Programming Assignment #6

Due Sunday, April 10, 2016 at 11:59 pm

This assignment will exercise your skills in constructing a graph and implementing a graph algorithm. The task is to implement a class called *ALGraph*, which represents a graph. The "AL" stands for adjaceny list, since the graph will hold the adjacency information using *lists* (as opposed to using an adjacency *matrix*). In addition to the constructor, there are only two methods that a client will use to construct the graph: *AddDEdge* and *AddUEdge*. *AddUEdge* is just a convenient wrapper around *AddDEdge*.

Method	Description
`ALGraph(unsigned size)`	Constructs an ALGraph containing *size* nodes with IDs 1 through *size*.
`~ALGraph()`	Destructor.
`AddDEdge(unsigned source, unsigned destination, unsigned weight)`	Adds a *directed* edge from *source* to *destination* to the graph. *source* and *destination* are valid node IDs. **(***source*, *destination***)**
`AddUEdge(unsigned node1, unsigned node2, unsigned weight)`	Adds an *undirected* edge to the graph. This function will actually add two directed edges to the graph. One from *node1* to *node2* and another from *node2* to *node* 1. *node1* and *node2* are valid node IDs. **{***node1*, *node2***}**

```
typedef std::vector<std::vector<AdjacencyInfo>> ALIST;

class ALGraph
{
  public:
    ALGraph(unsigned size);
    ~ALGraph();
    void AddDEdge(unsigned source, unsigned destination, unsigned weight);
    void AddUEdge(unsigned node1, unsigned node2, unsigned weight);

    std::vector<unsigned> SearchFrom(unsigned start_node, TRAVERSAL_METHOD method) const;
    ALIST GetAList() const;
    std::vector<DijkstraInfo> Dijkstra(unsigned start_node) const;
    static bool ImplementedSearches(); // for extra-credit
  private:
    // private structs, classes, methods and fields
};
```

Dijkstra's algorithm

There is a **const** method that is at the heart of the assignment: *Dijkstra*. Not surprisingly, the *Dijkstra* method implements Dijkstra's algorithm as it was discussed in class. The return value is a vector of *DijkstraInfo* (see ALGraph.h), which is a struct containing the information regarding the cost to reach each node in the graph as well as the path that led to each node. The path is just a vector of node IDs. See the driver and output for examples. There is also a public **const** method to get the adjacency list for the driver to display. The return is an ALIST (a typedef). See ALGraph.h for the definition of *AdjacencyInfo*. The edges in the ALIST are sorted by weight. If there are multiple edges with the same weight, they are then sorted by node ID, smallest to largest. (See the output for examples, specifically, test **Dijkstra4** in the driver.) Because the two primary methods for this assignment are **const**, you may need to use the **mutable** keyword in this assignment. (Don't cast the **const** away.)

Using the STL

Since graphs are composed of other primitive data types, you may find it very useful to leverage the containers and algorithms in the STL. Three of the public methods require `std::vector`, but you may want to use other standard types in your implementation (e.g queue, stack, map, set, algorithms, etc.) This can greatly reduce the amount of coding required for this assignment. Since you are using containers and algorithms from the STL, you may need to include the appropriate header files in ALGraph.h. This will be allowed. The ALGraph class itself is not a templated class. **To be clear**: Three of the public methods return specific STL containers. You DO NOT have to limit yourself to using these containers in your algorithms. You just need to be sure to construct containers of these types when you communicate with the client (driver). I fully expect that students at this programming level will create other classes and use other containers to support their algorithms. You also cannot modify the `ALIST`, `DijkstraInfo`, or `AdjacencyInfo` structs. Also, any helper functions or classes that you create must be in the **private** section of the class, or in an unnamed namespace in the implementation file.

Traversing the Graph (Extra-credit: 25 points)
The *SearchFrom* method finds all of the nodes that are reachable from the starting node, which is provided as the first parameter. This method takes a TRAVERSAL_METHOD as its second parameter. This parameter determines the way the search proceeds. Of course, since there are many different traversals possible from any start node, we will use a deterministic approach. That is, when presented with multiple edges to follow, you will follow them in *decreasing* order. This means that if you have three edges with weights 7, 9, and 3, you will follow the edge with weight 9 first, then the edge with weight 7, and finally, the edge with weight 3. This applies to both depth-first and breadth-first searching. If any edges have the same weight, you will use the destination's node ID to determine which edge to follow first. The smaller ID will be followed **before** a larger ID. **To be clear**, you first sort the edges by weight, and if more than one outgoing edge have the same weight, you will sort by ID, smallest to largest. There is an example in the driver and posted on the web page. Also, remember that when using a stack (depth-first), you need to push them in the reverse order so they are pulled out (popped) in the correct order, as I demonstrated in class. In order to earn the extra-credit, You **MUST HAVE ONLY ONE FUNCTION** that implements the traversals as the only difference between a breadth-first and a depth-first traversal is the container used to store unvisited nodes. This function must be named **Traverse.** (This will make it easier for the grader to find.) This function will be templated. There will be no "checking" done in this function to determine how to proceed. The template parameters should perform the correct functionality. This code was shown during the lectures and posted on the web. You just need to adapt it for this assignment. It will require a little extra design and thought on your part, which is another reason this is extra credit, especially since the code for traversing a graph is also posted in the notes. (You will have to adapt it to your needs.)

TO BE PERFECTLY CLEAR: You DO NOT have to limit yourselves to the structures used to communicate with the driver. I actually don't expect anyone to use them for anything other than communicating with the driver. However, if you want to use them for other purposes, that is fine, but you don't have to use them and you can't change them. Please don't complain about how the structures don't contain enough information and methods to keep track of all of the traversal details while implementing Dijkstra's algorithm. They aren't designed for that task.

Scoring
Since there is an extra-credit component, the scoring will be different. There are two tiers of credit, and only the first one must be implemented. The first one is to implement Dijkstra's algorithm. This algorithm follows the example I did in class. The pseudo-code is already provided on the website, so you should use that, along with your notes, as a starting point. You must implement this method before moving on to the extra credit portion. **Extra credit is only accepted if Dijkstra's algorithm passes all of the tests.** Turning in just a functioning Search algorithm is insufficient to earn any points. Both algorithms require you to build an adjacency list correctly, so this will already have been done for the implementation of Dijkstra's algorithm. If you want your code to be tested for the search algorithms (*SearchFrom),* you must return true from the static *ImplementedSearches* method.

Testing
Like other assignments in this class, there is a plethora of sample tests and output. Please be sure that you can pass all of the tests. The website for this assignment has more examples and details, including some pointers on using the STL with this assignment. Also, because all of the algorithms depend on an adjacency list, that is the first thing you need to code. If you don't have a **correctly functioning adjacency list**, nothing else will work. Make sure you can add nodes and edges to your data structure and create a correct adjacency list. Once you have that, you will be able to work on the other algorithms.

What to submit
You must submit your header/implementation files (ALGraph.h and ALGraph.cpp) in a .zip file to the appropriate submission page as described in the syllabus.

Files	Description
ALGraph.h	The header file for the ALGraph class. **No implementation is allowed in this file.** The public interface must be exactly as described above.
ALGraph.cpp	The implementation file. All implementation for the methods goes here. You must document the file (file header comment) and functions (function header comments) using Doxygen. Don't forget to include comments indicating why you are **#includ**ing certain header files, especially for the STL headers.

Usual stuff
Your code must compile using the compilers specified in this assignment to receive credit. Note that you must not submit any other files in the zip file other than the 2 files specified. Details about what to submit and how are posted on the course website and in the syllabus.

Make sure your name and other info is on all documents.