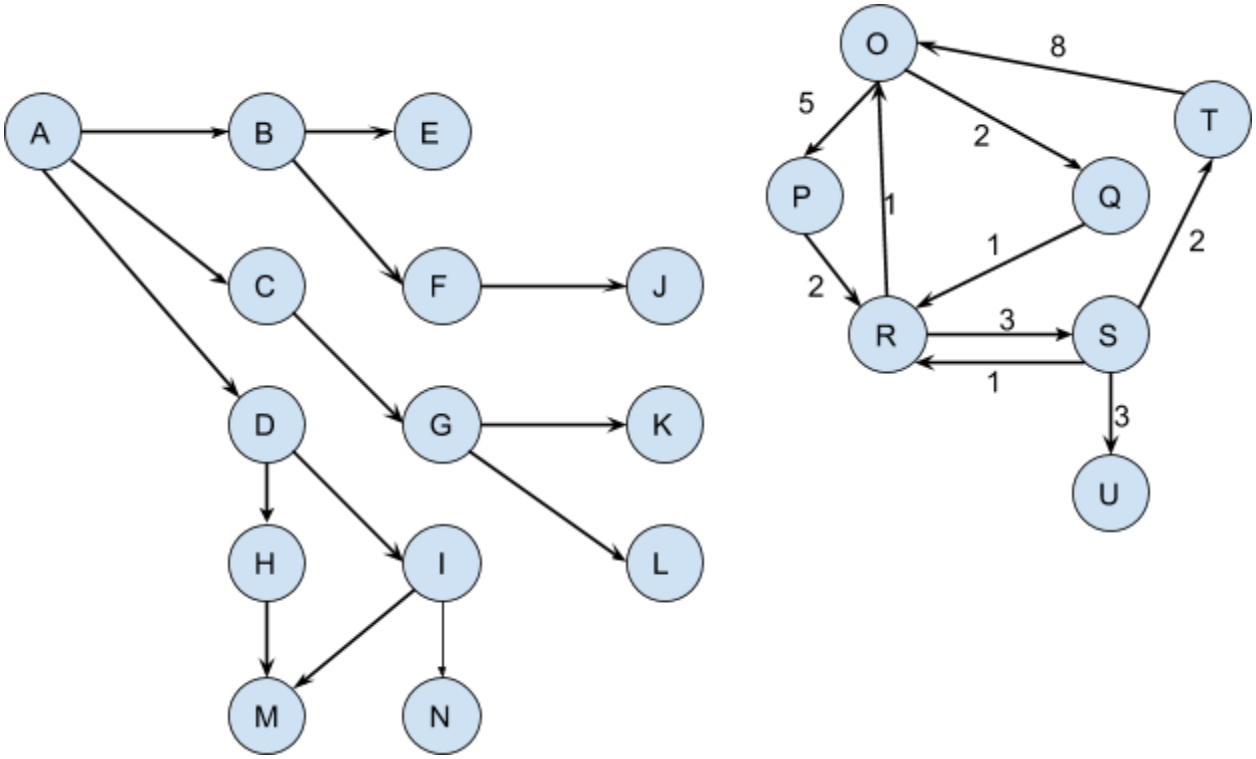
Ass 3: Graphs

Implement a Graph class based on the template below and test your class for depth-first search, breadth-first search and dijkstra's shortest path.

`graph.h` file is provided. All public functions for the Graph class has to be implemented. You might also want to implement Edge and Vertex classes.

`ass3.cpp` provided tests for some of the major functionality. You should expand this to include additional tests. `ass3.cpp` reads graphs from `graph0.txt`, `graph1.txt`, `graph2.txt` text files or some of the tests.

Visual representation for `graph2.cpp` is below:



Depth-first search from `O` gives: `OPRSTUQ`

Breadth-first search from `O` gives: `OPQRSTU`

Dijkstra from `O` gives weights of `[P:5][Q:2][R:3][S:6][T:8][U:9]` and previous values of `[P:O][Q:O][R:Q][S:R][T:S][U:S]`

Submissions

As always expected when programming, comment clearly and thoroughly. Clearly state any assumptions you make in the beginning comment block of the appropriate place, e.g., the class definition. Comments in the class definition file should describe the ADT, all functionality, and assumptions so someone could use the class and understand behavior and restrictions. Pre and post conditions are fine, but not required. See the example on Assignments page for a well-documented program.

You do NOT need to handle data type errors due to bad input.

I will run my own main to test your code. The main function provided doesn't test your program fully, so you need to supplement it.

Submit a single zip file, `ass3.zip` with all your files in it. When unzipped `ass3.zip` must create a directory named `ass3` and place all the files in that directory. See http://faculty.washington.edu/pisan/cpp/creating-zip.html on how to create a zip file.

Required files: `graph.h, graph.cpp, ass3.cpp, output.txt.` See Connecting and compiling files on linux labs on how to create output.txt

Optional files: `edge.h, edge.cpp, vertex.h, vertex.cpp`, and possibly others

Once your code is working on your own machine, test it once more on the linux machines (you have been testing incrementally and using valgrind, right?). See Connecting and compiling files on linux labs

Under unix, compile your code using

```
g++ -std=c++14 -g -Wall -Wextra *.cpp -o ass3
```

and create the `output.txt` file following the instructions on that page.

DO NOT include unnecessary files, only submit what is required. DO NOT zip up your Debug directory, your project directory, etc. Canvas will automatically rename your zip file ass3-2.zip, ass3-3.zip, etc depending on how many times you submit it. *This is fine*. Do not change your zip file name.

Hints and Suggestions

Lightweight `Edge` and `Vertex` classes, each with their own .h and .cpp file will make the implementation cleaner. These classes do not need any public constructors or functions since they are not visible to the outside. Use "`friend class Graph;`" to enable the `Graph` class to be able to access private variables and functions.

`Edge` is the simplest class, it only has two Vertex pointers and a weight.

`Vertex` needs to store all outgoing edges. You can use `vector<Edge*>` or another container type to store pointers to the edges. You must keep the list of edges sorted. You can use the sort function from algorithm class or implement your own version. `Vertex` is responsible for creating `Edge` objects, deleting them when they are disconnected or when the vertex is finally deleted.

`Graph` needs to keep track of all vertices. You can use a `vector<Vertex*>` type container or since we frequently have to find the corresponding vertex based on a string, you might want to use `map<string, Vertex *>` container for easy access. As a bonus, map will automatically sort them

as well. With using STL containers, whenever possible you should use a range based for loop rather than having an explicit index. See https://en.cppreference.com/w/cpp/language/range-for for examples.

[Update 10/24: You could use store objects instead of pointers to objects in the containers, BUT since containers make a copy of the object, you have to be extra careful about not making extra copies. When retrieving an object from the container, you must make sure it is returned and assigned by reference, so a copy is not created.

If you go with this approach, you could use `vector<Edge>` instead of `vector<Edge*>` as long as the `Edge` object is storing `Vector*` objects or `string` objects and and not actual `Vertex` objects. Similarly, you could store `vector<Vertex>` in `Graph` rather than `vector<Vertex*>` but you then must make sure that each time a `Vertex` is retrieved from the container, it is retrieved by reference and an accidental copy is not made.]

The STL containers already know their own size, so a separate variable for keeping track of the number of vertices is not necessary. However, since edges are stored in the `Vertex` objects, the `Graph` class will need to keep track of how many edges are created using a variable.

`findVertex` and `createVertex` are useful private functions for `Graph` class since these operations are frequently needed. Similarly `dfsHelper` and `bfsHelper` are useful in breaking up the code. `stack` and `queue` libraries from STL can make your implementations easier and cleaner.

Pay special attention to `Connect` and `Disconnect` functions so that a vertex is not connected to itself or has multiple edges to the same vertex. These functions also create new vertices if the graph does not have them already. Be careful not to create multiple vertices with the same label.

`Dijkstra` is the most complex function you will implement in this assignment. Take advantage of STL libraries such as `set` and `priority_queue`. Since putting pointer objects into a priority queue would not make sense (who wants to compare memory addresses!), you will need to create `pair` objects using `make_pair(pathCost, vertexPointer)`. You can setup the `priority_queue` with

```
using IV = pair<int, Vertex *>;
priority_queue<IV, vector<IV>, greater<IV>> pq;
```

where `using` is the modern equivalent of `typedef`. `Dijkstra` function takes two map objects by reference which slowly get filled as the algorithm visits all vertices. The `ass3.cpp` file has a `map2string` function to turn these maps into printable format for comparison.

For `ReadFile` to work properly, the graph files must be in the same directory as the executable. Different IDEs have different ways of setting working directory. For CLion, putting the `graph.txt` files into the cmake-build-debug directory is the easiest solution.

Take the time to design your functions and classes. If you do not approach this assignment systematically, you will generate lots of duplicate code which will make it impossible to debug. For comparison, my `graph.cpp` file has less than 200 lines of code (excluding comments), but it took many passes for it to shrink to that size.