ECS 36C: Programming Assignment #5

Instructor: Aaron Kaloti

Winter Quarter 2022

Contents

1 Changelog

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

- v.1: Initial version.
- v.2:
  - Added a small "Additional Features" section containing optional things you could add on your own time.
  - Added more to the "Grading Breakdown" section.
  - Part #2 directions.
  - In part #1 directions, added note about flow edges with weight zero.
- v.3: Added some clarifications to the part #1 directions.
- v.4: Autograder details.
- v.5: More autograder details.

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 Friday, 03/18. Gradescope will say 12:30 AM on Saturday, 03/19, 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.*

*This content is protected and may not be shared, uploaded, or distributed.

3 Purpose of This Assignment

- To increase your familiarity with concepts related to graphs, especially network flow.
- To, as usual, put your C++ coding skills to the test.

4 Reference Environment

The autograder, which is talked about at the end of this document, will compile and run your code in a Linux environment. That means that you should make sure your code compiles and behaves properly in a sufficiently similar environment. The CSIF, which I talk about in the syllabus, is one such environment. Do not assume that because your code compiles on an *insufficiently* similar environment (e.g. directly on your Mac laptop), it will compile on the Linux environment used by the CSIF.

You should avoid causes of undefined behavior in your code, such as uninitialized variables. If you have things like this in your code, then your code could end up generating the wrong answer when autograded, even if it generated the right answer when you tested it in a sufficiently similar environment.

5 Files

The only file that you will submit is `netflow.cpp`. You may find it helpful to start with the version of this file provided on Canvas. There are some macros in this file that contain the error messages that you must use with the required `std::runtime_error` exceptions.

The header file, `netflow.hpp`, is provided on Canvas. This version of `netflow.hpp` is the version that the autograder will use. If you submit your own `netflow.hpp`, the autograder will ignore it.

6 Part #1: Network Flow

Implement the function `solveNetworkFlow()` in `netflow.cpp`. This function's declaration and a description of its expected behavior are in the version of `netflow.hpp` provided on Canvas.

Update (v.2): It is fine if it is possible for your function to return flow edges whose weight is zero (indicating that no flow is sent through those edges) since such edges do not affect the max flow, so long as these edges correspond to edges in the flow network and are not completely made up edges. Note that for the example below, a max flow will have to send flow through each edge.

Update (v.3): Some clarifications:

- Multi-edges refer, for example, to something like an edge from A to B and another edge going from A to B.
- A self-loop is an edge that goes from a vertex to itself, e.g. an edge from A to A.
- You may assume that the flow network (capacities) will not contain antiparallel edges. Anti parallel edges refer, for example, to something like an edge from A to B and an edge from B to A.
- You may assume that, in the flow network, there will be no vertex that has zero incident edges, i.e. zero edges incoming *and* zero outgoing.
- You may assume that the flow network will not contain a cycle. Note that the requirement that you check if the flow network contains a self-loop is not talking about cycles. Self-loops and cycles are different.

Below is an example of how your `solveNetworkFlow()` function should behave. Note that there may be multiple max flows, so it is not necessary that your function return the same one as mine on a given flow network, and it is not necessary that the order of edges in the return value is the same either.

```
$ cat demo_netflow.cpp
#include "netflow.hpp"

#include <iostream>

int main()
{
    // The example below uses the flow network from slide #61 of
    // the graphs lecture slide deck, although the letters do not
    // correspond perfectly with 0 through 5, e.g. "a" from the slide
    // is not 0 here; it is 3. (This is to show you that the source
    // need not be the first vertex and the sink need not be the
    // last vertex.)
    std::vector<Edge> maxFlow = solveNetworkFlow({
            {3, 5, 5},
            {3, 0, 10},
            {0, 5, 5},
            {5, 4, 10},
            {0, 4, 3},
            {0, 1, 1},
            {4, 1, 20},
            {4, 2, 5},
            {1, 2, 7}
        },
        6);
    for (const Edge& edge : maxFlow)
        std::cout << edge.from << " -> " << edge.to
            << " (" << edge.weight << ")\n";
}
$ g++ -std=c++14 -Wall -Werror demo_netflow.cpp netflow.cpp -o demo_netflow
$ ./demo_netflow
0 -> 5 (3)
0 -> 4 (3)
0 -> 1 (1)
1 -> 2 (7)
3 -> 5 (5)
3 -> 0 (7)
4 -> 1 (6)
4 -> 2 (5)
5 -> 4 (8)
```

7 Part #2: Teaching Assignments

Implement the function `assignCourses()` in `netflow.cpp`. This function's declaration and the associated `Instructor` class definition are in the version of `netflow.hpp` provided on Canvas. The goal of the function is to generate a teaching assignment that *maximizes the number of courses assigned* and does not violate any instructor's preferences.

You do not need to perform any input validation. For example, you may assume that an instructor's preference list will never include a course that is not contained in the second argument. If you have any questions about what you are allowed to assume, you can always ask/email me for confirmation.

Restrictions: A big part of your approach to this part must involve network flow. Specifically, your implementation of `assignCourses()` must create a flow network that is passed to the function you implemented in part #1. The maximum flow that you get back from calling `solveNetworkFlow()` must be used to determine a correct answer. Failure to do this (i.e. implementing an approach that does not involve network flow) may result in you earning an automatic zero for this part.

Hint: You may find it helpful to understand the approach we discussed during lecture to find a maximum matching in a bipartite graph. (When I say "understand" here, I really do mean *understand*, not just be able to copy steps. Understand why our network flow approach worked, why the edges in the flow network had capacities of 1, why the edges from the source went to the vertices they did, etc.)

Below is an example of how your `assignCourses()` function should behave. Note that there may be multiple optimal assignments of courses, so it is not necessary that your function generate the same one as mine, and it is not necessary that the order of courses (when an instructor is assigned multiple courses) is the same either.

```
$ cat demo_courses.cpp
#include "netflow.hpp"

#include <iostream>

#define appendPref(course) \
    instructors.back().addPreference((course))

int main()
{
    /**
     * I use real instructors' names for fun, but do note that the
     * preference lists are not necessarily accurate.
     * I picked them in a way that leads to an interesting result
     * in which one of Butner or Posnett cannot be assigned
     * as many courses as he desires. (That is, if Posnett is
     * assigned ECS 154A, then Butner cannot be assigned two courses.)
     */
    std::vector<Instructor> instructors;
    instructors.emplace_back("Kaloti", 2);
    appendPref("ECS 50");
    appendPref("ECS 36C");
    appendPref("ECS 34");
    instructors.emplace_back("Nitta", 1);
    appendPref("ECS 150");
    appendPref("ECS 34");
    appendPref("ECS 154A");
    instructors.emplace_back("Porquet", 1);
    appendPref("ECS 150");
    appendPref("ECS 36C");
    instructors.emplace_back("Butner", 2);
    appendPref("ECS 36A");
    appendPref("ECS 154A");
    instructors.emplace_back("Posnett", 1);
    appendPref("ECS 154A");
    std::vector<std::string> courses{"ECS 34", "ECS 36A",
        "ECS 36C", "ECS 50", "ECS 150", "ECS 154A"};
    assignCourses(instructors, courses);
    for (const Instructor& instructor : instructors)
    {
        std::cout << "Prof. " << instructor.lastName
                  << " is assigned to teach:";
        for (const std::string& course : instructor.assignedCourses)
            std::cout << ' ' << course;
        std::cout << std::endl;
    }
}
$ g++ -std=c++14 -Wall -Werror demo_courses.cpp netflow.cpp -o demo_courses
$ ./demo_courses
Prof. Kaloti is assigned to teach: ECS 50 ECS 36C
Prof. Nitta is assigned to teach: ECS 34
Prof. Porquet is assigned to teach: ECS 150
Prof. Butner is assigned to teach: ECS 36A ECS 154A
Prof. Posnett is assigned to teach:
```

8 Epilogue: Additional Features

Do not add anything from this section into what you end up submitting to Gradescope.

Below is a non-exhaustive list of related features you could implement *on your own time* (again, not for your submission to Gradescope), in order to make it more complex and, perhaps, more résumé-worthy. Nothing we talked about in lecture makes it obvious how to go about implementing any of these, which makes them more interesting goals. In particular, both of the below may require that you can generate multiple optimal solutions to a network flow problem instance (or, while solving the network flow problem, actively steer towards solutions of a preferred nature), which may be quite challenging.

- In `assignCourses()`, if there are multiple optimal teaching assignments, prefer a teaching assignment that maximizes the number of different instructors assigned, i.e. that minimizes the number of instructors who are not assigned anything to teach. Such an adjustment would make it so that, in the part #2 example above, it is preferable to assign Posnett (instead of Butner) to ECS 154A.
- Treat each instructor's preferences as ranking the courses in the order in which they would prefer to teach them. For instance, in the part #2 example above, ECS 50 comes before ECS 36C in my preference list, indicating that I would prefer to be assigned ECS 50 over ECS 36C (if I could only be assigned one of them). An instructor is happier if he/she is assigned a course he/she has a stronger desire to teach. Modify your `assignCourses()` function so that if there are multiple optimal teaching assignments, the one that maximizes total instructor happiness is preferred. This will require you to quantify instructor happiness.

9 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 8% of your final grade.

- Part #1: 5.5%.
- Part #2: 2.5%.

10 Autograder Details

If you haven't already, you should read what I say in the syllabus about the Gradescope autograder.

Only submit `netflow.cpp`.

Make sure to have both of the required functions defined in `netflow.cpp`, even if you've only finished part #1 so far.

The autograder will grade your submission out of 80 points.

You can find the visible test cases' inputs on Canvas. There is a lot of code in the `test_netflow.cpp` file because it needs to verify that the flow / teaching assignment that you return is valid. If you are not concerned with that portion of the code, then just look at the case-specific function, e.g. `case15()` if you're curious about case #15. For the verification of your teaching assignment, I made a copy of the `Instructor` vector in case you modify aspects of it besides the assigned courses (which you shouldn't do anyways).

Below is a breakdown of the cases:

- Cases #1 through #14: input validation.
- Cases #15 through #19: rest of part #1 (8 points each).
- Cases #20 through #22: part #2 (8 points each).

UCDAVIS
COMPUTER SCIENCE