Computer Programming 3 – Assignment 2 Development, Analysis and Testing

J. Francis

# INTRODUCTION

T

HIS report will detail the development, testing and functionality of the “Graph” program noting any errors, complications and design decisions made throughout this process.

# DEVELOPMENT

The basic structure of this program was taken from Graph LAB. The first step in developing the Graph program was to construct the “**Graph Interface**” and the “**Adjacency List Directed Graph**” classes and implement the “**getVerticies**”, “**adjacentTo**”, “**addEdge**”, “**addVertex**”, “**hasEdge**”, “**getAdjacentEdges**”, “**hasVertex**”, “**readInFromFile**”, “**printGraph**”, “**dijkstra**”, “**dijkstraPriorityQueue**”, “**customDijkstraPriorityQueue**”, “**printDistances**”, “**praintOutgoingEdges**” and “**pathTo**” methods. This was mostly done by copying over the code from the **adjacencyListDirectedGraph** class from LAB3 and modifying it as necessary.

### Graph Interface

The “**graph**” interface was mostly copied over from the Graph LAB. Various methods were added such as the “**addVertex**”, “**getAdjacentEdges**”, “**readInFromFile**”, “**printGraph**”, “**Dijkstra**”, “**dijkstra Priority Queue**”, “**custom Dijkstra Priority Queue**”, “**printDistances**”, “**printOutgoingEdges**” and “**pathTo**”. The “**readInFromFile**”, Dijkstra’s algorithms and print methods were included in the interface because this assignment only required one class to implement the interface so it was just cleaner to incorporate all methods needed in the interface. Additionally, methods like “**readInFromFile**” will vary depending on the type of graph implemented so it makes more sense to have to override these methods within the implementation to suit specific graph implementations.

### AdjacencyListDirectedGraph Class

Next to be constructed was the “**adjacency List Directed Graph**” class. Again, the foundation for this was copied over from the Graph LAB. Various changes were made in the effort of making the way the graph reads files etc more efficient. The first of these changes was to how the adjacent vertices were stored. Previously, the “**Vertex**” class would store an ArrayList of neighbor vertices. The “**adjacency List Directed Graph**” was modified such that it would map a source vertex to a HashSet of edges that contain the weight and target vertex. The edge class will be discussed later. This list of changes to this class is as follows.

#### **adjacentTo** Method

The “**adjacentTo**” method was changed such that it would iterate through the set of edges mapped to the provided source vertex and add the target vertex of each edge to an ArrayList to be returned.

#### **getVerticies** Method

The “**getVerticies**” method remained unchanged. This method just sorts then returns the ArrayList of vertices within the graph.

#### **addVertex** Method

The “**addVertex**” method was added after realizing that the “**readInFromFile**” method only added the vertices connected by an edge. The result of this was that unconnected vertices were not added to the graph. As such, the “**addVertex**” method was created. This method checks the list of vertices within the graph and if it the vertex to be inserted not already inserted, it will be added to the list.

#### **addEdge** Method

The “**addEdge**” method needed to be completely changed in order to incorporate the idea of storing weights within an edge object. The “**addEdge**” method takes 3 parameters, a source vertex, a target vertex and a weight. If the list of vertices within the graph does not contain the source or target vertices, they are added to the list and the source vertex is then mapped to the corresponding edge object. The edge object, containing the source and target vertices and the weight are then added to a TreeMap within the **adjacencyListDirectedGraph**.

#### **hasEdge** Method

The “**hasEdge**” method was also modified such that it iterates through the set of edges associated with a given source vertex and returns a value of true if the directed edge exists. The “**directedEdgeExists**” method is used to achieve this and will be discussed later.

#### **getAdjacentEdges** Method

The “**getAdjacentEdges**” method was added such that all the edges mapped to a certain source vertex could be obtained for use within Dijkstra’s algorithm and the GUI. This method just returns the set of edges associated with a giver source vertex.

#### **hasVertex** Method

The “**hasVertex**” method remained unchanged from the code used within LAB3. The method simply returns true if the vertex in question is contained within the vertices list.

#### **readInFromFile** Method

The **readInFromFile**” method utilizes the JDOM library in order to parse .xml and .graphml files. This method extracts the vertices and edges listed within the .xml or .graphml file and inserts them into the graph. Various issues were encountered while trying to implement this method. Firstly, the program was unable to access the children of the root element. It was later discovered that the namespace needed to be specified. Upon correcting that problem, everything appeared to be working fine until I reached the more elaborate testing stage utilizing the try program. I discovered that I was only inserting the nodes mentioned in the edge elements. The result of this was that disconnected nodes were not added to the graph. This was fixed creating the “**addVertex**” method and by obtaining the list of nodes then adding them individually if they weren’t already added from adding the edges.

#### **Dijkstra** Method

The “**Dijkstra**” method was implemented utilizing some pseudocode from Wikipedia and some code from the breadth first search class from LAB3. The idea of mapping a vertex to a distance and a previously visited vertex came from LAB3. This method uses a standard queue as the data structure to hold the unfinished vertices. Firstly, this method initializes all vertices within the graph by mapping them all to a distance. The source was mapped to a distance of 0 and all others were mapped to a distance of Interger.MAX\_VALUE using a HashMap. Integer.MAX\_VALUE simulates infinite distance. All the vertices were also mapped to a previous vertex, set to null within the initialization. They are then all added to the queue. The algorithm is essentially while the queue is not empty, iterate though all vertices. If the queue does not contain the source vertex, set the source to the head of the queue without removing it. If the vertex’s distance of this iteration is less than the distance of the source vertex and if the vertex of this iteration is still in the queue, set the current vertex to the vertex of this iteration. This essentially finds the vertex with the smallest distance still in the queue. This vertex is then removed from the queue. For each outgoing edge of the current vertex, we see if the target vertex is in the queue. If it is, we calculate the new distance by adding the distance of the source vertex with the weight of that edge. If this calculated distance is less than the distance of the target vertex, the target vertex’s distance is replaced with the new one and the target vertex’s previous pointer is set to the current vertex. This is repeated while the queue is not empty updating the distance to each vertex and the path travelled.

#### **dijkstraPriorityQueue** Method

This method was again implemented using some pseudocode found on Wikipedia and some code from LAB3. This method has a similar initialization stage to the “**Dijkstra**” method where all the vertices are mapped to distances and previous vertices with HashMaps although only the source vertex is added to the queue which is now a priority queue. The algorithm is as follows. While the queue is not empty, get the first vertex from the head of the queue. For each outgoing edge of this vertex, calculate the new distance by adding the distance of the current vertex to the weight of the current edge. If the distance of the target vertex is greater than the calculated distance, remove the target from the queue, replace the target distance with the calculated distance and set the targets previous pointer to the current vertex, then add the target vertex back onto the queue. This method was much simpler than the previous to implement and is also more efficient. The time and complexity analysis will be discussed later.

#### **customDijkstraMinHeap** Method

This method is almost the same as the “**Dijkstra**” method except it now utilizes the custom min heap structure. This method consists of an initialization stage and the actual algorithm. The time complexity of this algorithm will be discussed later.

#### **pathTo** Method

This method was taken from LAB3 and slightly modified in order to retrieve the shortest path to a particular vertex. The only change made was to reverse to order of the list of vertices before returning it.

#### **printDistances** Method

This method was constructed such that the distance from a source node to all other nodes could be displayed within the GUI. This method just iterates through all verticies within the graph and outputs the current vertex, the path the get to that vertex from the source and the cost.

#### **printOutgoingEdges** Method

This method was created so the user can see all the outgoing edges from a chosen vertex. It simply iterates through set of edges associated with a given vertex and outputs them in a asthetically pleasing format.

#### **printGraph** Method

This method was taken from LAB3 and modified to output the list of vertices within the graph and the list of vertices each vertex is connected to within the GUI.

### Vertex Class

This class was taken directly from LAB3 and simplified as needed. The “**Vertex**” class contains a few very simple methods as that needed to be overridden due to implementing comparable such as “**hashCode**”, ”**equals**”, ”**compareTo**” and “**toString**”.

### DirectedEdge Class

This class was constructed in order to contain the weight given within the.xml and .graphml files and the source and target vertices. It contains the following methods.

#### **setWeight** Method

This method takes an integer as a parameter and assigns it to a private variable “**weight**”.

#### **getWeight** Method

This method simply returns the value of the private variable “**weight**”.

#### **getSource** Method

This method returns the source vertex of the directed edge.

#### **getTarget** Method

This method returns the target vertex of the directed edge.

#### **directedEdgeExists** Method

This method takes two vertices as parameters, a source and a target, and compares the labels with the source and target vertices of the directed edge. If they match, the method returns true, else it returns false.

### MinHeap Class

#### **isEmpty** Method

This method simply returns true if the heap is empty.

#### **Insert** Method

This method inserts the vertex objects into an array starting from position 0 and call the “**siftUp**” method. The distance associated with a vertex is stored in an additional array, at the same index.

#### **Remove** Method

This method removes a vertex from the top of the heap and calls the “**siftDown**” method before returning said vertex.

#### **siftUp** Method

This method resorts the heap structure, bottom up.

#### **siftDown** Method

This method resorts the heap structure, top down.

#### **getParent** Method

This method returns the index of the parent “node” within the array.

#### **getLeftChild** Method

This method returns the index of the left child “node” within the array.

#### **getRightChild** Method

This method returns the index of the right child “node” within the array.

#### **checkBoundary** Method

This method returns true is the given index of a “node” is situated on the boundary of the “tree”.

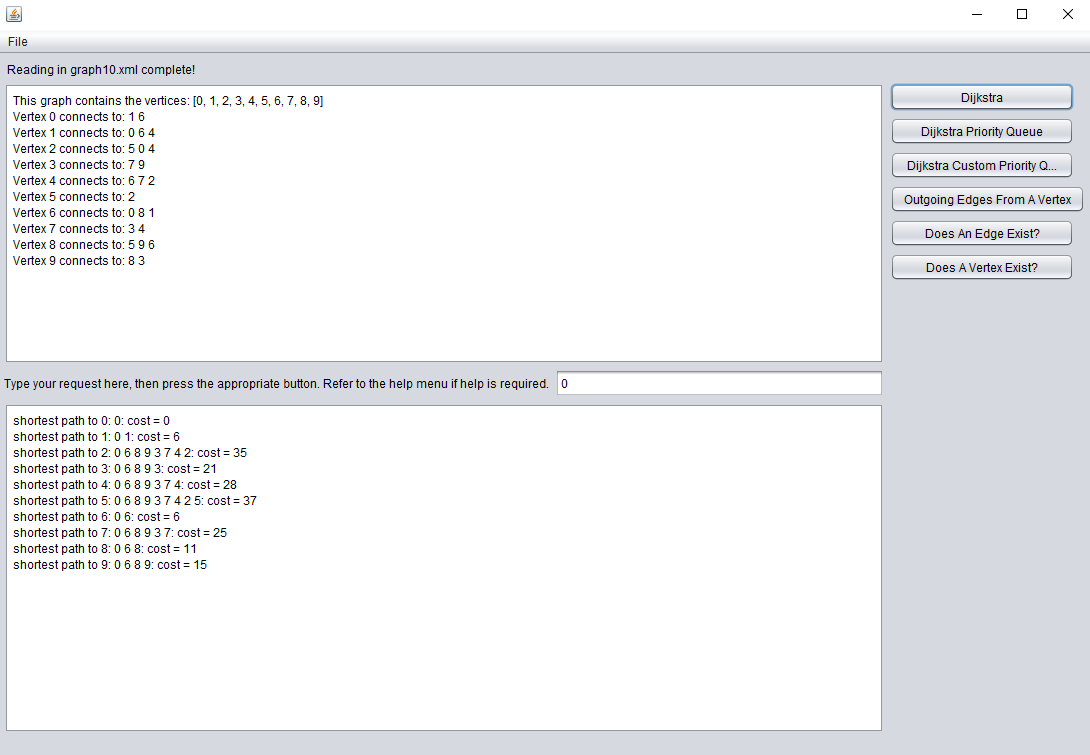
#### **isNull** Method

This method returns true is the given index does not contain a vertex.

#### **Swap** Method

This method simply swaps the position of 2 vertices within the heap. It also swaps the position of the associated distance within its respective array.

### Graphical User Interface



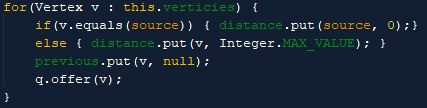
*Fig. 1. GUI.*

The final part was to construct the GUI. To start, the GUI was designed in NetBeans GUI design environment. The GUI consists of 2 large text areas. The upper text area displays the graph once it has been loaded via the “**printGraph**” function. The lower text area is where the results of various queries are output. There is a 3rd text area in the center where the user can input some information then press one of the buttons on the right in order to do various things like execute Dijkstra’s algorithm or see if an edge exists. Files can be loaded via the file menu and help can be accessed via the help menu. All edge cases have been handled like trying to load multiple files or incorrect input.

# ANALYSIS OF ALGORITHMS

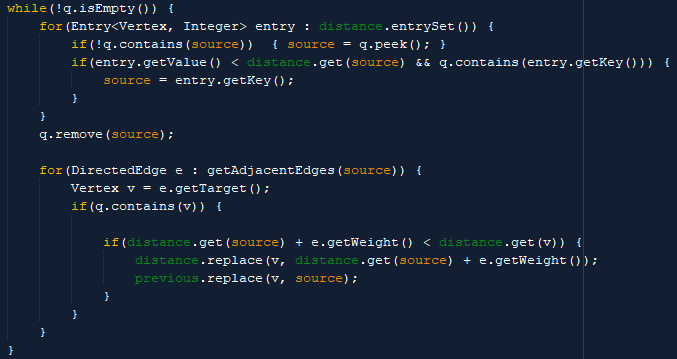
### Dijkstra’s Algorithm

While examining the code used in the “**dijkstra**” method, the time and space complexity can be determined as follows.



*Fig. 2. Dijkstra Method Initialization.*

It can be seen that the initialization code has V iterations, where V is the number of vertices. Within each iteration, there is a single comparison, an O(1) operation, 2 put method calls for a HashMap, both O(1) operations, and an offer method call, also a O(1) operation. Thus, the initialization segment is a constant time operation.



*Fig. 3. Dijkstra Method.*

Next is the actual algorithm. Since the queue is initially full, the while loop will undergo at least V iterations. Every iteration a for loop undergoes V iterations. Within the first for loop, q.contains() is called, an O(V) operation, a getValue() method is called, an O(1) operation, a get() method is called, an O(1) operation, and another q.contains() method, O(V) operation. Therefore, so far the worse case time complexity is as follows.

This looks bad but the queue will not always contain V vertices thus the time complexity can be roughly simplified as follows.

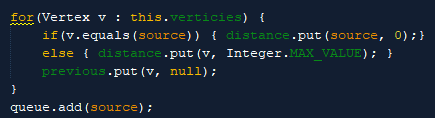
Next a for loop will undergo iterations. Since most of the example graphs are sparse, we can assume E V. Within the for loop a q.contains() method is called, O(V) operation, and all other operations are O(1). Therefore the time complexity of this section is as follows.

But since the queue will only have V vertices on the first iteration and a reduced number of vertices on subsequent iterations, the complexity can be roughly reduced to the following.

Therefore the total time complexity for this algorithm is:

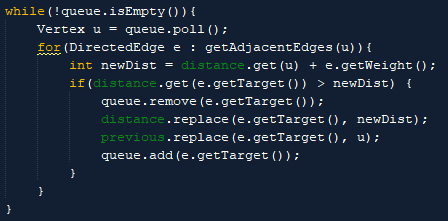
### Dijkstra’s Algorithm (Priority Queue)

While examining the code used in the “**Dijkstra Priority Queue**” method, the time and space complexity can be determined as follows.



*Fig. 4. DijkstraPirorityQueue Method initialization.*

As previously discussed, it can be seen that the initialization part of this algorithm is a constant time operation.

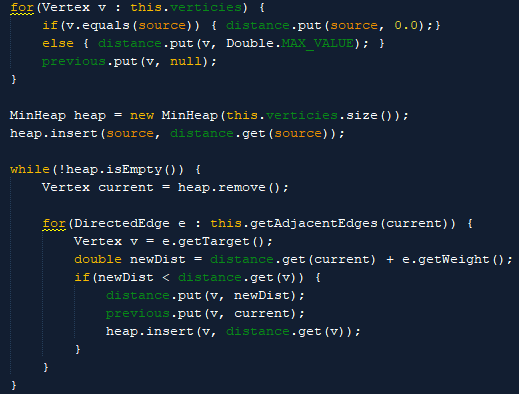


*Fig. 5. DijkstraPirorityQueue Method algorithm.*

The main part of the algorithm can be analyzed as follows. The while loop will undergo at least V iterations. Within the while loop, q.poll() is called, an O(log(V)) operation, then a for loop undergoes E iterations. Within the for loop, a q.add() method is called, an O(log(v)) operation and a q.remove() method is called, an O(log(v)) operation. All the other operations are constant time. This can be simplified to the following time complexity.

It can be seen that the time complexity for this algorithm is better that the previous. This can be confirmed by the run time displayed within the GUI.

### Dijkstra’s Algorithm (Custom Min Heap)



*Fig. 6. CustomDijkstraMinHeap Method.*

Again, the initialization stage of this algorithm is a constant time operation. The main part of the algorithm can be analyzed as follows. The while loop will undergo at least V iterations. For each iteration, remove() is called, a O(log V) operation, and an insert() is called, a O(log E) operation. This can be roughly simplified to the following.

O[V log E + E log E]

It can be seen that the time complexity for this algorithm is much better that the previous. This can, again, be confirmed by the run time displayed within the GUI when testing the various algorithms

# TESTING

The code was constantly tested throughout development and several issues were identified/rectified along the way. The testing regime was as follows.

### Testing the **readInFromFile** method.

After constructing the “**readInFromFile**” method, a driver class “**tester**” was constructed in order to create a new adjacencyListDirectedGraph and call g.readInFromFile(). Various System.out.println statements were added within the method to ensure the correct information was being extracted and passed to the “**addEdge**” method correctly. It was determined that the method was operating as expected. It was later discovered that when reading in more elaborate graphs that all the nodes were not being added to the graph. It was discovered that the “**readInFromFile**” method was only adding nodes defined within the edge elements and not from the node list. This meant that disconnected nodes were not being added to the graph. To solve this, a node list was created and after adding all the edges, the node list was checked to ensure that all nodes, including disconnected ones, were being added correctly. It was at this point the “**readInFromFile**” method was deemed to be operating correctly.

### Testing the **addEdge** method.

To test the “**addEdge**” method, the “**printGraph**” method was first taken from LAB3 such that I could test reading in a simple graph and compare the output with the .xml file to ensure edges were being added correctly. This in conjunction with System.out.println statements outputting the edges list every iteration enabled accurate checking and confirmation that the method was operating correctly.

### Testing the **hasEdge** method.

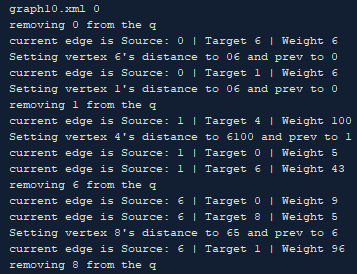
The “**hasEdge**” method was written while implementing the GUI (to provide extra functionality) as the method was not needed previously within any of the algorithms. Upon testing this method, it was discovered that the internal method, “**edgeExists**”, used to compare the source and target vertices would return true if the edge contained both of the given vertices, with no defined order. This meant that there was no concept of edge direction. As a result, a new method was created within the edge class, “**directedEdgeExists**” such that it compares a given source and target vertex with the source and target vertex stored within the edge object. If both these objects matched then the function would return true. After testing and correcting the problem, the function worked as expected.

### Testing the **printGraph** method.

The “**printGraph**” method was tested using try’s expected output as a formatting example. Upon submitting the code to try, it was evident that the output format of “**printGraph**” needed to be changed. Try was also a useful testing tool while testing the various Dijkstra algorithms.

### Testing the **Dijkstra** method.

The “**dijkstra**” method was tested by placing various System.out.println statements within the method and comparing the output each step of the way with a solution worked by hand. Various problems arose while trying to store the previous vertex and distance within the node object. To avoid having to write a custom comparator class, I utilized a HaspMap to map a vertex to a distance and previous vertex. This also meant I could use the code from LAB3 to retrieve the path of vertices travelled, “**pathTo**“. An example of the output during the troubleshooting stage can be seen below.



*Fig. 7. Example output.*

### Testing the **dijkstraPriorityQueue** method.

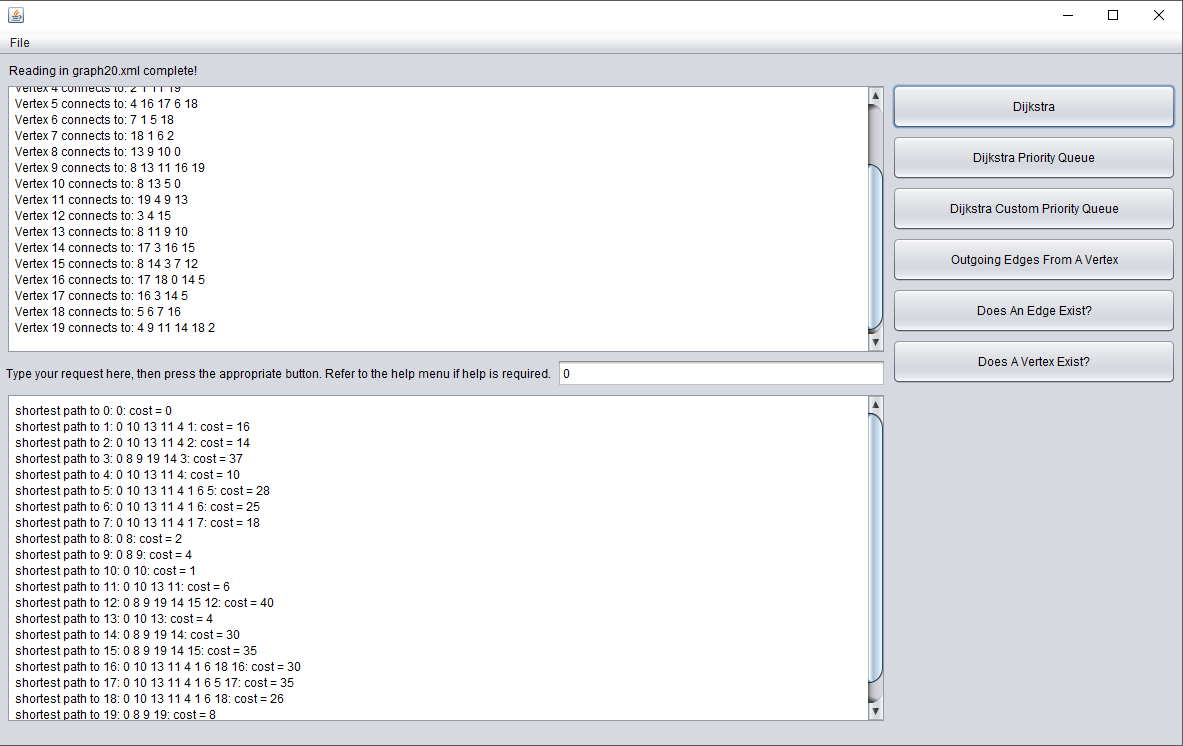
The “**dijkstraPriorityQueue**” method was tested in similar fashion to the previous method using System.out.println statements and try. The pseudocode used to implement the algorithm was taken from Wikipedia. After implementing it in code and examining the output, it appeared to work. Submitting to try served to further confirm this.

### Testing the **customDijkstraMinHeap** method.

Again, testing the “**customDijkstraMinHeap**” was similar to testing the previous methods. Examining the output with the expected output and using try confirmed the method was operating correctly.

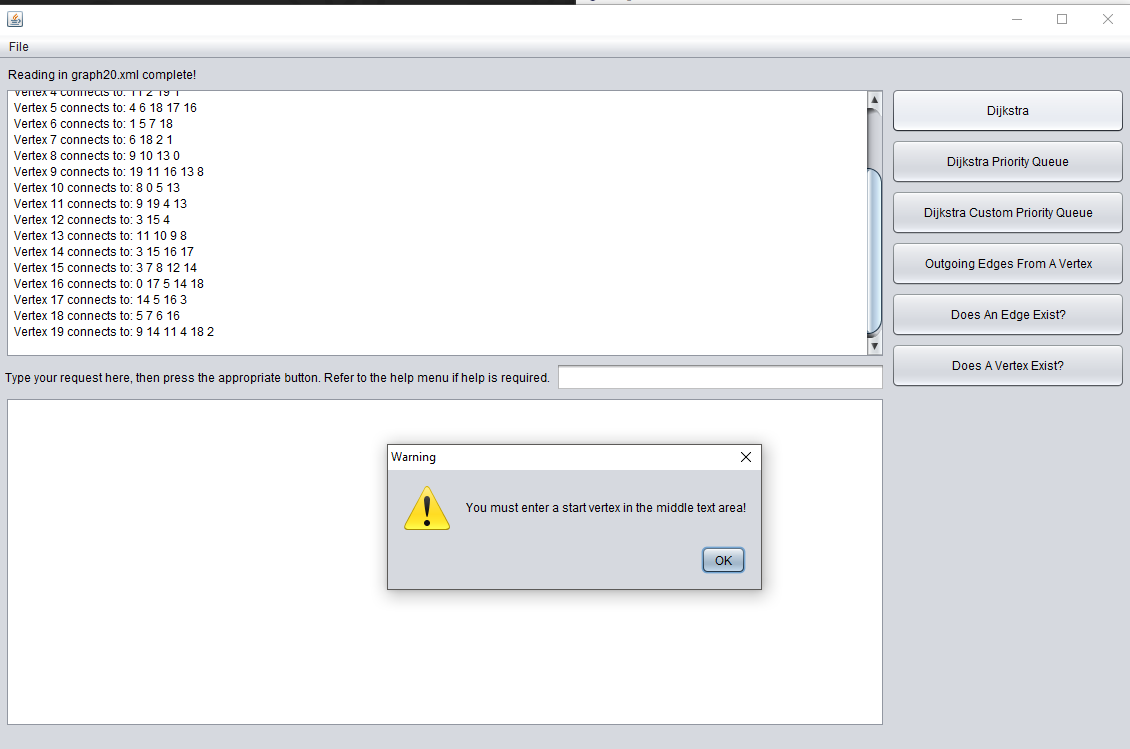
### Testing the **GUI** & final implementation

After implementing the GUI, all aspects from loading files and general operation of the program to the user’s ability to access help, should it be required, were tested thoroughly. To start, the functionality of the buttons and algorithms were tested as follows.

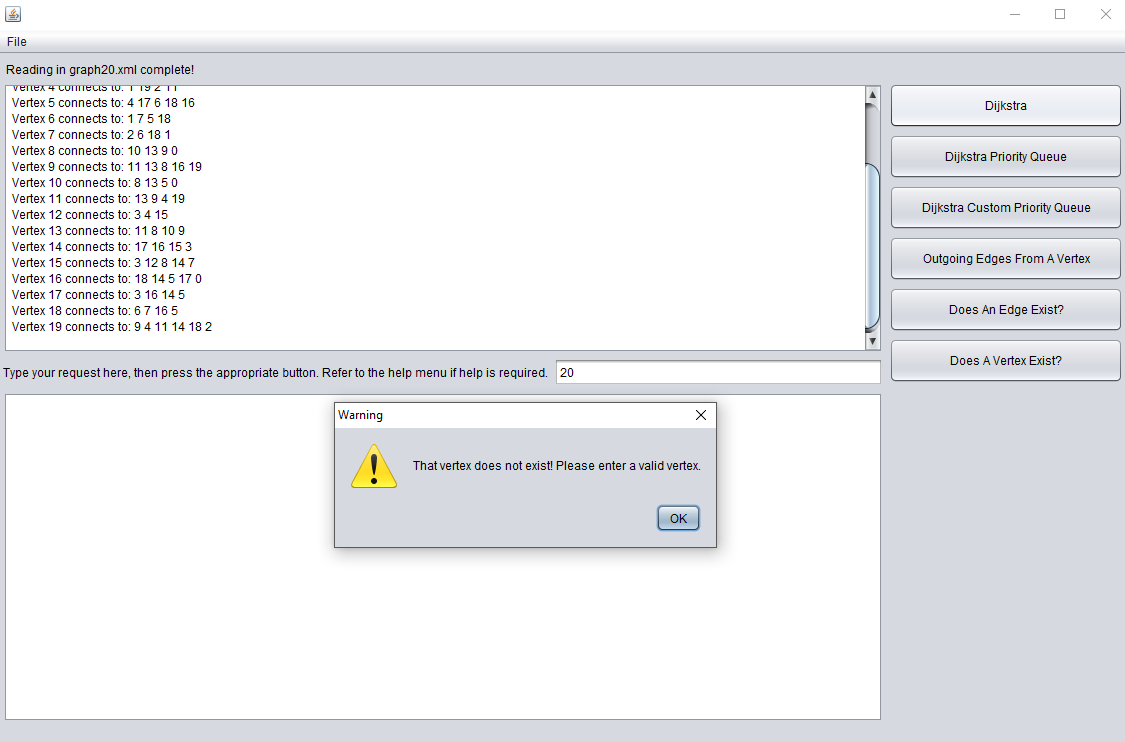


*Fig. 8. Testing Dijkstra functionality.*

Figure 8 shows testing Dijkstra’s algorithm within the GUI to ensure that the output is as expected.

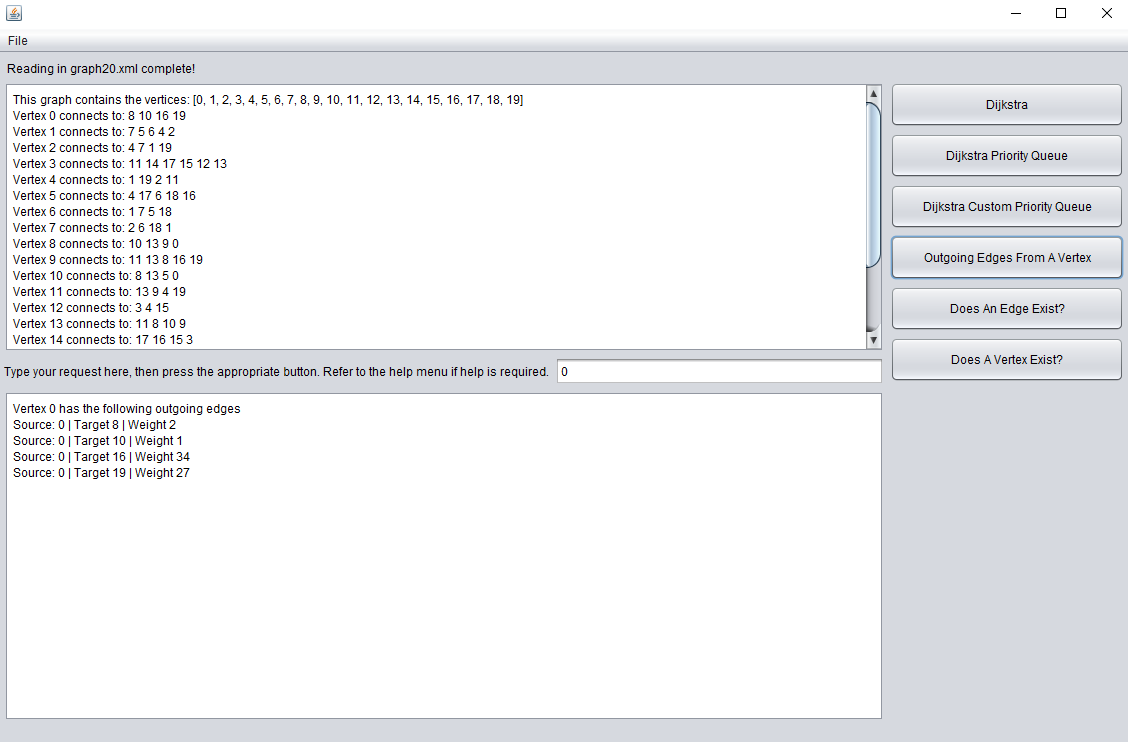


*Fig. 9. Testing Dijkstra functionality.*



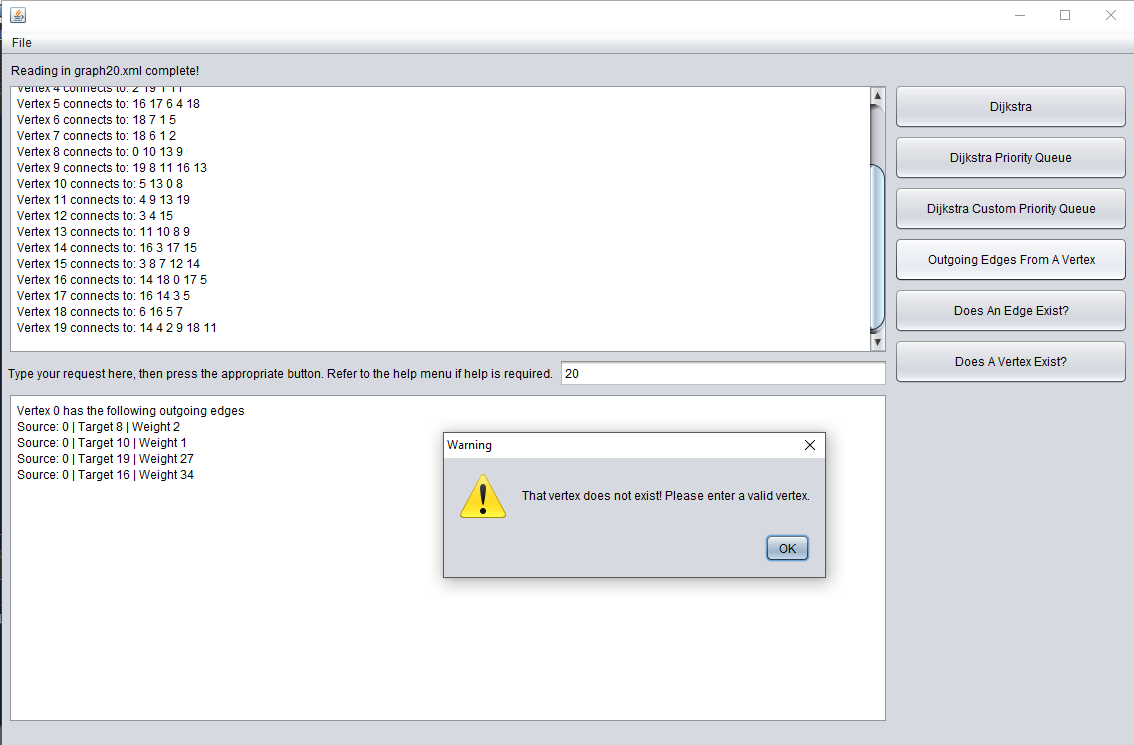
*Fig. 10.* *Testing Dijkstra functionality.*

Figures 9 & 10 shows testing Dijkstra’s algorithms with incorrect input.

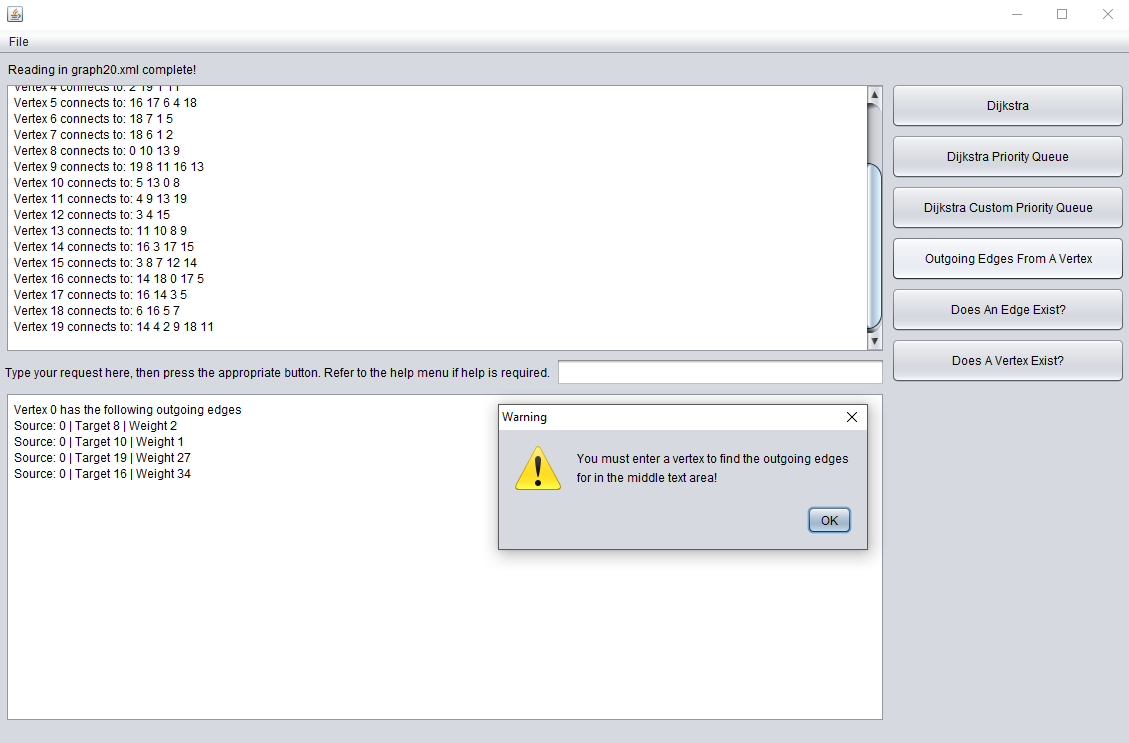


*Fig. 11.* *Testing outgoing edges functionality.*

Figure 11 shows the testing of the outgoing edges button to ensure the correct result is output.

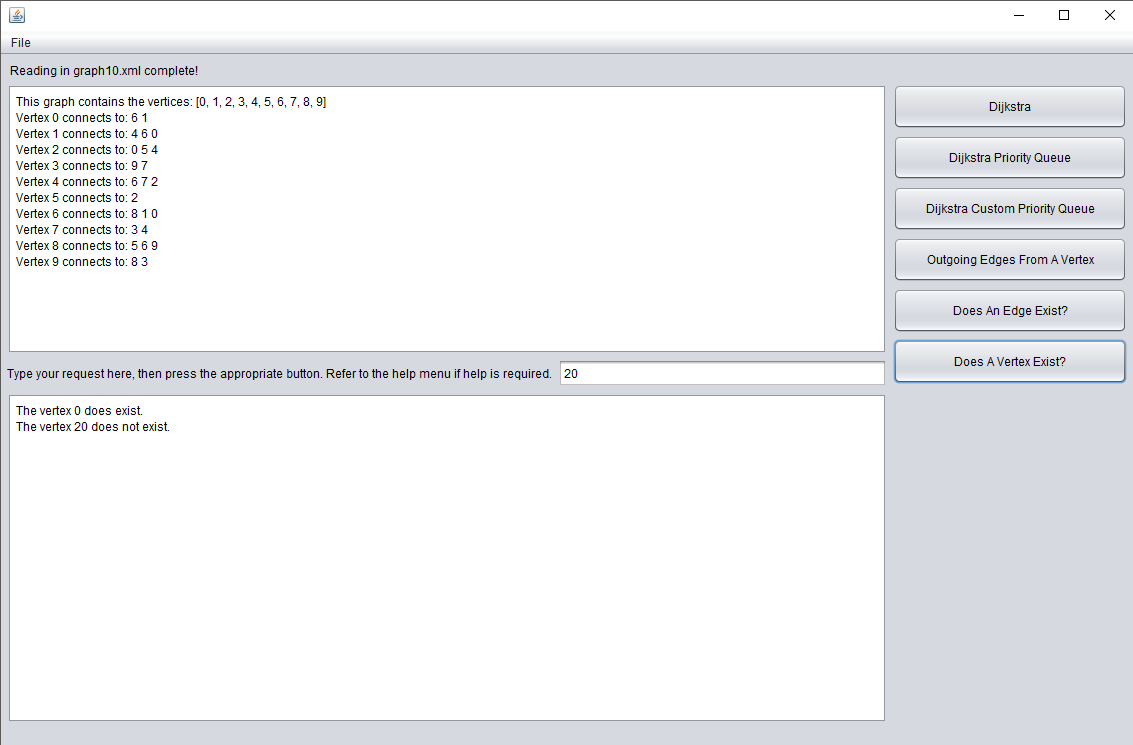


*Fig. 12.* *Testing outgoing edges functionality.*



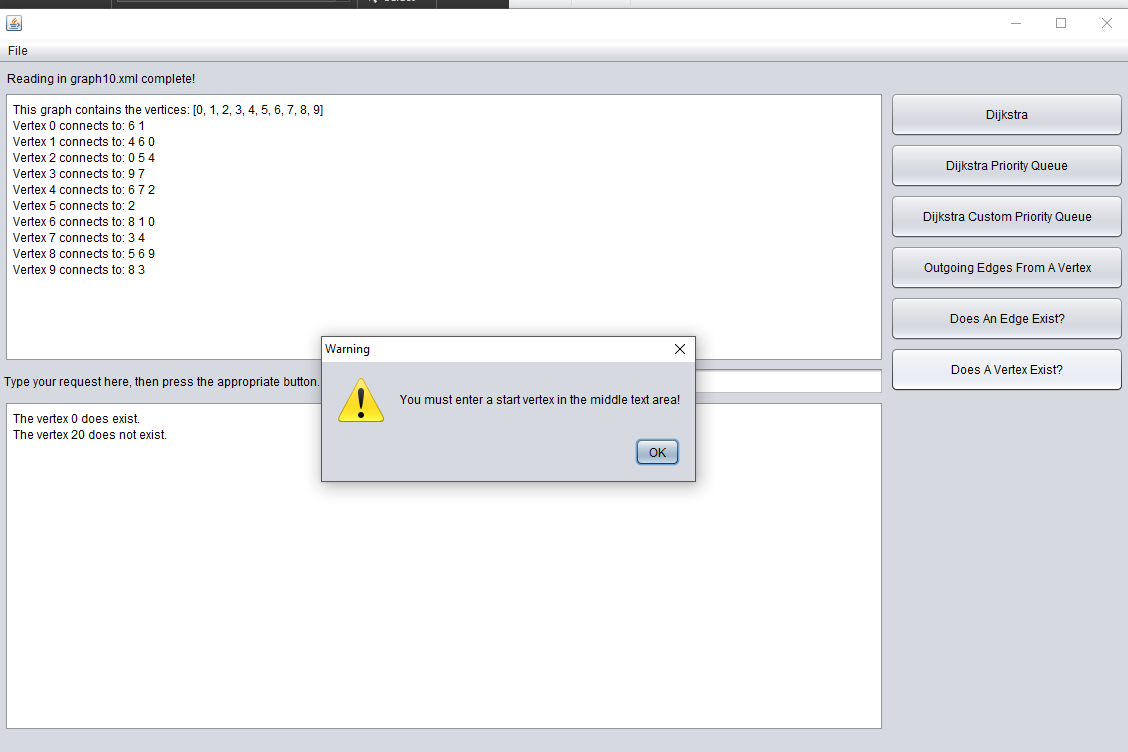
*Fig. 13.* *Testing outgoing edges functionality.*

Figures 12 and 13 show testing the outgoing edges with incorrect input.



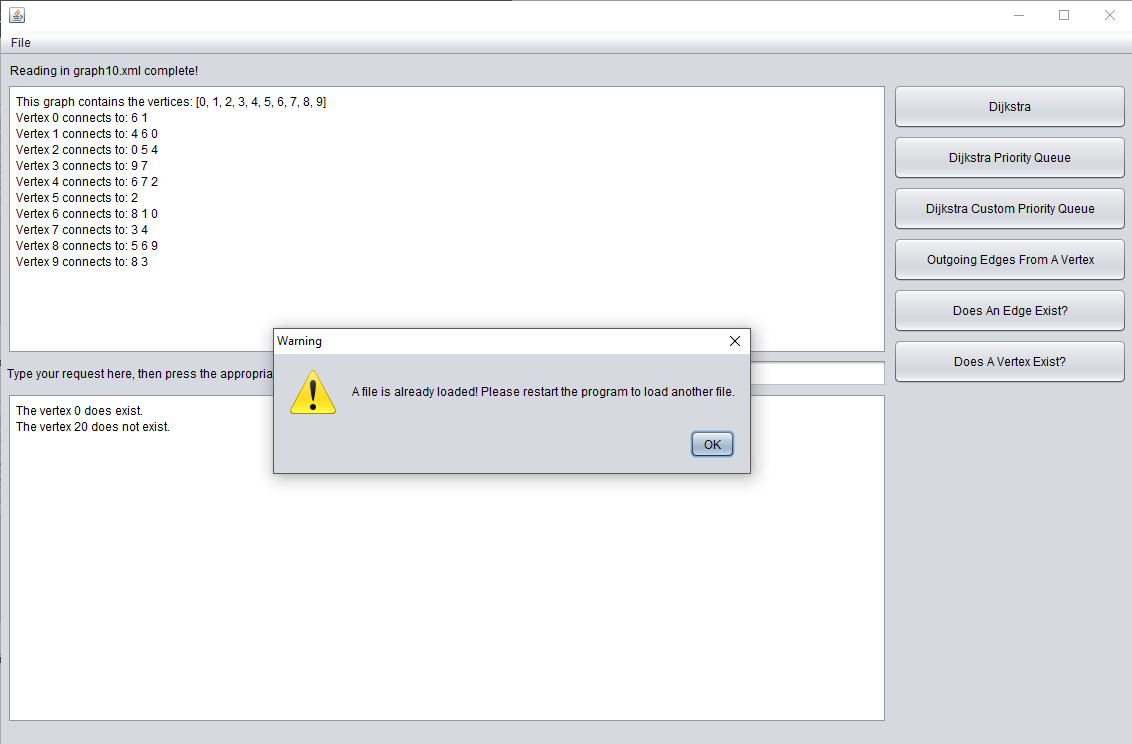
*Fig. 14.* *Testing vertex exists functionality.*

Figure 14 shows the testing of the vertex exists button to ensure correct output.



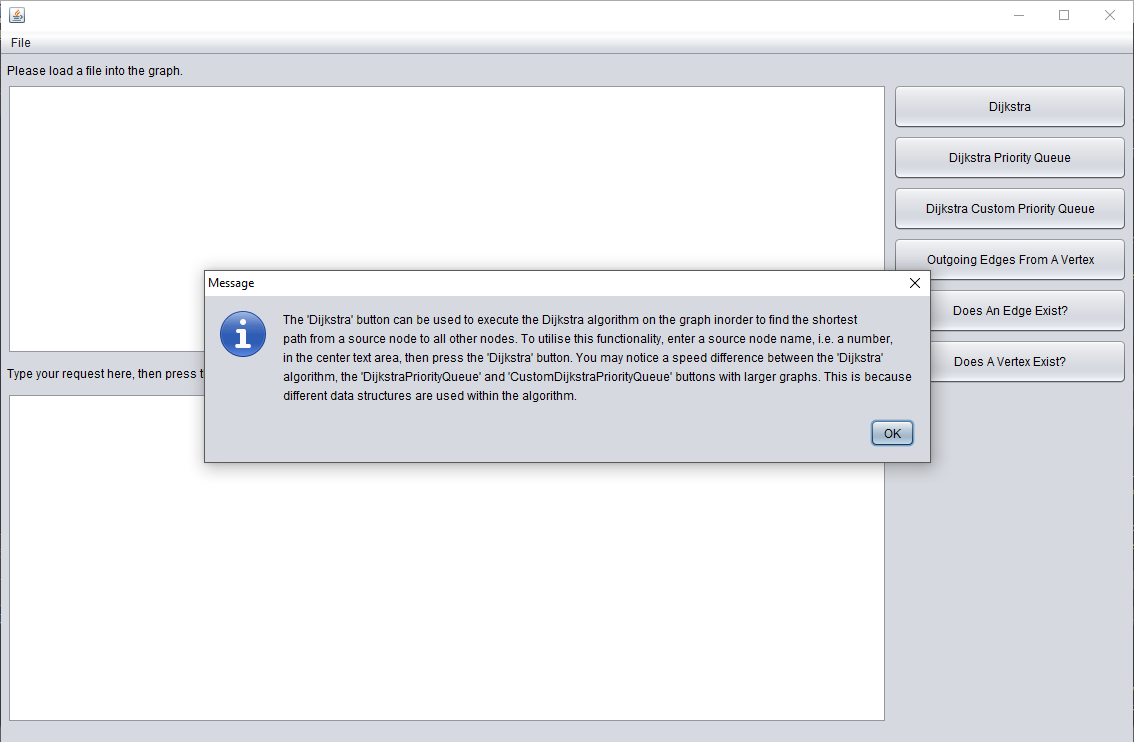
*Fig. 15.* *Testing vertex exists functionality.*

Figure 15 shows testing the vertex exists functionality with incorrect user input.



*Fig. 16.* *Testing multiple file functionality.*

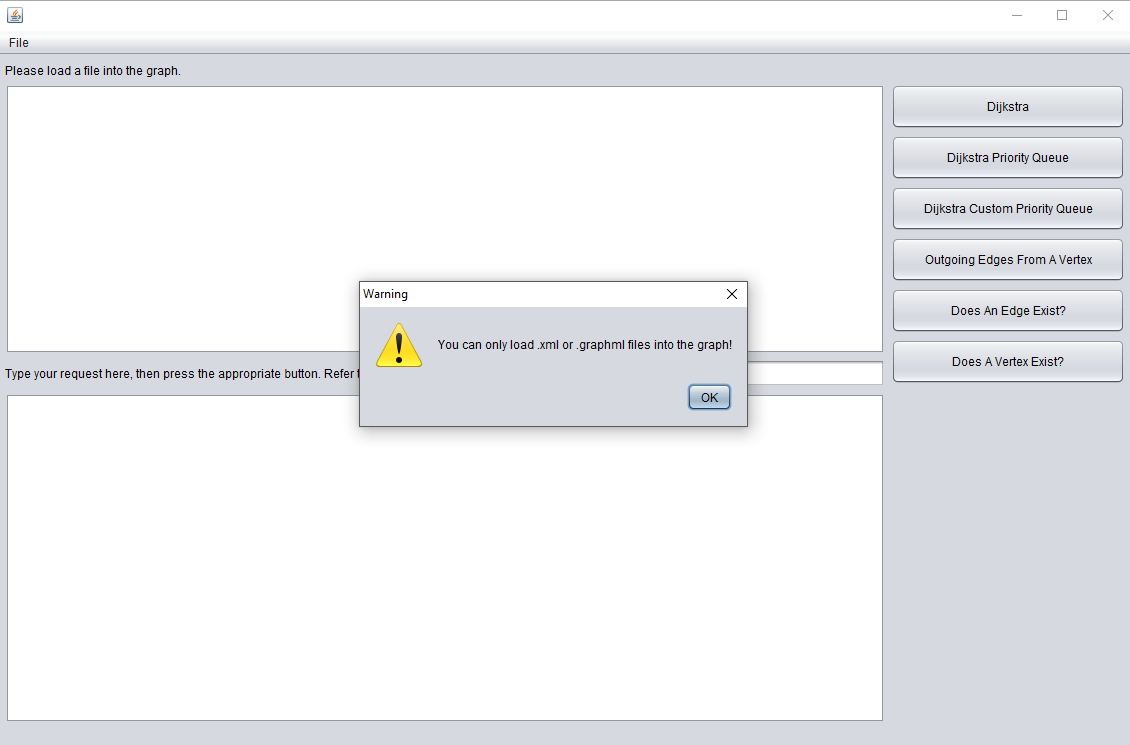
Figure 16 shows the testing results of trying to load multiple files into the graph.



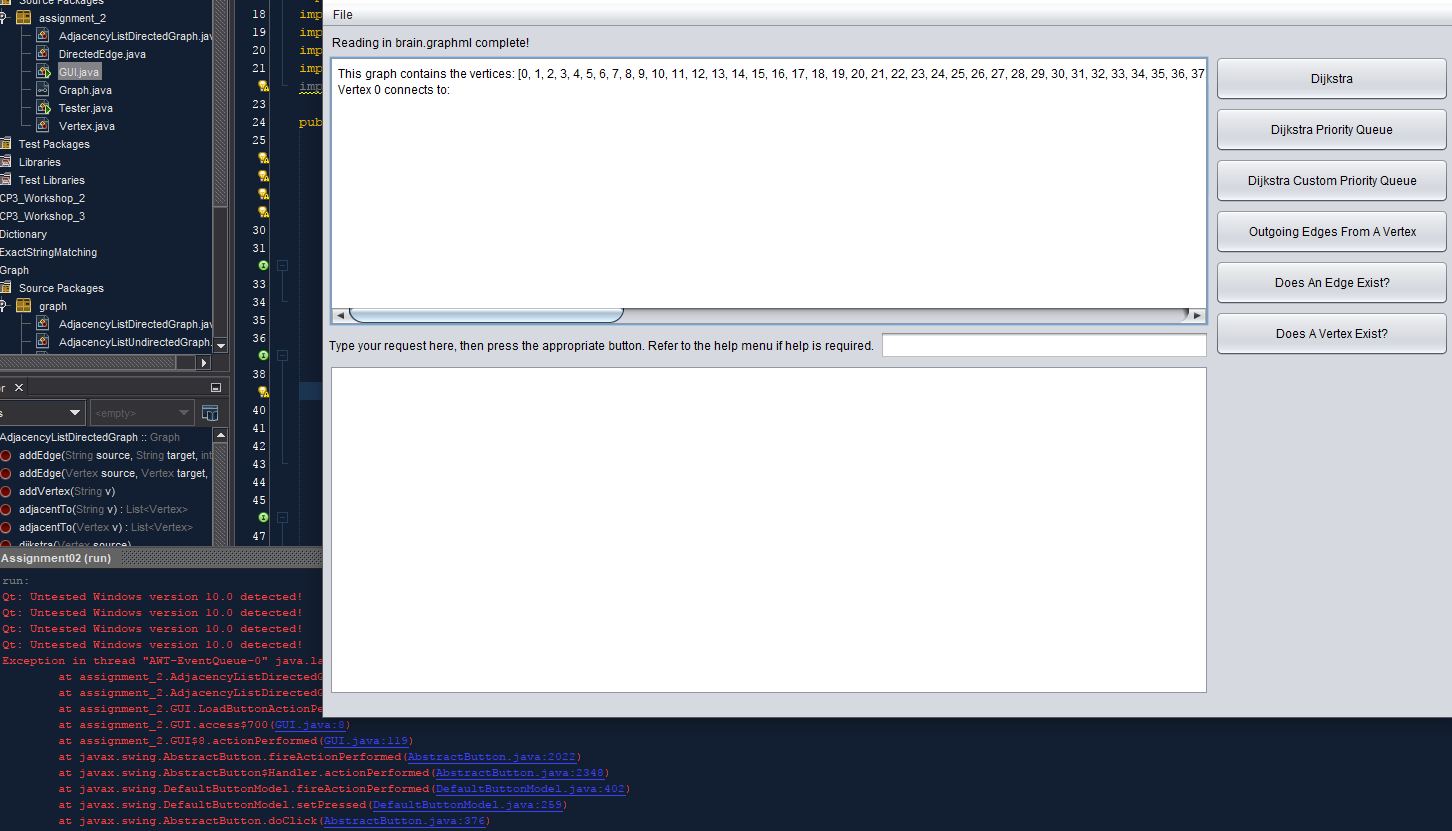
*Fig. 17.* *Testing help functionality.*

Figure 17 shows testing the help functionality that can be accessed via the file menu.

The thorough testing of the GUI revealed a few problems with incorrect input etc that were easily fixed but also problems like the following.



*Fig. 18.* *Testing different file extensions.*



*Fig. 19.* *Testing different file extensions.*

As shown in Figure 19, various problems were discovered when testing .graphml files. It was found to be because whenever disconnected nodes were inserted, they were not mapped to a set of edges. After discovering this, it was an easy fix.

# COMPLICATIONS & DESIGN DECISIONS

Various complications were encountered throughout the development of this program, some of which were minor although some influenced the restructure of methods thus influencing design decisions.

### “**readInFromFile**” method complications.

When developing the “**readInFromFile**” method, I discovered that I was only adding nodes mentioned within the edges elements. It was later discovered that disconnected nodes were not being added. As such, a node list was created within “**readInFromFile**” from which the nodes that weren’t added via adding the edges were then added to ensure the entire graph was read in correctly. Additional problems like not specifying a file extension when trying to read in a file resulting in IOExceptions of not specifying namespaces when trying to access child elements resulting in null lists were easily overcome once discovered..

### “**addEdge**” method complications.

Having to now store weights, I realized that an edge class would be the best solution and as such, the add edge method needed to be redesigned. I decided to map the source vertex to a set of edges using a HashSet for simplicity and speed when retrieving adjacent edges of a node. Various problems arose from doing it this way like trying to get a value from a key vertex but having nothing returned as the vertex I was trying to retrieve was modified in some way. This was overcome by modifying the compare to method within the vertex class.

### “**dijkstra**” method complications.

The largest issue faced was when I was developing the “**dijkstra**” methods. I originally wanted to store the previous vertex and the distance variables within the vertex class but for some unknown reason, I could not get it to work properly. Remembering the methods used within LAB3, namely the breadth first search method, I decided to use HashMaps within the adjacencyListDirectedGraph class in order to map a vertex to its previous vertex and a distance value. Not only was this easier to implement but it allowed me to use the “**pathTo**” method from LAB3 with minimal editing, thus saving time.

### **GUI** complications.

When implementing the ability to print the graph within the 1st textarea, I discovered there seemed to be no way to alter the automatically generated text from the graphical GUI construction tool within NetBeans. After a bit of research, I found a work around which allowed me the make the declaration of the 1st textarea static such that I could append text to it from within the ajdacencyListDirectedGraph class without having to instantiate a GUI object. Problem solved. Additional problems arose when the user attempted to read in files other than .xml or .graphml. this was fixed after doing some research and discovering file name extension filters. Many other little problems arose but most were due to logic errors etc. Easy fixes.

### Design Decisions.

* Chose to use a TreeMap for mapping a vertex to a HashSet of edges. Chose to use the TreeMap for its ability to sort and order keys, possibly improving performance, and chose to use a HashSet instead of an array because I assumed there would not be a need for storing duplicate edges.
* I chose to use an ArrayList to store the vertices within the graph for the ability to use Collections.sort before returning the list in order to make outputting the list easier.
* Chose to make a GUI as I had little experience making GUI’s and wanted to increase my knowledge.
* Chose to use the adjacency list method as most of the graphs given were sparse.
* I chose to implement a lot of the functionality within the adjacencyListDirectedGraph class like printing and Dijkstra’s algorithm as they seemed specific to my implementation of the graph. This may mean extra code will be needed if another graph class that implements the graph interface is desired but this way, it allows the programmer to make graph specific optimizations to the Dijkstra algorithm.
* Chose to implement 2 separate arrays within the MinHeap class in order to store the vertices and distances due to time constraints.

# FUNCTIONALITY

The final product is a relevantly clean, dynamic graphical user interface that serves to demonstrate the functionality of the graph data structure. It possesses the ability to load .xml and .graphml files into the data structure, the ability to execute 3 different variations of Dijkstra’s algorithm and display the run time, the ability to determine if specific vertices or edges exist and the ability to print the graph and outgoing edges from a vertex etc*.* In order to run the program, run the GUI.

Limitations worth noting:

* GUI resizability is not as fluid as I would like it. This is due to the inability to edit automatically generated code within netbeans and likely, my experience with Java.