CS6771 Assignment Three 2016

Generic Directed Weighted Graph (GDWG)

Due: 11:59pm Friday 16 September 2016 **(Extended to 11:59pm Monday 19 September 2016)**

Worth: 10%

1. Aims

- Function and Class Templates
- Smart Pointers
- Exception Handling
- Lambda Functions

In this assignment, you will write a Generic Directed Weighted Graph (GDWG) with value-like semantics in C++. Both the data stored at a node and the weight stored at an edge will be of generic types. Both generic types may be different. For example, here is a graph with nodes storing std::string and edges weighted by int:

```
gdwg::Graph<std::string,int> g;
```

Formally, this directed weighted graph G=(N,E) will consist of a set of nodes N and a set of weighted edges E. Give a node, an edge directed into it is called an *incoming edge* and an edge directed out of it is called an *outgoing edge*. The *in-degree* of a node is the number of its incoming edges. Similarly, the *out-degree* of a node is the number of its outgoing edges. Given a directed edge from src to dst, src → dst, src is the *source* node and dst is known as the *destination* node. G is a multi-edged graph, as there may be two edges from the same source node to the same destination node with two different weights. However, all nodes are distinct, as they contain different data values.

2. Internal Representation

You must use smart pointers to represent the nodes and edges in your implementation.

3. Operations

There are many ways to implement a generic directed weighted graph. For our purposes, a node can contain the data stored at that node and a set of its outgoing edges and an edge may be a tuple containing its source node, its destination node and the weight associated with this edge. However, your solution does not have to follow strictly this suggestion, except that it must adhere to the public interface defined below.

In what follows, we assume the graph template class is declared as:

```
template <typename N, typename E> class Graph;
```

where N is the Node value data type and E is the Edge weight data type.

Member	Description and Hints	Examples
Constructor	A user-defined or compiler-synthesised default constructor for an empty graph.	`gdwg::Graph<std::string,int> g;`
Copy and Move Constructors	User-defined or compiler-synthesised copy and move constructors as required.	
Copy and Move Assignment operators	User-defined or compiler-synthesised copy and move assignment operators as required.	
bool addNode(const N& val)	Adds a new node with value val to the graph. This function returns true if the node is added to the graph and false if there is already a node containing val in the graph (with the graph unchanged).	`std::string s = "a";` `g.addNode(s);` `a.addNode("b");`
bool addEdge(const N& src, const N& dst, const E& w);	Adds a new edge src → dst with weight w. This function returns true if the edge is added and false if the edge src → dst with weight w already exists in the graph. A std::runtime_error is thrown if either src or dst is not in the graph.	`std::string u = "c";` `g.addEdge("a",u,1);`
bool replace(const N& oldData, const N& newData);	Replaces the original data, oldData, stored at this particular node by the replacement data, newData. If no node that contains value oldData can be found, then a std::runtime_error is thrown. This function returns false if a node that contains value newData already exists in the graph (with the graph unchanged) and true otherwise.	`g.replace("a","e");`

void mergeReplace(const N& oldData, const N& newData);	Replaces the data, oldData, stored at one node, denoted OldNode, with the data, newData, stored at another node, denoted NewNode, in the graph. If either node cannot be found in the graph, then a std::runtime_error is thrown. After the operation has been performed successfully, every incoming (outgoing) edge of OldNode becomes an incoming (outgoing) edge of newNode, except that duplicate edges must be removed. Note that the edges that connect OldNode and NewNode are handled identically by this edge merging rule. See test6.cpp for an example.	`g.mergeReplace("c","e");` `// node "c" is destroyed`
void deleteNode(const N&) noexcept;	Deletes a given node and all its associated incoming and outgoing edges. This function does nothing if the node that is to be deleted does not exist in the graph. Hint: if you are using weak ptrs for edges you may be able to do this quite simply. This function should not throw any exceptions.	`g.deleteNode("b");`
void deleteEdge(const N& src, const N& dst, const E& w) noexcept;	Deletes an edge from src to dst with weight w, only if the edge exists in the graph. No exceptions are thrown.	`g.deleteEdge("b","c",1);`
void clear() noexcept;	Remove all nodes and edges from the graph. New nodes or edges can be added to the graph afterwards.	`g.clear();`
bool isNode(const N& val) const;	Returns true if a node with value val exists in the graph and false otherwise.	`g.isNode("a")`
bool isConnected(const N& src, const N& dst) const;	Returns true if the edge src → dst exists in the graph and false otherwise. This function throws a std::runtime_error if either src or dst is not in the graph.	`g.isConnected("e","b")`
void printNodes() const;	Prints the data stored in all the nodes in the graph, with one node per line, starting from the node with the smallest outdgree to the node with the largest. If two nodes have the same edge count, then the one with the smaller node value determined by the < operator is printed first. See test11.cpp for an example.	`g.printNodes();`
void printEdges(const N& val) const;	Prints the outgoing edges of a given node, SrcNode, containing value val. The first line must be "Edges attached to Node X", where X is the data stored at SrcNode. Then, the outgoing edges of SrcNode are printed in increasing order of their weights, with one edge per line adhering to the following format: `[the data at the destination node DstNode] [the cost of the edge]` For example, if SrcNode → DstNode has a weight 3 (of type int), where the data at DstNode is "abc" (of type std::string), then the print out is "abc 3". If SrcNode does not exist in the graph, then a std::runtime_error is thrown with no output printed. If the outdegree of SrcNode is 0, then the first line of the print should work and the second line should be "(null)". If two edges have the same weight, then the edge whose destination node has a smaller node value, again determined by the < operator, is printed first. See test12.cpp for an example.	`g.printEdges(s);`

In addition to these operations, you should also implement the following four operations, which provide a fake iterator for enumerating all the node values in a graph. **You can abstract a graph in any sequence, as long as the sequence consists of all and only the nodes in the graph.**

Member	Description
void begin() const	Sets an internal iterator, i.e., ``pointer'' to the first element of a sequence.
bool end() const	Returns true if the iterator goes past the last element of the sequence and false otherwise.
void next() const	Moves the iterator to the next element of the sequence.
const N& value() const	Returns the value of the node pointed to by the iterator.

You can use the four member functions as follows:

```
gdwg::Graph<std::string,int>> g
for (g.begin(); !g.end(); g.next())
    std::cout << g.value() << std::endl;
```

This `iterator' is not nearly as powerful as a proper iterator (e.g., the one to be implemented in Assignment 4), but it should suffice to give you an idea of what iterators are all about. You must make sure that these member functions can be invoked correctly on both const and non-const graphs.

Hint: These four functions have short implementations. Review the **mutable** qualifier. In the reference solution, there are altogether 4 lines in the bodies of these four functions, with 1 line per function.

Note: Your Graph class should not expose any other public members other than those listed above, although you can add whatever private member functions and classes as you see fit. This implies that your Node and Edge classes should be private nested classes.

4. Getting started:

- Name your class as Graph, with its interface in Graph.h and implementation in Graph.tem.
- Include your implementation inside your interface file as follows:

  ```
  Graph.h:

  namespace gdwg {

     Your class interface

     #include "Graph.tem"

  }
  ```

- Place the class declaration and definition inside the namespace gdwg
- Make sure you include Header Guards in Graph.h
- Your Graph.h and Graph.tem should not include main().
- Your class will be compiled against a series of test files that include main().
- Your code should not read or write any files or print anything to the screen unless called to do so through a test file that contains `#include "Graph.h"`.

5. Sample test files:

Here are some test cases for testing some functionalities required:

- test1.cpp result1.txt - Graph default construction and Node insertion
- test2.cpp result2.txt - Edge insertion and print ordering (more in test11.cpp and test12.cpp)
- test3.cpp result3.txt - Exception and error handling
- test4.cpp result4.txt - Checks data integrity
- test5.cpp result5.txt - Replace Node data
- test6.cpp result6.txt - Merges two nodes
- test7.cpp result7.txt - Deleting data
- test8c.cpp result8c.txt - Copy construction
- test9c.cpp result9c.txt - Copy assignment
- test8m.cpp result8m.txt - Move construction
- test9m.cpp result9m.txt - Move assignment
- test10.cpp result10.txt - constness
- test11.cpp result11.txt - printing nodes
- test12.cpp result12.txt - printing edges
- test13.cpp result13.txt - fake iterators

In C++, a moved-from object is always in a valid but unspecified state. In this assignment, a moved-from graph is assumed to be an empty graph.

6. Tips:

- Your code should be const correct.
- Use C++14 style as much as possible - especially smart pointers.
- The lecture slides and tutorials have many code snippets that you may find helpful.
- Your code should be well documented, clearly describing how each function operates.
- For a value-like class, you should look at Week 4's lecture slides.
- Do not use other libraries (e.g., boost).
- You should use exceptions for error handling, as required, instead of using C-style asserts. In the other parts of this assignment, you are free to use asserts as you see fit.
- The reference solution is around 500 lines including comments.

7. Testing

You need to make sure that your solution compiles on the CSE machines using the command:.

```
g++ -std=c++14 -Wall -Werror -O2 -o testX testX.cpp
```

The dry run for give will be the first test case.

To run your code, type:

```
./testX
```

You should create your own test cases to check all the functionalities of your code against the specifications.

8. Marking

Your submission will be given a mark out of 100 with 70% an automarked performance component for output correctness and 30% a manually marked component for code style and quality.

As this is a third-year course, your code is expected to be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions.

If you write in C or use C types (e.g. union) or #define macros, you will lose performance marks and style marks.

A number of test cases will be used to mark your solution. To pass a test case, your solution must produce exactly the same output as the reference solution. The results from both will be compared by using the linux tool diff.

9. Submission Instructions

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well, then submit using the command:

```
give cs6771 gdwg Graph.h Graph.tem
```

You do not need to submit a makefile or other test files. We will supply a makefile and test cases for each test.

Late submissions will be penalised unless you have legitimate reasons for an extension which is arranged before the due date. Any submission after the due date will attract a reduction of 20% per day to your individual mark. A day is defined as a 24-hour day and includes weekends and holidays. No submissions will be accepted more than three days after the due date.

Plagiarism constitutes serious academic misconduct and will not be tolerated.

Further details about lateness and plagiarism can be found in the Course Introduction.