

## Project #08 (v1.4)

**Assignment:**     **Graph class**  
**Submission:**    **via Gradescope (limited to 8 submissions)**  
**Policy:**           **individual work only, late work is accepted**  
**Complete By:**    **Friday March 10 @ 11:59pm CST for full credit**

Late submissions: 02% penalty for each 12-hour period past the due date (max penalty 08%)  
NO submissions accepted 48 hours after the due date

**Pre-requisites:**     **none**

### Overview

Congratulations, last project! Take a moment to congratulate yourself, you have worked hard and implemented some non-trivial C and C++ programs. Well done.

A fitting end is to implement a container class in C++, and write some unit tests to validate correctness. Here in Project 08 you're going to implement the Graph class we used in the previous project, with some performance restrictions, then test it using googletest. You are free to use any of the built-in containers in standard C++ 17, or you can use C-style programming and build your own linked data structures.

### Getting Started

We will continue working on the EECS computers. The first step is to setup your project08 folder and copy the necessary files:

1. If you are working on a Mac or Linux, open a terminal. If you are working on Windows, open Git Bash.
2. Login to moore: `ssh moore` or `ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`
3. Make a directory for project 08                    `mkdir project08`
4. Make this directory private                        `chmod 700 project08`
5. Move ("change") into this directory            `cd project08`
6. Copy the files needed for project 08:          `cp -r /home/cs211/w2023/project08/release .`
7. List the contents of the directory               `ls`
8. Move ("change") into release dir               `cd release`

9. List the contents of the directory

**ls**

10. Build the program

**make build**


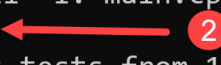
AT THIS POINT you should have the following files / folders in your release directory:

graph.h  
gtest

gtest.o  
main.cpp

makefile

We will be using googletest again for writing our unit tests; recall we used googletest in project 04. The folder “gtest” contains the .h files for googletest; the file “gtest.o” is the googletest library that implements the googletest framework. The makefile compiles main.cpp with gtest.o. The resulting program contains two “dummy” tests, one passes and one fails. Here’s the output when you run ./a.out:

```
hummel@moore$ make 
rm -f ./a.out
g++ -std=c++17 -g -Wall -I. main.cpp gtest.o -lpthread
hummel@moore$ ./a.out 
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from testing
[ RUN      ] testing.test01
main.cpp:8: Failure
Value of: 2 == 3
  Actual: false
Expected: true
[  FAILED  ] testing.test01 (0 ms)
[ RUN      ] testing.test02
[      OK  ] testing.test02 (0 ms)
[-----] 2 tests from testing (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] testing.test01

1 FAILED TEST
```

## Part one: Graph class

The project has two parts. Part 01 is to implement a Graph class defined by the given “graph.h”, which has string-based vertices and edge weights of type double. The public methods must remain exactly the same in all aspects --- name, return type, parameters, const, and behavior. All data members must remain private; you can add public methods if you want (e.g. to help you with testing). Create a “graph.cpp” file and provide your graph implementation in that file; do not implement the class in “graph.h”. Be sure to update your makefile and add “graph.cpp” to the list of files that need to be compiled (line 3 of the makefile).

In class we implemented the Graph class using an adjacency matrix. Here in project 08, your Graph class must meet the following requirements:

1. *Space requirement of at most  $O(V + E)$ , where  $V$  is the # of vertices and  $E$  is the # of edges. Note that this means “on the order of”, so using space  $2V + E$  is still  $O(V + E)$  since the constant is dropped.*
2. *Adding a vertex may take at most  $O(\lg V)$  time.*
3. *Adding an edge  $(x, y, w)$  may take at most  $O(\lg V + e)$  time, where  $e$  is the # of edges starting from  $x$ .*
4. *Getting an edge’s weight may take at most  $O(\lg V + e)$  time.*
5. *Must support all possible weights (negative, 0, or positive).*
6. *Must support cyclic edges, i.e. an edge from a vertex to itself.*
7. *Does not need to support multi-edges --- i.e. assume at most one edge from vertex  $x$  to vertex  $y$ . [ To be clear, vertex  $x$  can have an edge to itself, an edge to vertex  $y$ , an edge to vertex  $z$ , and so on. There just can’t be multiple edges from  $x$  to  $y$ , at most only one from  $x$  to  $y$ . ]*

This implies you cannot use an adjacency matrix, since it has a space requirement of  $O(V^2)$ . This also implies you cannot use a single linked-list or vector, since these would fail the time requirements. And linear search is not an option when searching for a vertex; this will fail the time requirements. Your implementation needs to be a bit more sophisticated. Here is the provided “graph.h” file that you must implement:

```
/*graph.h*/

//
// Graph class implemented as ...
//
//      ?????????
//
// For this implementation of graph, the vertices are
// string, and the weights are doubles. Example:
//
//      graph G;
//      G.addVertex("Chicago");
//      G.addVertex("New York");
//      G.addEdge("Chicago", "New York", -999.99);
//
// ???YOUR NAME???
//
// Northwestern University
// CS 211: Winter 2023
//

#pragma once
```

```

#include <iostream>

using namespace std;

class graph
{
private:
    //
    // TODO:
    //

public:
    //
    // default constructor: no parameters
    //
    graph();

    //
    // NumVertices
    //
    // Returns the # of vertices currently in the graph.
    //
    int NumVertices() const;

    //
    // NumEdges
    //
    // Returns the # of edges currently in the graph.
    //
    int NumEdges() const;

    //
    // addVertex
    //
    // Adds the vertex v to the graph and returns true. However,
    // if v is already in the graph, it is not added again and
    // false is returned.
    //
    bool addVertex(string v);

    //
    // addEdge
    //
    // Adds the edge (from, to, weight) to the graph, and returns
    // true. If the vertices do not exist then no edge is added
    // and false is returned.
    //
    // NOTE: if the edge already exists, the existing edge weight
    // is overwritten with the new edge weight.
    //
    bool addEdge(string from, string to, double weight);

    //
    // getWeight
    //
    // Returns the weight associated with a given edge. If
    // the edge exists, the weight is returned via the reference
    // parameter and true is returned. If the edge does not

```

```

// exist, the weight parameter is unchanged and false is
// returned.
//
bool getWeight(string from, string to, double& weight) const;

//
// neighbors
//
// Returns a set containing the neighbors of v, i.e. all
// vertices that can be reached from v along one edge. If
// v does not exist, an empty set is returned.
//
set<string> neighbors(string v) const;

//
// getVertices
//
// Returns a vector containing all the vertices currently in
// the graph. There is no guaranteed order to the vertices
// in the returned vector; the order should be considered
// random.
//
vector<string> getVertices() const;

//
// print
//
// Prints the internal state of the graph to the given
// output stream for debugging purposes. The output will
// contain the # of vertices, the # of edges, the vertices,
// and the edges. Vertices are output as strings and the
// weights as doubles.
//
// Example:
//   graph G;
//   ...
//   G.print(cout); // print to console
//
void print(ostream& output) const;
};

```

The format of the output produced by your `print()` method is up to you, as long as it provides the required information: # of vertices, # of edges, the vertices (output in string-form), and the edges (with vertices output in string-form). Note that a copy constructor and destructor is only needed if you are dynamically allocating memory in your constructor. If you dynamically-allocate memory, you'll need to implement these methods in order to fulfill the requirements of C++ when a copy is needed and the destructor is called.

Before you start on part 01, read part 02 and work on both parts simultaneously. In other words, write a Graph method in "graph.cpp", then add 1 or more unit tests in "main.cpp" to test it. Build, run, test. Then implement another Graph method, and write more unit tests. Build, run, test. Repeat until done. Building and testing incrementally is proper software engineering.

## Part 02: Unit Testing

Part 02 of the project is to write at least 5, non-trivial unit tests that test the public methods of the Graph class. Be sure to test all aspects of each method. For example, be sure to test:

1. `addVertex(v)` returns false if `v` is already in the graph.
2. `addEdge(x, y, w)` returns false if `x` or `y` do not exist.
3. `addEdge(x, y, w)` updates the existing weight if the edge already exists.
4. `getWeight(x, y, w)` returns false if a vertex does not exist, or the edge does not exist.

You do not need to test the `print()` method, the staff will manually review to make sure it outputs the required information.

What is a “non-trivial” unit test? Each unit test must create a graph with at least 8 vertices and 12 edges; create the graph using explicit calls to `addVertex()` and `addEdge()`, do not build by inputting from a file because Gradescope will not be configured to support this. Create different graphs, do not create the same graph over and over again. Each test case does not need to call every function, instead think of each test case as testing one aspect of the graph:

1. simple test building a basic graph
2. test functions returning false
3. test `getVertices()` on an empty graph, `neighbors(v)` on an empty graph / where `v` does not exist / where `v` has no neighbors
4. test all the functions in various combinations
5. stress test a BIG graph (millions of vertices and edges) to make sure you are meeting time requirements --- you should be able to build a big graph in a few seconds

One of your required tests must be a stress test as described above. We encourage you to add more tests if you want; you can never have too many unit tests.

To encourage good testing, Gradescope submissions will be limited here in project 08. You will be allowed at most 8 submissions, with the output of some tests cases hidden (all you receive is pass/fail). Note that you cannot use a peer’s submissions, that will be considered academic misconduct if you submit via their account.

If you want to check the code coverage statistics of your unit tests, note that **gcov** is available on the EECS computers. Recall that we discussed code coverage in class on Tuesday January 31<sup>st</sup>; lecture notes [PDF](#). In short, you’ll need to

1. update makefile to compile with code coverage options (see slide 17 of PDF)
2. build and run program as usual
3. `gcov graph.cpp`
4. open “`graph.cpp.gcov`” in editor and review the coverage information

## Optional Part 03: a templated graph class

You can earn “extra credit” by completing this completely optional, part 03 exercise. Note that you must successfully complete parts 01 and 02 (via 80/80 gradescope submission) before considering part 03. This is especially true because you have to copy and modify your solution to parts 01 and 02 in order to start on part 03.

By “extra credit”, note that this can only be used to bump you up if you’re at a breakpoint in the final grading. If you have an A-, this will earn you an A. If you have a B+, this will earn you an A-, and so on.

A class that is templated must be defined completely in the .h file, so make a copy of all your files before you start on part 03:

- |   |                                     |
|---|-------------------------------------|
| 1. move up and out of your release directory    | <code>cd ..</code>                  |
| 2. list the contents --- you should see release | <code>ls</code>                     |
| 3. make a copy of this directory                | <code>cp -r release optional</code> |
| 4. change into your new directory               | <code>cd optional</code>            |

Copy-paste the contents of your “graph.cpp” file into your “graph.h” file. Move each method inside the { } of the graph class, and remove the **graph::** prefix from each method. Modify your makefile and remove “graph.cpp” from the list of files that are compiled (i.e. remove from line 3 of your makefile). Build and run your program --- the test cases should all pass exactly as before. Your “graph.h” file is now a complete implementation of your graph class.

Next, add the template prefix to the class declaration in “graph.h”:

```
template<typename VertexT, typename WeightT>
class graph
{
```

Now re-program the class --- data members and methods --- in terms of **VertexT** and **WeightT** instead of string and double.

Change your unit tests to declare your graph objects like this: `graph<string, double> G`. Now build and run your unit tests, and they should all pass as before. [ *Don’t you love unit testing? Just run and see what happens, no other typing and no output to review!* ]

Finally, just to make sure all is well, add a couple more unit tests that try different type combinations to make sure the templated code is written properly. Try `graph<long long, int>?` Try `graph<string, string>?` Try 1-2 more, maybe `graph<string, vector<int>>?`

A separate Gradescope submission will be provided for this optional component; watch Piazza for details.

## Grading and Electronic Submission (parts 01 and 02)

You will submit all your .cpp and .h files to Gradescope for evaluation. To submit from the EECS computers,

run the following command (you must run this command from inside your **project08/release** directory):

```
/home/cs211/w2023/tools/project08 submit main.cpp graph.cpp graph.h
```

The “Project08” submission will open a few days before the due date. Once available, you will be allowed at most 8 submissions; Gradescope will reject all submissions after your 8<sup>th</sup> one.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your “Submission History”. This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements and reviewing your overall approach. However, you may lose correctness points if you have a feature correctly implemented, but manual review reveals that the proper steps were not taken (“no nested loops in main( )”):

1. *Graph methods meet space and time requirements; your score could drop to 0 if you do not meet these requirements. [ For example: if you give us an implementation based on adjacency matrix, your overall score will be 0 / 100. ]*
2. *At least 5 non-trivial test cases as described; -20 points for each test case that is not valid.*
3. *At least one of the test cases must represent a stress test of your graph class; -20 if not.*
4. *Update the header comment at the top of “graph.h” to describe how you implemented your graph; include your name as well. Copy-paste this header comment to the top of “graph.cpp”.*
5. *Add a header comment to “main.cpp” with summary and your name.*
6. *A brief header comment at the top of each test case summarizing what is being tested.*
7. *No memory errors AND no memory leaks. You can run valgrind on the EECS computers as we did on replit: **valgrind ./a.out***

## Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found [here](#). In summary, here are NU’s eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do your own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU’s academic integrity [website](#). With regards to CS 217, unless stated otherwise, all work submitted for grading *\*must\** be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for



grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- in how to solve the problem. Talking to other students about the problem, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL.