CS 271 Project 7 (Spring 2025) – Graphs: Everything Everywhere All at Once

Instructor: Prof. Michael Chavrimootoo
Due: Wednesday, April 23, 2025, 11:59PM

1 Project Overview

In this project, you will be implementing a graph class along with some standard graph algorithms. You will become more familiar with adjacency list representations and the details of BFS, DFS, and topological sorting.

1.1 Learning Goals

By the end of this project you will have a better understanding of directed graphs, adjacency lists, BFS, DFS, and topological sorting. You will also learn about using `unions` for efficient and practical data storage.

1.2 Use of Other Work

While you may consult any code you have personally written for this class, and any code in the course textbook, you are not allowed to use any other sources beyond the ones listed in the next paragraph, which includes and is not limited to, your peers outside your group, other textbooks, the internet, and AI tools (such as Gemini and ChatGPT).

External sources you are allowed to use for this project, as with all projects, are `https://stackoverflow.com`, `https://en.cppreference.com`, and `https://cplusplus.com`. In particular, you may find it useful to familiarize yourself with

1. `union`: `https://en.cppreference.com/w/cpp/language/union` (see more later in this document).

Note: The above list is not exhaustive, and if you use any additional external source, *you must cite it in your report and make a clear reference in your code of where it is used.*

Learning to read documentation, e.g., C++'s, is a *crucial* skill for any CS major, and if you're not comfortable with it, now is a great time to develop that skill.

1.3 Group Work

You can complete this project individually or in pairs. I will leave the pairing if you wish to do it that way. However, if you work in pairs, you must detail your individual contributions in your report (see Section 2.5). I expect everyone to contribute meaningfully to **both** the coding and non-coding parts of the project.

2 Submissions

2.1 Gradescope

Projects will be submitted via Gradescope, and you will be graded on passing certain tests. Each test contains multiple assertions, and you will need to carefully design your tests, which you will also need to provide. Every time you make a new submission, Gradescope will run the autograder to check that you pass all the tests, and will give you your results.

2.2 Compatibility

The tests are compiled using GCC and C++17. Different compilers/platforms may exhibit different behaviors, so I recommend testing your code using GCC, e.g., on the Olin machines, before submitting as those are closest to what Gradescope will test your code on. Note that it is possible for your code to compile and run on the Olin machines and still fail on Gradescope; that is due to the compiler configurations there that may allow subtle differences. If your project fails to compile/run, you will receive a zero.

2.3 Code Submission

You must submit these files:

1. `Graph.tpp` - which will contain the implementations for the header file named `Graph.hpp`.
2. `studentTests.cpp` - which will contain your own tests. *You will be graded on the quality of your own tests.* Here is how I will be compiling your tests, all your files are in the same directory
   ```
   g++ -std=c++17 studentTests.cpp -o studentTests
   ./studentTests
   ```

If you're missing either of these files, the autograder will halt, and give you a zero.

Gradescope allows you the ability to upload the contents of a GitHub repository, so feel free to work on GitHub and upload from there. The autograder will ignore your other files.

Tip 1 *Learning to use* `git` *correctly for both collaborative and personal projects is a tremendous skill to have; you will certainly use it throughout your careers. Remember to make meaningful commit messages throughout the development of the project.*

Moreover, you must document your code properly. This includes:

- A full comment header at the top of each source file containing the file name, the developer names, the creation date, and also a brief description of the file's contents.
- Each method should have a full comment block including a description of what the method does, a "Parameters" section that lists explicit pre-conditions on the inputs to the function, and a "Return" section that explicitly describes the return value and/or any side effects.

- Make careful use of inline comments to explain various parts of the code that may not be obvious from the code syntax.

You will be graded on the quality of your code.

2.4 Technical Considerations

This section covers some technical considerations new to this project.

2.4.1 Reading from `STDIN`

The `readFromSTDIN()` function is very useful as it allows you to create a graph from a file in the following way. Suppose your file `studentTests.cpp` looks as follows

```
#include <iostream>
#include "Graph.hpp"

int main() {
    Graph g = Graph::readFromSTDIN(); // creates the graph based on what is passed in STDIN
}
```

and is compiled into a binary called `studentTests`. If your graph's representation (in the format below) is store in file `myGraph.txt`, you can do

`./studentTests < myGraph.txt`

to pass the contents of `myGraph.txt` to STDIN for your `studentTests` binary to read.

The expected format is the following

```
n m
u_1 v_2
...
u_m v_m
```

The first line contains two numbers n and m, where n is the number of vertices and m is the number of edges. Then each of the following m lines is a pair u_i and v_i representing edge (u_i, v_i) to be added to the graph.

2.4.2 The `union` Type

A union type can only one of several types of data. For example, given the following union type

```
union S {
    int a;
    double b;
    std::string x;
};
```

I can then declare an element of type `S`, and have access to one of those field, and only one of those fields. The linked C++ documentation article provides a few examples on how it works. Why is a union useful? Well, the power comes from their use with anonymous unions and anonymous structs. Consider the type `TraversalData` in `Graph.hpp`. In your code, a variable of type `TraversalData` will have its third member (which is anonymous, i.e., has not been given a name) be of the type defined by one of the two structs within that union. This is useful, because in your BFS code, you can write things like

```
std::vector<TraversalData> data;
// ... data is filled using BFS
// let u be an int representing a vertex
std::cout << data[u].distance << std::endl; // prints the distance of u from s after running BFS
```

2.4.3 On Ordering

Your DFS implementation must compute the topological ordering of the graph it is being run on based on the algorithm we saw in class and in the reading. Of course, if the graph has a cycle, the result will be meaningless, but, the algorithm will compute an attempt at an ordering, and your code must do the same. In particular, for vertices $0, \ldots, n-1$, the first vertex in the ordering must be assigned 1, the second 2, and so on. The easiest way to do this, is to set a variable `order` to `n` at the start of your DFS implementation, and pass that variable by reference to the `dfsVisit` function. Then after a node is done (i.e., you assign a value to `.finish` for that node), set the `.order` value to the current value of `order`, and then decrement `order`. This will label the nodes with their ordering. Given such an ordering, it is easy to actually order the nodes in linear time, as a byproduct of running DFS.

2.5 Report Submission

Along with your code, you must submit a report on Gradescope as a single PDF file. The report is an essential component of this project, and it is yet another way in which you communicate your technical contributions as a group. An individual who has never seen your project's description should be able to understand what the project is about and how your solution works solely by reading the report and your comments.

Your report should be organized in the following (brief) sections:

- Project Contributors, and their individual contributions (if working in pairs).
- Group Challenges (if applicable) - Discuss any major problem your group encountered while implementing the project. This can also include challenges you encountered with your group.
- Individual Reflection - Describe your individual contributions to the project and briefly describe any challenges you (not your group) faced in this project.
- Known Issues - If there are any known issues with your code, mention those here. For each issue, try to best explain what may be causing the issue and what would be a potential solution; by thinking through the cause of the issue, you may find a way to fix it :)

- Additional Information - Any additional information you find relevant; keep it brief.
- Further Considerations - see Section 2.6.

You may use your notes from class or the textbook to guide you through the implementations, and you cannot use any other resources.

While you cannot collaborate to *write* your reports, you are certainly free to discuss elements of the reports with your group, the TA, and me. However, you cannot produce any written material or recordings during those discussions.

Be careful: When writing your reports, you may not use text from the project description or other sources verbatim. You can only use your own words. If there is evidence that you received unauthorized help in your reports, you will receive a zero for that part of the assignment and a formal academic integrity violation report will be filed.

2.6 Further Considerations

In your report, briefly respond to the following questions:

1. What was the hardest part of doing this project?
2. What was the easiest part of doing this project?
3. What is something that you learned about graphs in this project?
4. Did you learn anything new about C++ in this project?
5. How would you modify your implementation to work for weighted graphs? Would you modify your BFS and DFS implementations? DO NOT PROVIDE CODE.