cs280-a4-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: `AddDEdge` and `AddUEdge`. `AddUEdge` is just a convenient wrapper around `AddDEdge`.

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 outputs 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 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.

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.

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.

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.

Using the STL will greatly simplify your tasks. However, if you've not used the STL much outside of CS170 and CS225, you may run into some problems with your coding.

Creating a priority queue with the default behavior is trivial:

```
  // Create a PQ that uses the default std::less<> class.
  // std::less uses operator< so whatever you place in the PQ
  // must support operator< (the MyAdjInfo implements this).
std::priority_queue<MyAdjInfo> edges;
```

This priority queue will keep things sorted from smallest to largest.

Creating a priority queue that provides a different behavior requires slightly more effort:

"'C++ // Create a comparison functor to override the default behavior. // We can use std::greater to get the correct behavior. MyAdjInfo // also must implement operator> for this. std::greater cmp;

// Create a PQ using the functor above. Since you are providing a value for // the 3rd template parameter, you must provide all three parameters. // (Remember, default parameters in C++ must be defaulted right-to-left.) // You must also pass the comparison functor to the constructor. std::priority_queue<MyAdjInfo, std::vector, std::greater > edges(cmp);

Now the queue will sort in the reverse order. You can use these priority queues to help implement the algorithms.

Creating the same interface for stacks and queues

```
// This "fixes" the problem where the STL has a different method
// for removing from queues and stacks. Now, they both have the
// std::stack interface: the top() method simply calls front()

// Inheritance - Just add a top method and forward to front
template<typename T>
class My_queue : public std::queue<T>
{
  public:
       // this is required when calling a base class method in a templated hierarchy
    T top() { return this->front(); }
};
```

or

```
// Aggregation - Forward all calls to the internal queue object
template<typename T>
class My_queue
{
  public:
    T top() { return q.front(); }
    void push(T item) { q.push(item); }
    bool empty() { return q.empty(); }
    void pop() { q.pop(); }
  private:
    std::queue<T> q;
};
```

Setting up infinity in C++ code. Put this in your .cpp file. DO NOT USE MAGIC NUMBERS.

```
const unsigned INFINITY_ = static_cast<unsigned>(-1);
```

I use the underscore because I've discovered that some libraries have already defined INFINITY.

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. 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.

Files Provided

The ALGraph interface[1].

Sample driver[2] containing loads of test cases.

Diagrams[3] to help your understanding.

There are a number of sample outputs as usual in the **data** folder.

The "normal" tests

- `TestDijkstra0(1)`'s output txt file[4] and it's gviz txt file[5] for the gviz tool to churn out the visual gviz img file[6].
- `TestDijkstra1(1)`'s output txt file[7] and it's gviz txt file[8] for the gviz tool to churn out the visual gviz img file[9].
- etc…

Graphviz tool

To experiment with the Graphviz[10] package on your own computer, you'll need to install it.

There is a GUI front-end[11] that Prof Matthew Mead (DigiPen Redmond) created. It's more convenient than the one that comes with the Graphviz package. Sadly, it's a Windows program. (It's still a work-in-progress) It will work under Wine on Linux, but it won't view the graphics output.

Nevertheless you can install graphviz by some simple googling based on your platform, then use the command line `dot` command, e.g.,

```
dot -T png -o graph.png graph.txt
```

The stress tests

These should run typically in less than 3secs on a modern machine, e.g., Win10 on core i7-xxxx, 8GB, L1 256KB, L2 1 MB, L3 8 MB.

- `TestDijkstra0(1)`'s output txt file[12] and it's gviz txt file[13] for the gviz tool to churn out the visual gviz img file[14].
- etc…

Compilation:

These are some sample command lines for compilation. GNU should be the priority as the executable will be used for grading. You STILL need to compile cleanly with Microsoft cl though.

[1]code/ALGraph.h
[2]code/driver-sample.cpp
[3]docs/diagrams.pdf
[4]data/output-D0-1.txt
[5]data/Dijkstra0-gviz.txt
[6]data/Dijkstra0-gviz.txt.png
[7]data/output-D1-1.txt
[8]data/Dijkstra1-gviz.txt
[9]data/Dijkstra1-gviz.txt.png
[10]http://www.graphviz.org/
[11]tools/GVizEd.exe
[12]data/output-Big-10x100.txt
[13]data/TestBig-10-100-gviz.txt
[14]data/TestBig10-100.txt.png

GNU g++: (Used for grading)

```
g++ -o vpl_execution ALGraph.cpp driver-sample.cpp \
    -std=c++14 -pedantic -Wall -Wextra -Wconversion -Wno-deprecated
```

Microsoft: (Good to compile but executable not used in grading)

```
cl -Fems ALGraph.cpp driver-sample.cpp \
   /WX /Zi /MT /EHsc /Oy- /Ob0 /Za /W4 /D_CRT_SECURE_NO_DEPRECATE
```

Deliverables

You must submit your program files (header and implementation file) by the due date and time to the appropriate submission page as described in the syllabus.

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 #including certain header files, especially for the STL headers.