Due 3/23

For programming assignment 4 we will be implementing a Graph class that implements the ConnectedGraphFunctions interface. Your class and filename will be "your_last_name_in_lower_case_Graph.java" and similarly for the class name.

I have posted Edge.java, GraphException.java, ConnectedGraphFunctions.java, and makeGraph.java as part of the assignment. Your program should throw a GraphException if a duplicate vertex is attempted to be added, along with not adding the duplicate vertex. Do similarly for duplicate edges. Remember that in an undirected graph, the edges (u, v) and (v, u) are considered the same. While in a directed graph, edges (u, v) and (v, u) are not the same. Your program should also throw a GraphException, and not add the edge, when adding edge (u, v) if either of the vertices u or v do not exist in the list of vertices.

The majority of the work for this program will be implementing the isConnected() function, which must work for both directed and undirected graphs.

We have already discussed an algorithm to determine if a graph is connected for undirected graphs or strongly connected for directed graphs. For our Graph class, we should have three fields, all should be private & final.

1) private final ArrayList<Integer> vertices, contains the vertices
2) private final ArrayList<Edge> edges, contains the list of edges
3) private final boolean isDirected, tells us if the graph is or is not directed

Your program needs to have two constructors, one that takes no parameters, defaulting to an undirected graph, and one that takes a boolean parameter, that specifies if the graph is directed or not. If the boolean parameter is true, then the graph is directed.

For my implementation of isConnected() I did the following:

1) We need to construct the set of connected vertices, which I called connectedSubset, from some starting vertex. I used a HashSet<Integer> to keep track of mine.
   a) To create an initially empty HashSet<Integer>
      i. java.util.HashSet<Integer> connectedSubset = new java.util.HashSet<>();
      ii. HashSet is an implementation of a set, so it only has unique elements, just like a real set
      iii. I typically check if an element is already contained in the HashSet using the HashSet.contains() method, so I never actually add any duplicates into my HashSet
2) I used the vertex at index 0 in the ArrayList<Integer> vertices as my starting vertex
   a) connectedSubset.add(vertices.get(0));
   b) You can use any vertex you want for the starting point
3) I also used an ArrayDeque<Integer>, as a queue, to keep track of when vertices were initially added to the connected vertices set
   a) java.util.ArrayDeque<Integer> newlyAddedVertices = new java.util.ArrayDeque<>();
   b) When I checked if a vertex was in connectedSubset, in 1) a) iii, if the vertex was not in connectedSubset, then it was added to both connectedSubset & newlyAddedVertices

4) The algorithm we described in class takes each vertex added to the connected vertices, connectedSubset, and checks if there are any edges from the vertex that go to vertices that are not in the connected vertices set, and if there are, they are added to the connected vertices set and to the newly added vertices list.
    a) The newly added vertices list is used to limit the number of times that we search the list of edges for a vertex to a single time

Below is pseudo code for the algorithm:

1) Define and initialize the connected set of vertices, connectedSubset, and the list of newly added vertices, newlyAddedVertices
    a) java.util.HashSet<Integer> connectedSubset = new java.util.HashSet<>();
    b) java.util.ArrayDeque<Integer> newlyAddedVertices = new java.util.ArrayDeque<>();
    c) connectedSubset.add(vertices.get(0));
    d) newlyAddedVertices.add(vertices.get(0));
    e) while newlyAddedVertices is not empty
        i. get the first element from newlyAddedVertices using int currentVertex = newlyAddedVertices.pollFirst()
            a) This will get the first element from newlyAddedVertices and also remove it from newlyAddedVertices
                1) The vertex is removed since we only want to do the below once for each vertex that is added to the connected vertices set
        ii. Loop over all of the edges
            a) For an undirected graph, if currentVertex is one of the vertices contained in the current edge, then check if the other vertex of the edge is in connectedSubset
                1) If it is not contained in connectedSubset
                    a) Add it to connectedSubset and to newlyAddedVertices
            b) For a directed graph, if currentVertex is the from vertex contained of the current edge, then check if the to vertex of the edge is in connectedSubset
                1) If it is not contained in connectedSubset
                    a) Add it to connectedSubset and to newlyAddedVertices
    f) Check if the connected vertices set, connectedSubset, contains all of the vertices of the graph
        i. If there are no duplicate vertices, you can just check if the size of the vertices ArrayList has the same size as the connected vertices set, connectedSubset
            a) If a duplicate vertex is attempted to be added to the graph, your code should throw a GraphException and not add the duplicate vertex

For an undirect graph, the above pseudo code will tell you if the graph is connected, by simply checking if connectedSubset contains all of the vertices of the graph.

For a directed graph, you need to do the following:

1) Use the above pseudo code to see if there is a forward path from the starting vertex, the one initially added to connectedSubset, to every other vertex in the graph
2) Do the same, but this time reverse all of the edges, to see if there is a reverse path

from every vertex to the starting vertex

3) For my code, I made a “private boolean isConnected(ArrayList<Edge> edges)” method, and then called this from my “public boolean isConnected()” method, passing the appropriate list of vertices to “isConnected(ArrayList<Edge> edges)”
   1. For a directed graph, there are two calls to “isConnected(ArrayList<Edge> edges)”, one with the original edges and one with the reversed edges
   2. For an undirected graph, there is one call to “isConnected(ArrayList<Edge> edges)” with the original edges

Some things to remember:

1) HashSet and ArrayDeque have an add() method to add an element, use this to add a vertex to them
2) HashSet has a contains() method to check if something is contained in the HashSet
3) ArrayDeque has a pollNext() methods that returns the first element in the ArrayDeque and also removes it
4) HashSet and ArrayDeque have a size() method to get the size of the HashSet or ArrayDeque
5) The Edge record has methods to get the from vertex, fromVertex(), and the to vertex, toVertex(), of the edge
   a) The methods are automatically provided by the compiler, since Edge is a record
6) ArrayList has a get(int ind) method to return the ArrayList element at index ind
   a) You can use this with a for loop to loop over the elements of the ArrayList of edges

```
for( int j = 0; j < edges.size(); j++ )
{
    Edge currentEdge = edges.get(j);
    // put code here to work with the current edge
}
```

   b) Or you can use ther enhanced for loop

```
for( Edge currentEdge : edges )
{
        // put code here to work with the current edge
}
```

The Graph.toString() methods is to return the following:

1) Line 1 is to be “G = (V, E)”
2) Line 2 is to be “V = {v_1,v_2,v_3, …,v_n}”, where v_i, i = 1, …, n are the vertices of G
   a) The vertices are to be listed in the same order they are added – this should be the natural ordering
3) Lines 3 is to be “E = {(u_1,v_1),(u_2,v_2), …,(u_k,v_k)}”, where (u_i,v_i) , i = 1, …, k are the edges of G
   a) The edges are to be listed in the same order they are added – this should be the natural ordering

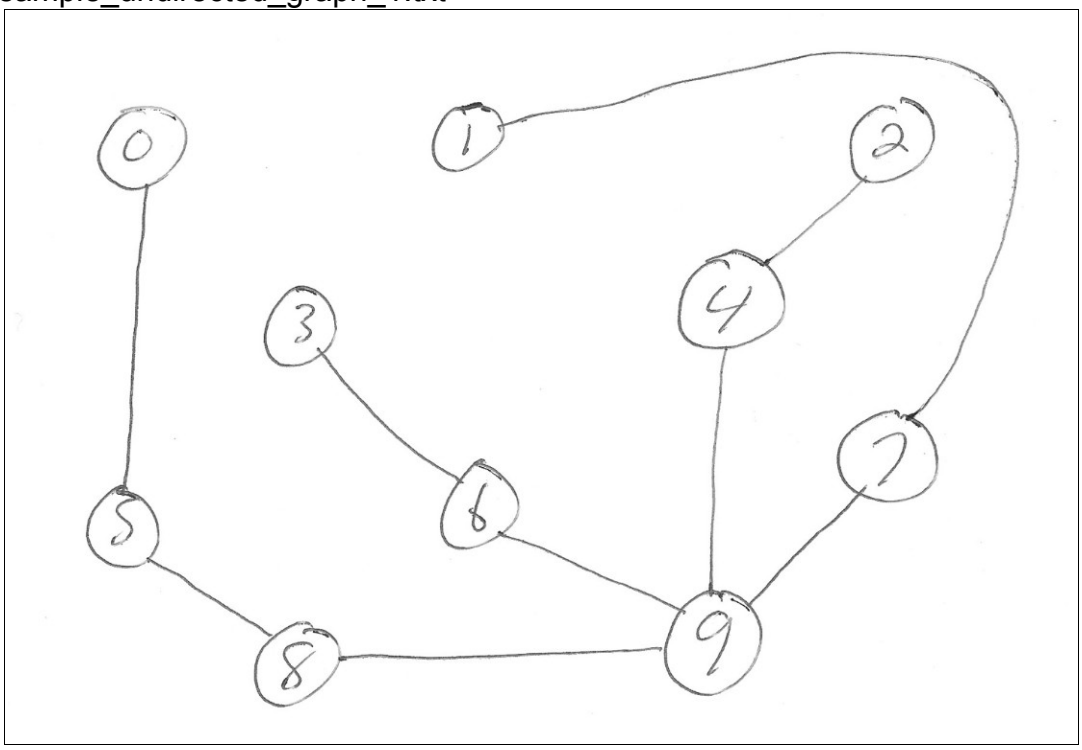
You should use a StringBuilder to build the output of toString(), since at least one of my graphs will be fairly large, and appending to an unmutable String will be slow. You can use

Due 3/23

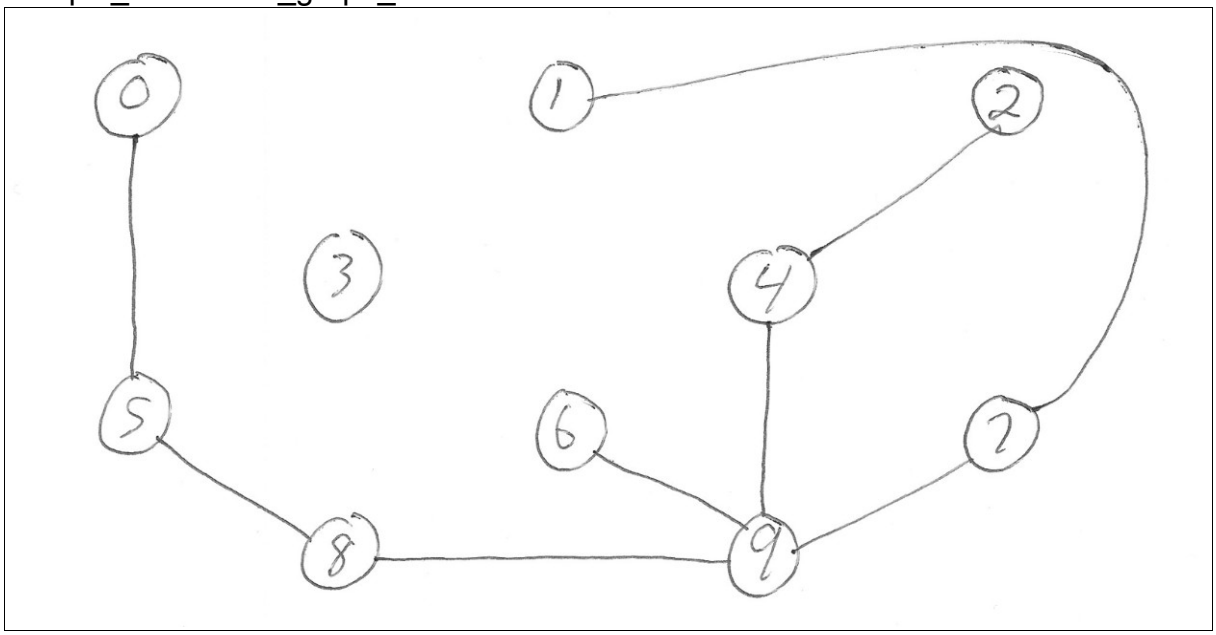
the StringBuilder.append() and StringBuilder.toString() methods to build the string and return it.

I have also posted four small sample graphs, two undirected and two directed.

sample_undirected_graph_1.txt

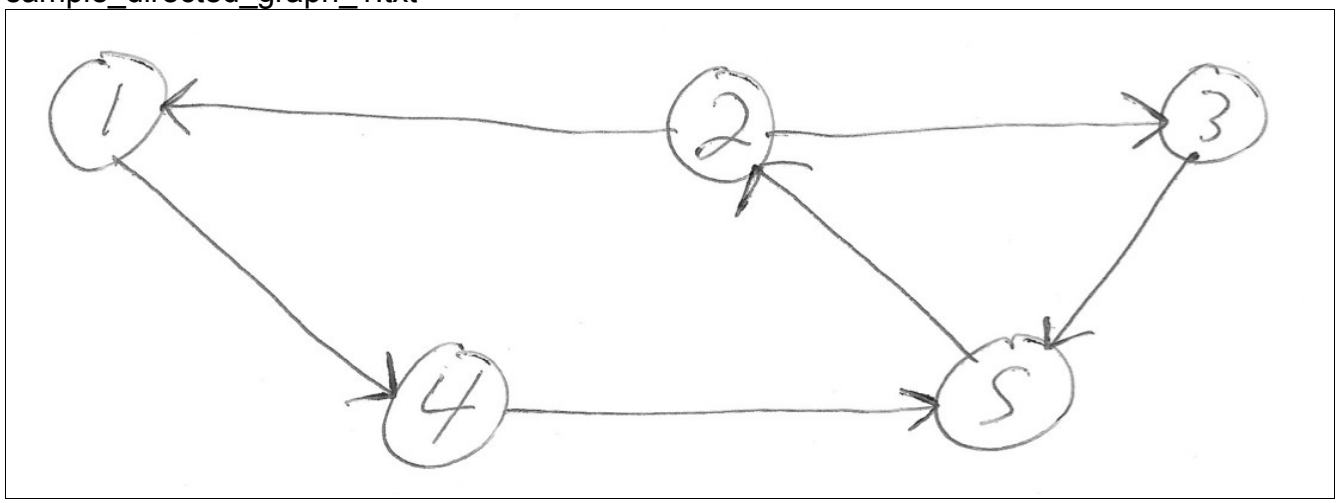


sample_undirected_graph_2.txt

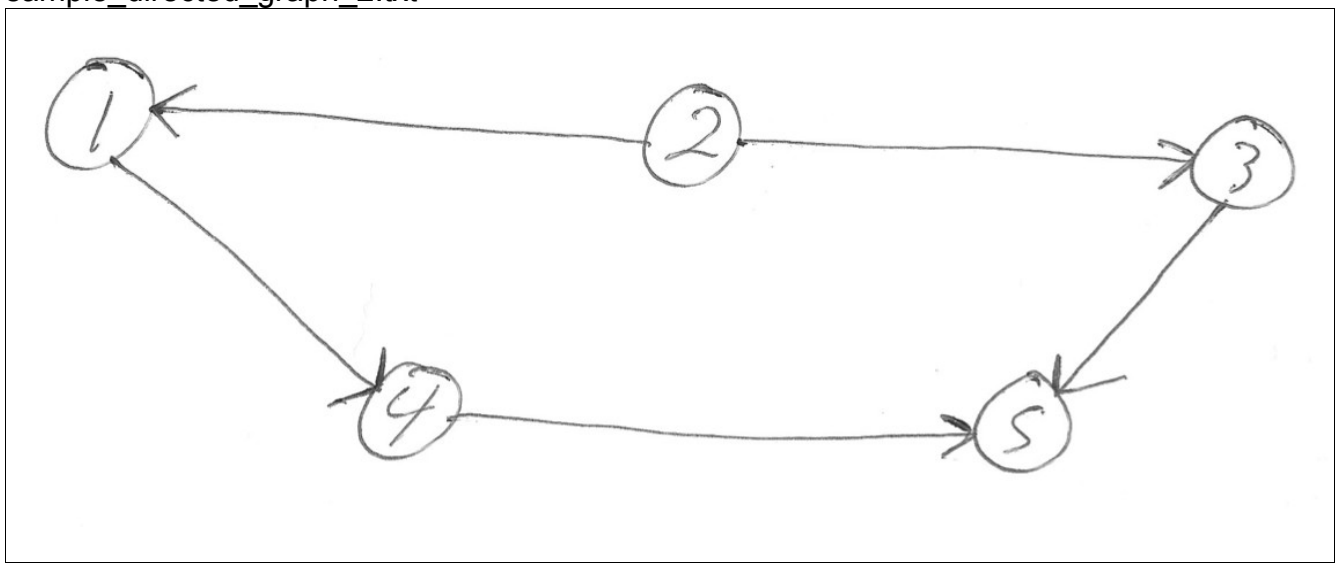


Due 3/23

sample_directed_graph_1.txt



sample_directed_graph_2.txt



To submit your program, e-mail it me at bi92798@binghamton.edu by midnight on the due date, with a subject of “cs 140 program 4. Attach the file “your_last_name_in_lower_case_Graph.java” to the e-mail. Do not put you java file in a zip file, tar file, rar file, or any other archive file.