ECS 34: Programming Assignment #3

Instructor: Aaron Kaloti

Spring 2021

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, 05/05. Gradescope will say 12:30 AM on Thursday, 05/06, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Grading

This assignment is worth 9% of your final grade. 8% of this will be from the autograder. 1% of this will be reserved for manual review (see below).

3.1 Manual Review

I will also manually review your C++ code and provide comments/tips regarding implementation quality and code style that you will be able to view on Gradescope. You will be graded on the style and quality of your code. You should consult the code style/quality guide that is on Canvas here.

4 Submitting on Gradescope

Only submit `graph.hpp` and `graph.cpp`. You may be penalized for submitting additional files. You have infinite submissions until the deadline.

Once the deadline occurs, whatever score the autograder has for your active submission (your last submission, unless you manually change it) is your *final* score (unless you are penalized for violating any restrictions mentioned in this document).

4.1 Regarding Autograder

The reference environment is the CSIF.

As always, your output must match mine *exactly*.

As was the case in my ECS 32C course, there are hidden test cases whose results will not be shown to you until after the deadline.

Once the autograder is released, details about the visible test cases will be added here.

5 Prerequisite C++ Concepts

There are several minor concepts involved (or that may be involved, depending on your implementation) in this assignment that we have not talked about in this course. We will talk about these concepts during the 04/26 lecture.

6 Reminder about the CSIF

I don't know how to stress this enough, but the **CSIF is the reference environment**. That is, if your code compiles and behaves properly on a CSIF computer, then it should also compile and behave properly on the Gradescope autograder, since both environments use a recent Ubuntu operating system[1]. It is recommended that you use the CSIF as much as possible or – at least – every now and then, especially if you see that your code is failing to compile or misbehaving on the Gradescope autograder despite working find on your computer.

7 Graph Class

7.1 File Organization

`class Graph` is defined in `graph.hpp`, and its methods are *declared* in this file too. These methods are to be *defined* in `graph.cpp`. I provide a version of `graph.hpp` that you ***must*** start with. You *are* allowed to modify this file – e.g. to add member variables or (ideally) `private` helper methods – but you must not modify the provided method declarations, because the autograder code will call those methods. For example, you should not change the return type of `getBFSOrdering()` or the number of parameters taken by `getAdjacencyMatrix()`.

You should read through the comments and method declarations in `graph.hpp`. That will tell you much about how your code is expected to behave. The rest of this document contains details that did not seem appropriate for `graph.hpp`.

7.2 Graph File Formats

As you can see in `graph.hpp`, the first `Graph` constructor takes as argument the name of a file whose contents follow a certain format. ***Your code may assume that the graph file is properly formatted,*** although – as mentioned in `graph.hpp` – the graph information itself may be invalid / lead to the throwing of an exception.

As mentioned later in this section, there are three types of formats that the file could use. Regardless of the format, the following is true:

[1] There is one exception I know of that occurs if you have uninitialized variables, as such variables may be "initialized" differently on the CSIF vs. on the Gradescope autograder. It is bad form to have uninitialized variables in general.

- The first line of the file will contain either the word "Unweighted" or the word "Weighted", indicating whether the graph is weighted or not.
- The second line of the file will contain either the word "Undirected" or the word "Directed", indicating whether the graph is directed or not.
- The third line of the file will contain one of the following three words, indicating which of the three formats the graph vertex/edge data is given in:
  1. "AdjMatrix": adjacency matrix format.
  2. "AdjList": adjacency list format.
  3. "ListEdges": list of edges format.
- The fourth line of the file will contain the number of vertices.

After the fourth line, the graph vertex/edge data is given, and the format of this part depends on which of the three formats is specified in the third line of the file.

7.2.1 Adjacency Matrix Format

If the file is using this format, then after the fourth line, the graph contains an adjacency matrix of weights, where a weight of 0 denotes a nonexistent edge. Below is an example.

```
$ cat demo_graph_files/adj_matrix1.txt
Weighted
Directed
AdjMatrix
5
0 3 5 0 1
2 0 -3 4 0
10 5 0 -8 17
4 7 1 0 3
5 0 100 0 0
```

In the above example, the fourth line states that the number of vertices is 5, and sure enough, the adjacency matrix that follows is 5-by-5. In the first row of the matrix, we see information about the edges leaving vertex #0. For example, the second entry of this row (the 3) indicates that there is an edge from vertex #0 to vertex #1 that has a weight of 3. In the following row, the first entry (the 2) indicates that there is an edge from vertex #1 to vertex #0 that has a weight of 2.

In the case of an unweighted graph, the values are T or F, where F denotes a nonexistent edge.

In the case of an undirected graph, half of the matrix is redundant, so only half of the matrix is provided. Below is an example. Note that the leading spaces in all but the first row of the adjacency matrix should not trouble your implementation if you use the tools that C++ provides to you for reading from a file. (That is, your implementation should not need to care about the leading spaces, which are there for the convenience of us humans.)

```
$ cat demo_graph_files/adj_matrix2.txt
Unweighted
Undirected
AdjMatrix
4
F T F F
  F T T
    F T
      F
```

7.2.2 Adjacency List Format

This format should be more straightforward to explain. Below is an example.

```
$ cat demo_graph_files/adj_list1.txt
Unweighted
Directed
AdjList
4
1 3
2

0 2
```

In the above example, after the fourth line (which reports that the number of vertices is 4), the neighbors of each vertex are given. For example, thanks to the row containing "1 3", we know that there are edges from vertex #0 to vertices #1 and #3, and thanks to the empty row two lines later, we know that there are no edges leaving vertex #2.

In the case of a weighted graph, after each neighbor, the weight of the edge to that neighbor is given. Below is an example. Consider the first adjacency list row, which is "1 -5 3 2". This row is saying that there are is an edge from vertex #0 to vertex #1 with weight -5 and an edge from vertex #0 to vertex #3 with weight 2. Now, consider the row afterwards, which is "2 10". This row is saying that there is an edge from vertex #1 to vertex #2 with weight 10. The row after that, which is empty, is saying that there are no edges leaving vertex #2. The last row corresponds to the fourth vertex and is saying that there is an edge from vertex #3 to vertex #0 with weight 8 and an edge from vertex #3 to vertex #2 with weight 4.

```
$ cat demo_graph_files/adj_list2.txt
Weighted
Directed
AdjList
4
1 -5 3 2
2 10

0 8 2 4
```

You may assume that if the graph file is using this format, the graph will be directed[2].

7.2.3 List of Edges Format

With this straightforward format, the file specifies the start and end of each edge. For instance, in the file below, the line "5 3" indicates an edge from vertex #5 to vertex #3, and the line "0 4" indicates an edge from vertex #0 to vertex #4.

```
$ cat demo_graph_files/edge_list1.txt
Unweighted
Directed
ListEdges
6
5 3
0 4
2 5
3 5
2 0
```

In the example below, the graph is weighted, so the weight of each edge is given as the third value in that edge's line. For example, the line "1 2 -5" indicates an edge from vertex #1 to vertex #2 that has a weight of -5. Note that since the graph is undirected, this line *also* indicates an edge from vertex #2 to vertex #1.

```
$ cat demo_graph_files/edge_list2.txt
Weighted
Undirected
ListEdges
4
1 2 -5
0 2 -3
1 3 20
```

7.3 Exceptions

Certain constructors and methods throw exceptions under certain circumstances. In such cases, the exact exception message *will not be checked by the autograder*; all that will be checked is that your code throws an `std::logic_error` exception in such situations.

7.4 Demo

On Canvas, you will find a file called `demo_graph.cpp`. Below are its contents.

```cpp
/**
 * Code (released to the students) for demonstrating the use of the Graph
 * class.
 * Uses a mix of assert()-based unit testing and printing out values.
 */

#include "graph.hpp"

#include <cassert>
#include <cstdlib>
```

[2] I am permitting this assumption because I could not think of an undirected graph adjacency list format that wasn't awkward.

```cpp
#include <iostream>

#define INPUT_FILE_DIR "demo_graph_files"

static void printAdjMatrix(
    const std::vector<std::vector<int>>& adjMatrix,
    bool weighted)
{
    for (unsigned i = 0; i < adjMatrix.size(); ++i)
    {
        std::cout << i << ": ";
        for (int weight : adjMatrix[i])
        {
            if (weighted)
            {
                // Unfortunately and strangely, if I use the ternary operator
                // here, the ASCII value of '.' gets printed out. I guess
                // the C++ compiler assigns a specific type to the output of
                // the ternary operator.
                if (weight == 0) std::cout << '.';
                else std::cout << weight;
                std::cout << ' ';
            }
            else std::cout << (weight == 0 ? '.' : 'X') << ' ';
        }
        std::cout << std::endl;
    }
}

static void printAdjList(
    const std::vector<std::vector<std::pair<unsigned, int>>>& adjList,
    bool weighted)
{
    for (unsigned i = 0; i < adjList.size(); ++i)
    {
        std::cout << i << ": ";
        for (std::pair<unsigned, int> neighborEntry : adjList[i])
        {
            std::cout << neighborEntry.first;
            if (weighted) std::cout << " (" << neighborEntry.second << ')';
            std::cout << ' ';
        }
        std::cout << std::endl;
    }
}

static void demo1()
{
    std::vector<std::pair<unsigned, unsigned>> edges;
    // More on emplace_back(): https://www.cplusplus.com/reference/vector/vector/emplace_back/
    edges.emplace_back(3, 2);
    edges.emplace_back(0, 1);
    edges.emplace_back(0, 2);
    // Invoke the second of the three constructors.
    Graph g1{4, edges, true};
    assert(!g1.isWeighted());
    assert(g1.isDirected());
    assert(g1.getNumVertices() == 4);
    assert(g1.getNumEdges() == 3);
    std::cout << "=== g1's adjacency matrix ===\n";
    printAdjMatrix(g1.getAdjacencyMatrix(), g1.isWeighted());
    std::cout << "=== g1's adjacency list ===\n";
    printAdjList(g1.getAdjacencyList(), g1.isWeighted());

    // Invoke the first of the three constructors.
    Graph g2{INPUT_FILE_DIR"/adj_list2.txt"};
    assert(g2.isWeighted());
    assert(g2.isDirected());
    assert(g2.getNumVertices() == 4);
    assert(g2.getNumEdges() == 5);
    std::cout << "=== g2's adjacency matrix ===\n";
    printAdjMatrix(g2.getAdjacencyMatrix(), g2.isWeighted());
    std::cout << "=== g2's adjacency list ===\n";
    printAdjList(g2.getAdjacencyList(), g2.isWeighted());
```

```

    // Again, invoke the first of the three constructors, this time with
    // an undirected graph.
    Graph g3{INPUT_FILE_DIR"/edge_list2.txt"};
    assert(g3.isWeighted());
    assert(!g3.isDirected());
    assert(g3.getNumVertices() == 4);
    assert(g3.getNumEdges() == 3);
    std::cout << "=== g3's adjacency matrix ===\n";
    printAdjMatrix(g3.getAdjacencyMatrix(), g3.isWeighted());
    std::cout << "=== g3's adjacency list ===\n";
    printAdjList(g3.getAdjacencyList(), g3.isWeighted());

    // Try a nonexistent file.
    Graph g4{"nonexistent_file"};

    // Not reached, because an exception is thrown above.
    std::cerr << "End of " << __FUNCTION__ << "() reached.\n";
}

static void demo2()
{
    // Try a graph in which invalid vertex IDs are used.
    Graph g5{INPUT_FILE_DIR"/edge_list_bad1.txt"};

    // Not reached, because an exception is thrown above.
    std::cerr << "End of " << __FUNCTION__ << "() reached.\n";
}

static void demo3()
{
    // Try a graph in which there is a self loop.
    Graph g6{INPUT_FILE_DIR"/adj_list_bad1.txt"};

    // Not reached, because an exception is thrown above.
    std::cerr << "End of " << __FUNCTION__ << "() reached.\n";
}

static void demo4()
{
    // Demonstrate the remaining graph methods.

    Graph g7{INPUT_FILE_DIR"/edge_list3.txt"};
    assert(!g7.isWeighted());
    assert(g7.isDirected());
    assert(g7.getNumVertices() == 4);
    assert(g7.getNumEdges() == 5);
    std::cout << "One BFS ordering (many are possible), starting at 0:\n";
    std::vector<unsigned> ordering = g7.getBFSOrdering(0);
    for (unsigned v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Another BFS ordering, starting at 2:\n";
    ordering = g7.getBFSOrdering(2);
    for (unsigned v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "One DFS ordering, starting at 3:\n";
    ordering = g7.getDFSOrdering(3);
    for (int v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Transitive closure:\n";
    auto tc = g7.getTransitiveClosure();
    for (auto row : tc)
    {
        for (bool val : row)
            std::cout << val << ' ';
        std::cout << '\n';
    }

    Graph g8{INPUT_FILE_DIR"/edge_list4.txt"};
    assert(!g8.isWeighted());
    assert(g8.isDirected());
```

```
    assert(g8.getNumVertices() == 6);
    assert(g8.getNumEdges() == 6);
    std::cout << "One BFS ordering (many are possible), starting at 0:\n";
    ordering = g8.getBFSOrdering(0);
    for (unsigned v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Another BFS ordering, starting at 2:\n";
    ordering = g8.getBFSOrdering(2);
    for (unsigned v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "One DFS ordering, starting at 3:\n";
    ordering = g8.getDFSOrdering(3);
    for (int v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Transitive closure:\n";
    tc = g8.getTransitiveClosure();
    for (auto row : tc)
    {
        for (bool val : row)
            std::cout << val << ' ';
        std::cout << '\n';
    }
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        std::cerr << "Wrong number of command-line arguments.\n";
        return 1;
    }
    int caseNum = atoi(argv[1]);
    switch (caseNum)
    {
        case 1: demo1(); break;
        case 2: demo2(); break;
        case 3: demo3(); break;
        case 4: demo4(); break;
        default:
            std::cerr << "Invalid case number.\n";
            return 2;
    }
    return 0;
}
```

You can compile this program with the line `g++ -Wall -Werror -std=c++14 demo_graph.cpp graph.cpp -o demo_graph`. Which command-line argument you pass to this program determines which demonstrate is done.

Below is the output of the *first* demonstration. As mentioned above, the exact exception error message will not be checked.

```
$ ./demo_graph 1
=== g1's adjacency matrix ===
0: . X X .
1: . . . .
2: . . . .
3: . . X .
=== g1's adjacency list ===
0: 1 2
1:
2:
3: 2
=== g2's adjacency matrix ===
0: . -5 . 2
1: . . 10 .
2: . . . .
3: 8 . 4 .
=== g2's adjacency list ===
0: 1 (-5) 3 (2)
1: 2 (10)
2:
3: 0 (8) 2 (4)
```

```
=== g3's adjacency matrix ===
0: . . -3 .
1: . . -5 20
2: -3 -5 . .
3: . 20 . .
=== g3's adjacency list ===
0: 2 (-3)
1: 2 (-5) 3 (20)
2: 1 (-5) 0 (-3)
3: 1 (20)
terminate called after throwing an instance of 'std::logic_error'
  what():  Failed to open file
Aborted (core dumped)
```

Below is the output of the *second* demonstration.

```
$ ./demo_graph 2
terminate called after throwing an instance of 'std::logic_error'
  what():  Invalid graph
Aborted (core dumped)
```

Below is the output of the *third* demonstration.

```
$ ./demo_graph 3
terminate called after throwing an instance of 'std::logic_error'
  what():  Invalid graph
Aborted (core dumped)
```

Below is the output of the *fourth* demonstration.

```
$ ./demo_graph 4
One BFS ordering (many are possible), starting at 0:
0 1 2 3
Another BFS ordering, starting at 2:
2
One DFS ordering, starting at 3:
3 0 1 2
Transitive closure:
1 1 1 1
1 1 1 1
0 0 1 0
1 1 1 1
One BFS ordering (many are possible), starting at 0:
0
Another BFS ordering, starting at 2:
2 1 0 5 3
One DFS ordering, starting at 3:
3 0
Transitive closure:
1 0 0 0 0 0
1 1 1 1 0 1
1 1 1 1 0 1
1 0 0 1 0 0
0 0 0 0 1 0
1 1 1 1 0 1
```