CSCE 221 Report For Programming Assignment 6

Josh Zschiesche

CSCE 221-505

Jzschiesche1

Jzschieshce1@tamu.edu

UIN: 523000614

I) Description of your implementation, C++ features used, assumptions on input data

I implemented every function that was asked of us.

1. **Edge:**
   a. The class Edge was implemented by assigning the elements start, weight, and end to s, e, and w. This gives every object of type Edge these qualities.
2. **Vertex:**
   a. The class Vertex was implemented by implementing connectTo, vertex, and overloading the << operator so that it could print out to the screen.
   b. Vertex was implemented by passing in the label and then assigning that label to the new vertex.
   c. ConnectTo was implemented by passing in the end and making a new edge, and passing in all of the needed variables. That edge was then pushed back into the edgeList.
   d. The operator was overloaded by passing in the out stream and a Vertex, outing the label, assigning the temporary edgeList at that spot, and iterated through the temporary edgeList, outing the elements as it went along. After every element, a -1 was placed and a new line created. The out stream was then created.
3. **Graph:**
   a. Graph was implemented by passing in size then doing nothing with that, this is simply to hold the variable.

   b. Build graph was implemented by passing a character pointer, opening string streams to the file and reading every element then calling connectTo over and over again.

   c. displayGraph simply calls the overloaded operator.

   d. A custom function was implemented called return_size() that returns the size of the graph

   e. A custom implementation of DepthFirstSearch (Called DFS) was implemented that recursively calls DFS_Util() until the condition that every node in the list is visited. Until that happens, it created vectors and vectors of vectors that push hold all of the elements and their respective values. DFS also created a

map that maps each vertex to a Boolean, false is for the node is npt visited, and true is for when the node is visited.

f. DFS_Util is another function that “helps out” the real DFS by being recursive. This allows the regular DFS to not be recursive, and therefore, much easier to use. If an element is found to be false, the DFS_Util will make the map true, out the label and push it into a vector (more about this vector in other functions) It then iterated through the vertices and assigns the next edge to search and calls DFS_Util again. The DFS finished by returning a populated vector of vectors of intigers (this will be used when we have the acyclic components)

g. getTranspose is another custom function that gets the transpose of the passed in graph by iterating through all of the elements in the edgeList stored inside of the vertices vector and reversing the start and the end using the connectTo function.

h. The function displayStronglyConnected was another custom function that works by passing a reference to a vector of ints that are the lables. This will get traversed and will have the edge searched for, having an edge match means that there is a strongly connected component, and this will get couted as a strongly connected component.

The most important assumption made on this assignment is that we are reading in from a file and that we are not given a graph with a node that points to itself.

II) Three testing cases for checking correctness of your code

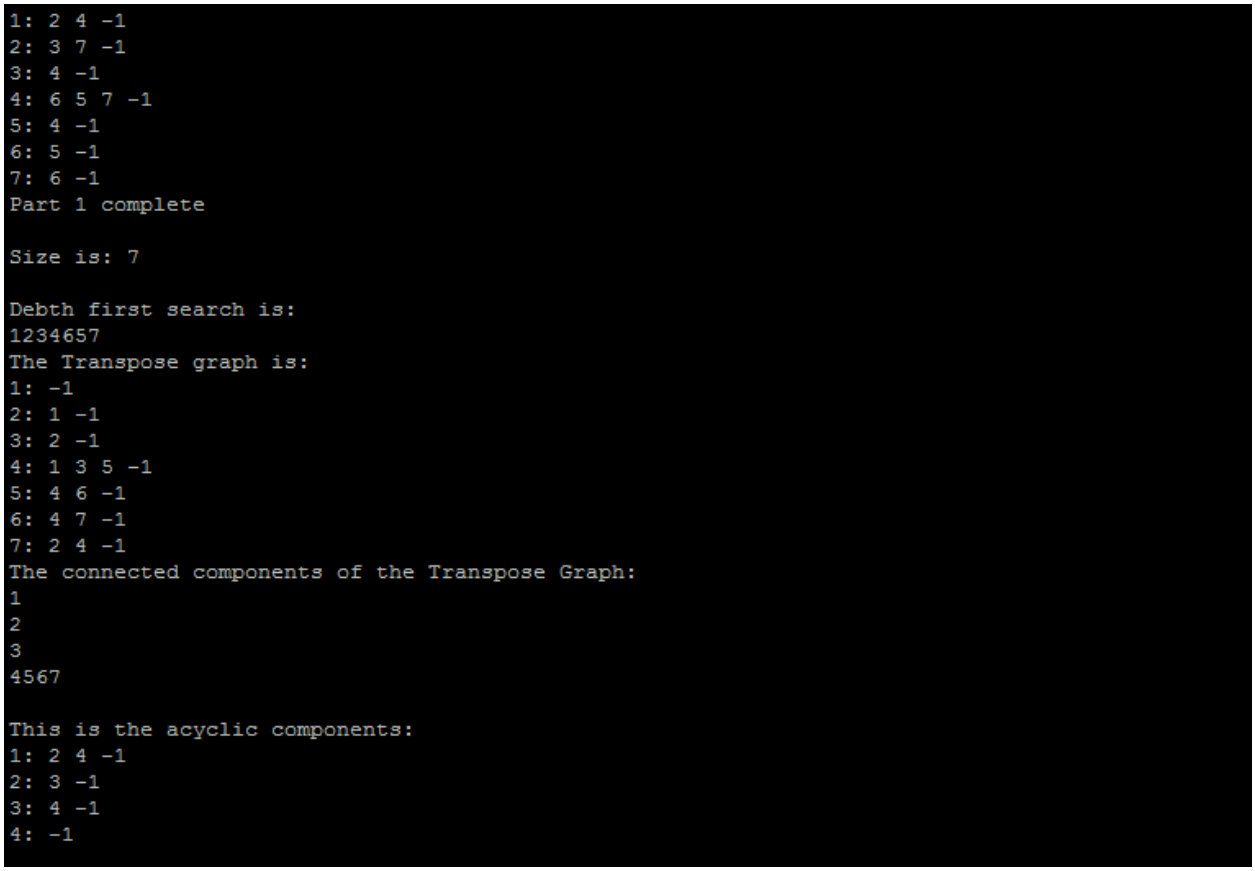
The Three testing cases are input.txt, test2.txt, and test3.txt. All should be in the submitted .tar file.

Input.txt:
This gives all of the correct output.

```
[jzschiesche1]@build ~/CSCE_221_Stuff/Programming_Assignment/PA6/NM> (22:07:43 04/24/16)
:: ./main input.txt
1: 2 4 5 -1
2: 3 4 7 -1
3: 4 -1
4: 6 7 -1
5: 4 -1
6: 5 -1
7: 6 -1
Part 1 complete

Size is: 7

Debth first search is:
1234657
The Transpose graph is:
1: -1
2: 1 -1
3: 2 -1
4: 1 2 3 5 -1
5: 1 6 -1
6: 4 7 -1
7: 2 4 -1
The connected components of the Transpose Graph:
1
2
3
4567

This is the acyclic components:
1: 2 4 -1
2: 3 4 -1
3: 4 -1
4: -1
```

Test2.txt:

I am pretty sure this gives the correct output, though I have not had time to fully test this file on paper.



Test3.txt:
I am pretty sure this gives the correct output, though I have not had time to fully test this file on paper.

III) Running Time expressed in terms of Big-O

Depth First Search = O(nodes+vertices)
Graph = O(1)
connectTo =O(1)
overloaded operator = O(n)
buildGraph = O(vertices+edges)
return_size = O(1)
getTranspose = O(n^2)
displayStronglyConnected = O(n^2)

Note: I did not have time to justify these or check if there are correct.

IV) Description of a real life application where your program can be used

One possible use of this data structure in the real world is in the field of sociology. Social science researchers can use graph theory to measure the interconnectedness of various populations. If people are the nodes, and the connectedness is factors such as environment, money, presige, time spent together, etc, etc, social scientists can perform this algorithm (or have a computer scientist do it for them) and find out the most connected groups in a population, and what factors make a more connected population. This will also tell them which people specifically are the most connected in a population.

Another possible use is to speed dijstras and prim's algorithms by a factor of log(n), also making the applications that run those algorithms speed up by a factor of log(n).

V) How to Run:

Simply put everything into a directory that was in the tar file, and then type make clean and then make and then type ./main [input file] (without the breackets and with an actual file name e.g. input.txt)