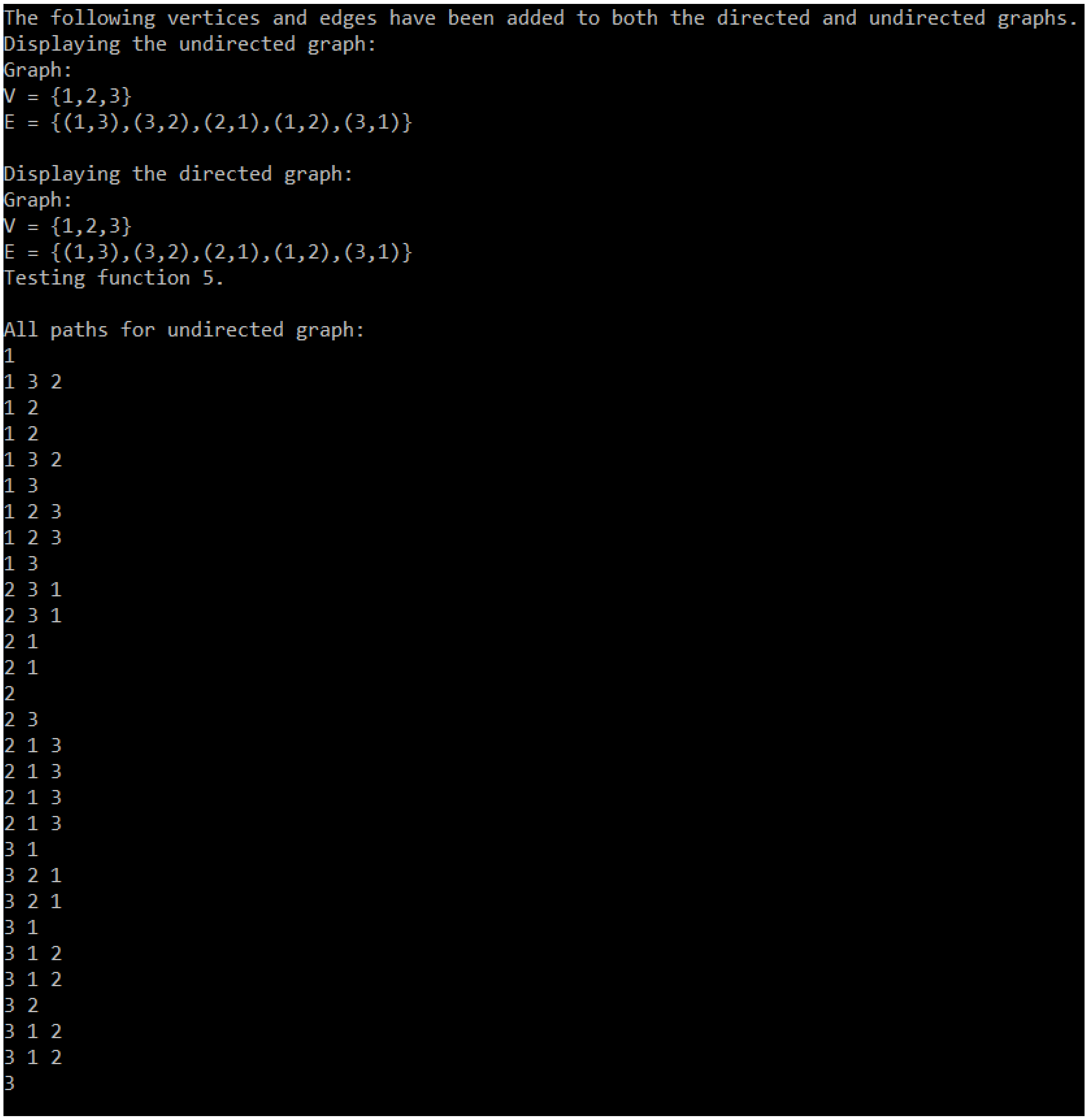
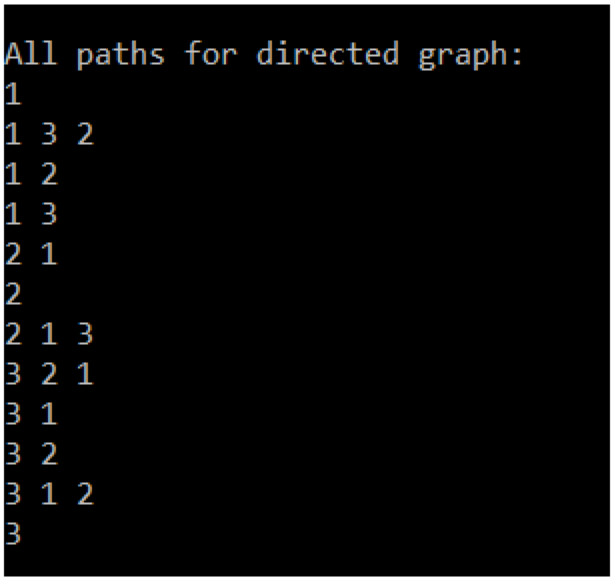

```
Testing function 3 and 4.
To test function three please enter the number of vertex you wish to create: 3
Please enter data value for vertex 1: 10
Please enter data value for vertex 2: 15
Please enter data value for vertex 3: 20
Please enter the number of edges you wish to create: 5

Note that to ease testing the graph will use the value of the id of the vertex and not the data value stored at each vertex.
Thus, since 3 vertices were created the ids will range from 1 to 3.
Please enter the starting vertex for edge 1: 1
Please enter the ending vertex for edge 1: 3
Please enter the starting vertex for edge 2: 3
Please enter the ending vertex for edge 2: 2
Please enter the starting vertex for edge 3: 2
Please enter the ending vertex for edge 3: 1
Please enter the starting vertex for edge 4: 1
Please enter the ending vertex for edge 4: 2
Please enter the starting vertex for edge 5: 3
Please enter the ending vertex for edge 5: 1
```

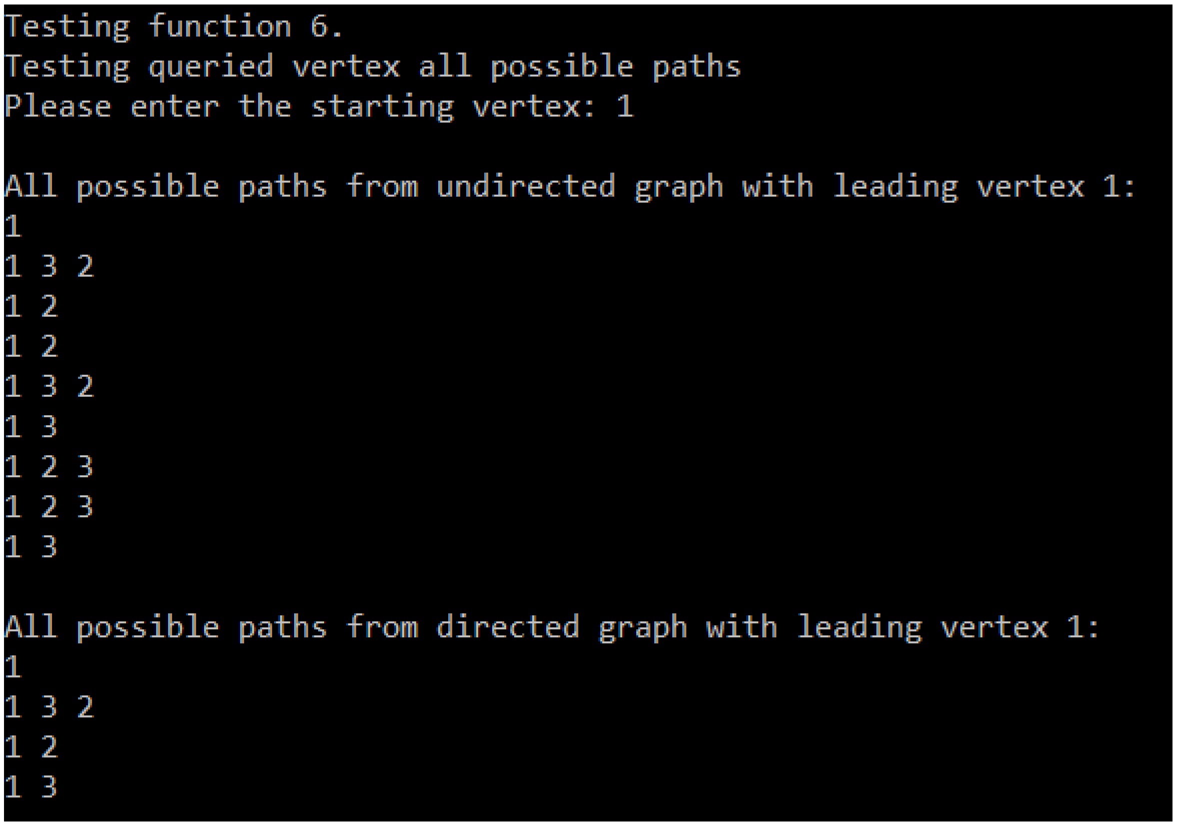
Figure 4: The program prompts the user for the number of vertices and edges. The attributes of the two classes are set by the user.


```
The following vertices and edges have been added to both the directed and undirected graphs.
Displaying the undirected graph:
Graph:
V = {1,2,3}
E = {(1,3),(3,2),(2,1),(1,2),(3,1)}

Displaying the directed graph:
Graph:
V = {1,2,3}
E = {(1,3),(3,2),(2,1),(1,2),(3,1)}
Testing function 5.

All paths for undirected graph:
1
1 3 2
1 2
1 2
1 3 2
1 3
1 2 3
1 2 3
1 3
2 3 1
2 3 1
2 1
2 1
2
2 3
2 1 3
2 1 3
2 1 3
2 1 3
3 1
3 2 1
3 2 1
3 1
3 1 2
3 1 2
3 2
3 1 2
3 1 2
3
```



```
All paths for directed graph:
1
1 3 2
1 2
1 3
2 1
2
2 1 3
3 2 1
3 1
3 2
3 1 2
3
```

Figure 5: Program outputs the content of the directed and undirected graphs as well as all the roads from each intersection.



```
Testing function 6.
Testing queried vertex all possible paths
Please enter the starting vertex: 1

All possible paths from undirected graph with leading vertex 1:
1
1 3 2
1 2
1 2
1 3 2
1 3
1 2 3
1 2 3
1 3

All possible paths from directed graph with leading vertex 1:
1
1 3 2
1 2
1 3
```

Figure 6: Programs outputs all the possible paths from a user specified vertex.

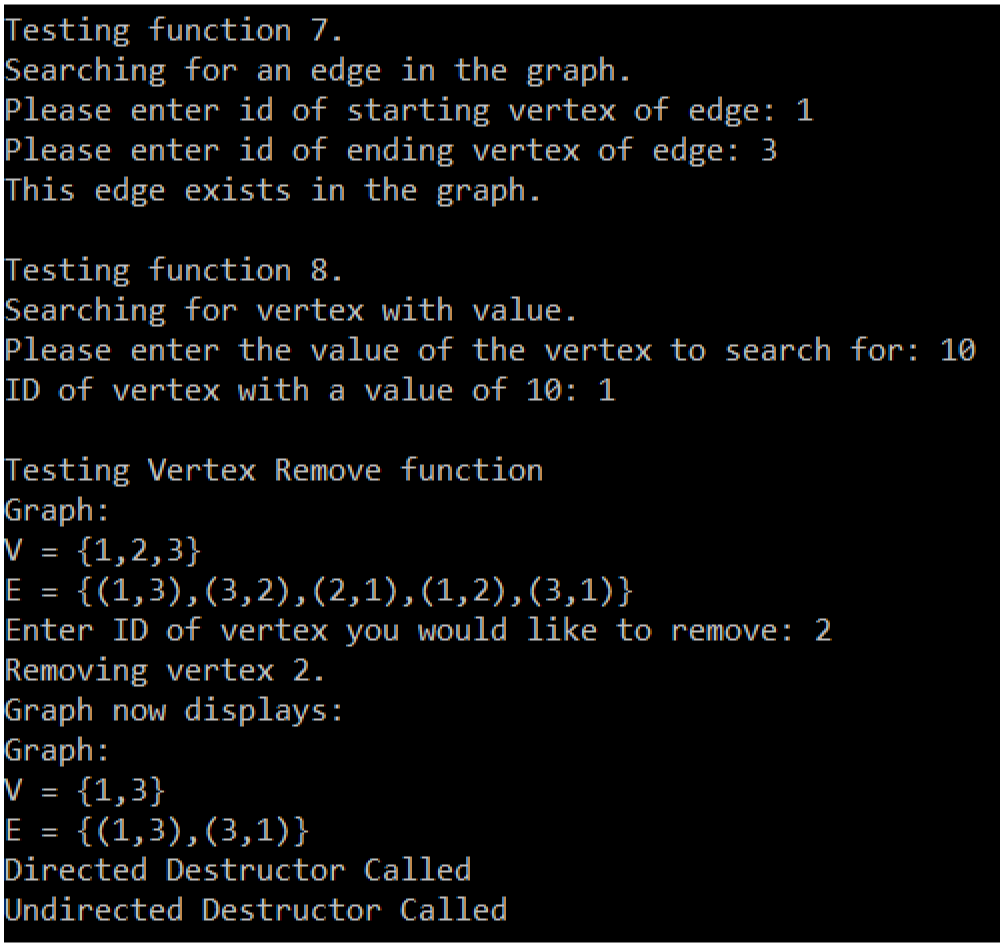
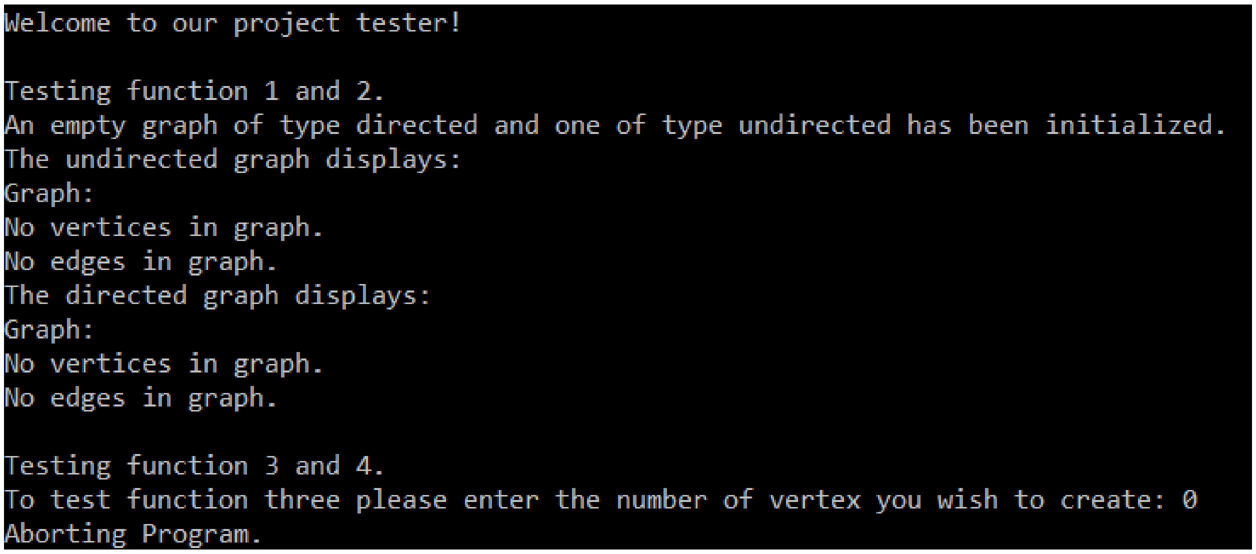
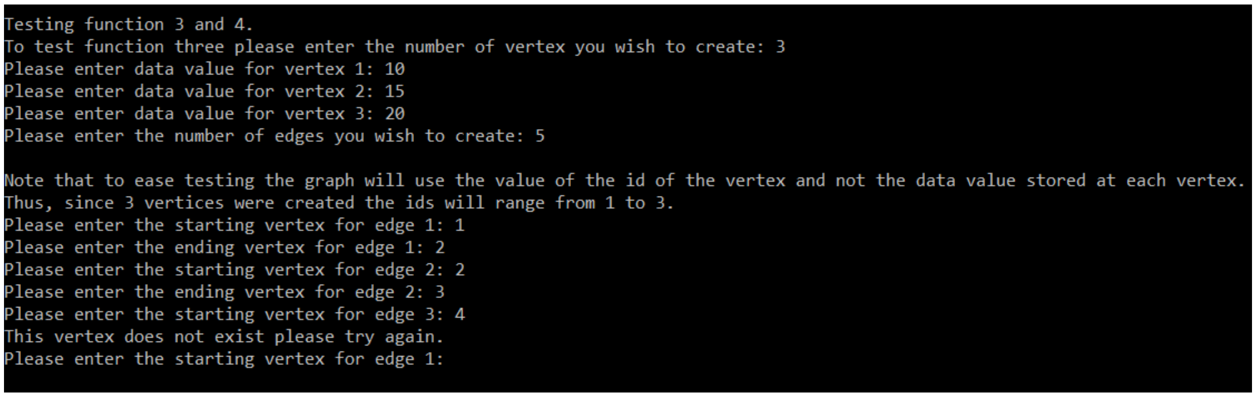

```
Testing function 7.
Searching for an edge in the graph.
Please enter id of starting vertex of edge: 1
Please enter id of ending vertex of edge: 3
This edge exists in the graph.

Testing function 8.
Searching for vertex with value.
Please enter the value of the vertex to search for: 10
ID of vertex with a value of 10: 1

Testing Vertex Remove function
Graph:
V = {1,2,3}
E = {(1,3),(3,2),(2,1),(1,2),(3,1)}
Enter ID of vertex you would like to remove: 2
Removing vertex 2.
Graph now displays:
Graph:
V = {1,3}
E = {(1,3),(3,1)}
Directed Destructor Called
Undirected Destructor Called
```

Figure 7: Testing the search vertex and remove vertex functions. As shown, the output values are correct with the inputs shown previously.



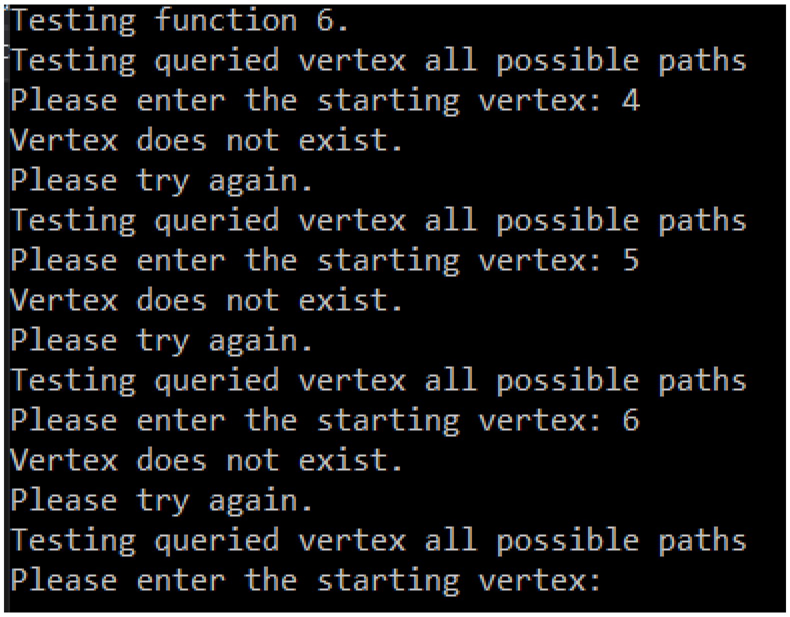
```
Welcome to our project tester!

Testing function 1 and 2.
An empty graph of type directed and one of type undirected has been initialized.
The undirected graph displays:
Graph:
No vertices in graph.
No edges in graph.
The directed graph displays:
Graph:
No vertices in graph.
No edges in graph.

Testing function 3 and 4.
To test function three please enter the number of vertex you wish to create: 0
Aborting Program.
```

Figure 8: Program ends if the number of vertices is 0. This is done using a try/catch block.



```
Testing function 3 and 4.
To test function three please enter the number of vertex you wish to create: 3
Please enter data value for vertex 1: 10
Please enter data value for vertex 2: 15
Please enter data value for vertex 3: 20
Please enter the number of edges you wish to create: 5

Note that to ease testing the graph will use the value of the id of the vertex and not the data value stored at each vertex.
Thus, since 3 vertices were created the ids will range from 1 to 3.
Please enter the starting vertex for edge 1: 1
Please enter the ending vertex for edge 1: 2
Please enter the starting vertex for edge 2: 2
Please enter the ending vertex for edge 2: 3
Please enter the starting vertex for edge 3: 4
This vertex does not exist please try again.
Please enter the starting vertex for edge 1:
```

Figure 9: The function which adds edges to the graph will loop back to the start if the user enters a vertex that does not exist.



```
Testing function 6.
Testing queried vertex all possible paths
Please enter the starting vertex: 4
Vertex does not exist.
Please try again.
Testing queried vertex all possible paths
Please enter the starting vertex: 5
Vertex does not exist.
Please try again.
Testing queried vertex all possible paths
Please enter the starting vertex: 6
Vertex does not exist.
Please try again.
Testing queried vertex all possible paths
Please enter the starting vertex:
```

Figure 10: The function that prints all the possible paths from a user specified vertex will loop indefinitely if the input does not exist.

Reference

GeeksforGeeks, *Print all paths from a given source to a destination - GeeksforGeeks,* Accessed on: Apr 15, 2021. [online]. Available at: https://www.geeksforgeeks.org/find-paths-given-source-destination/