**Programming Assignment Graph**

This assignment will exercise your skills in constructing a graph and implementing Dijkstra's algorithm. The task is to implement a class called ALGraph, which represents a graph. The "AL" stands for adjacency 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: AddDEdgeand AddUEdge*.* AddUEdgeis just a convenient wrapper around AddDEdge*.*

|  |  |
| --- | --- |
| **Method** | **Description** |
| ALGraph(unsigned size) | Constructs an ALGraph containing sizenodes with IDs 1 through *size*. |
| ~ALGraph(void) | 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**);

**void** AddDEdge(**unsigned** source, **unsigned** destination, **unsigned** weight);

**void** AddUEdge(**unsigned** node1, **unsigned** node2, **unsigned** weight);

ALIST GetAList(**void**) **const**;

std::vector<DijkstraInfo> Dijkstra(**unsigned** start\_node) **const**;

**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 *Dijkstra* is **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**: Two 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.

**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.

# 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 algorithm.

# What to submit

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

|  |  |
| --- | --- |
| **Files** | **Description** |
| ALGraph.h | The header files 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 3 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.