ECS 34: Programming Assignment #5

Instructor: Aaron Kaloti

Spring 2021

Contents

1 Changelog

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

- v.1: Initial version.
- v.2: Stated explicitly what `assignTutoring()` is supposed to return.

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 Tuesday, 05/25 Gradescope will say 12:30 AM on Wednesday, 05/26, 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

As stated in the *updated* syllabus, this assignment is worth 11% of your final grade. 9.5% of this will be from the autograder. 1.5% 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.

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

4 Submitting on Gradescope

Only submit `netflow.cpp`. The autograder will use the exact version of `netflow.hpp` that you are provided on Canvas. 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.
Once the autograder is released, details about the visible test cases will be added here.
Your output must match mine *exactly*.

5 Network Flow and Bipartite Matching

As stated above, you will submit the file `netflow.cpp`; your modifications to `netflow.hpp` will not be noticed by the autograder. On Canvas, I provide `netflow.hpp` and a version of `netflow.cpp` that you may find helpful to start with.

In this assignment, you will implement two functions. The first function will solve the network flow problem. The second function will solve the bipartite matching problem, i.e. the problem of finding a maximum matching in a bipartite graph. We covered the network flow and the bipartite matching problems during the 04/28 and 05/14 lectures.

5.1 Network Flow

You must implement the `solveNetworkFlow()` function in `netflow.cpp`. This function takes as argument an `AdjList` object that represents the flow network (i.e. the capacities). For your convenience, here are the relevant definitions from `netflow.hpp`.

```
struct NeighborEntry
{
    unsigned neighbor;
    unsigned weight;  // Since application is network flow,
                      // weights are nonnegative.
};

typedef std::vector<std::vector<NeighborEntry>> AdjList;
```

As its name suggests, an `AdjList` object represents an adjacency list, and the vertices are represented as integers from 0 to $n-1$, where n is the number of vertices. Below is a snippet from `demo_netflow.cpp` that shows how the graph on slide #59 of the graphs lecture slide deck would be created as an `AdjList` object. Vertex `a` corresponds to 0, vertex `b` corresponds to 1, etc.

```
int main()
{
    AdjList adjList{6};
    adjList[0].push_back({1, 5});
    adjList[0].push_back({2, 10});
    adjList[2].push_back({1, 5});
    adjList[1].push_back({3, 10});
    adjList[2].push_back({3, 3});
    adjList[2].push_back({4, 1});
    adjList[3].push_back({4, 20});
    adjList[3].push_back({5, 5});
    adjList[4].push_back({5, 7});
    // ... rest of main() ...
}
```

The `solveNetworkFlow()` function must return an `AdjList` object representing a max flow. This `AdjList` object, needless to say, should have the same vertices and edges (but not necessarily the same edge *weights*) as the flow network.

Additionally, the `solveNetworkFlow()` function must perform some rigorous input validation. The following requirements must be checked, in the order shown.

1. Each vertex (represented by an unsigned integer) that is listed as a neighbor must be in the range 0 to $n-1$, where n is the number of vertices.
2. Each vertex's neighbor list must be sorted by neighbor. For example, if the neighbors of vertex 2 were listed as 3, 1, 6, then that would be an unsorted neighbor list. It would need to be 1, 3, then 6.

3. The graph must be connected.
4. The graph must contain exactly one source node, i.e. exactly one node with an in-degree of 0. Note that this node/vertex need not be node 0.
5. The graph must contain exactly one sink node, i.e. exactly one node with an out-degree of 0.

The moment that an unmet condition is detected, a `std::logic_error` exception should be thrown with a message that matches my implementation's. You can find the messages in the version of `netflow.cpp` that is provided on Canvas.

You may find yourself reusing code from your P3 submission in some scenarios. That is fine. Just mention in a comment that you did that, if you do. When finding the augmenting path, if you use BFS or DFS, you will have to modify the algorithm to make it easier to recrease a path from the source to the sink. This can be done by creating a "parent array" to track – for each node v – the node whose processing caused v to be put in the queue or stack in the first place. (In my opinion, it is easier to make this modification for BFS than for DFS.) If you decide to instead use Dijkstra's algorithm to find an augmenting path, you may want to use the naive version of the algorithm. If you don't use the naive version, then you may find `std::priority_queue` helpful.

5.2 Bipartite Matching Problem

In this assignment, the bipartite matching problem is presented as a problem in which you must assign tutors to courses. For simplicity, each course will have *at most* one tutor, and each tutor can be a tutor for *at most* one course. (Update) The function `assignTutoring()` should return an assignment of tutors to courses that maximizes the number of tutors assigned without violating any tutor's preferences or the simplifying constraint mentioned one sentence ago. The function `assignTutoring()` takes two arguments:

1. The preferences of each tutor, represented as a `std::vector<std::vector<std::string>>` object.
2. A list of courses that are offered, represented as a `std::vector<std::string>>` object.

Below is a snippet from `demo_netflow.cpp` that demonstrates how these arguments may be set up. The below is saying that tutor #0 is willing to tutor for ECS 20 and ECS 36A, tutor #1 is willing to tutor for ECS 20, ECS 36B, and ECS 36C, etc.

```
std::vector<std::vector<std::string>> prefs(5);
prefs.at(0).insert(prefs[0].begin(), {"ECS 20", "ECS 36A"});
prefs.at(1).insert(prefs[1].begin(), {"ECS 20", "ECS 36B", "ECS 36C"});
prefs.at(2).insert(prefs[2].begin(), {"ECS 36B", "ECS 36C"});
prefs.at(3).insert(prefs[3].begin(), {"ECS 20", "ECS 36C"});
prefs.at(4).insert(prefs[4].begin(), {"ECS 36C"});
auto assignment = assignTutoring(prefs,
    {"ECS 20", "ECS 36A", "ECS 36B", "ECS 36C"});
```

You should convince yourself that there is a bipartite graph within this setup. Each tutor would have a corresponding node, and each course would have a corresponding node. What would the edges be? You should be able to piece together the nature of the bipartite graph from here.

Since you already did (or will have already done) rigorous input validation for `solveNetworkFlow()`, you will not have any input validation for `assignTutoring()`. You get to assume that the input is valid, properly formatted, etc.

5.3 Example

On Canvas, you can find a file called `demo_netflow.cpp`. Below is the file and an example output. Because an instance of the network flow problem may have multiple maximum flows, it is possible that your output from running `demo_netflow.cpp` may differ from mine.

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

#include <iostream>

static void printAdjList(const AdjList& adjList)
{
    for (unsigned from = 0; from < adjList.size(); ++from)
    {
        const auto& neighborList = adjList[from];
        std::cout << from << ": ";
        for (unsigned toIndex = 0; toIndex < neighborList.size(); ++toIndex)
            std::cout << neighborList[toIndex].neighbor << " ("
                << neighborList[toIndex].weight << ") ";
        std::cout << '\n';
    }
}

static void printAssignment(const std::vector<std::string>& preferences)
{
    for (unsigned i = 0; i < preferences.size(); ++i)
        std::cout << "Tutor #" << i << " assigned to: " << preferences[i] << '\n';
}

int main()
{
    // Create the flow network shown in slide 59 of the graphs slide deck.
    AdjList adjList{6};
    // I could have also done adjList[0].emplace_back(1, 5) if NeighborEntry
    // had a constructor that took the neighbor and weight as parameters.
    adjList[0].push_back({1, 5});
    adjList[0].push_back({2, 10});
    adjList[2].push_back({1, 5});
    adjList[1].push_back({3, 10});
    adjList[2].push_back({3, 3});
    adjList[2].push_back({4, 1});
    adjList[3].push_back({4, 20});
    adjList[3].push_back({5, 5});
    adjList[4].push_back({5, 7});
    auto maxFlow = solveNetworkFlow(adjList);
    std::cout << "=== Adjacency list of the flow ===\n";
    printAdjList(maxFlow);
    std::vector<std::vector<std::string>> prefs(5);
    prefs.at(0).insert(prefs[0].begin(), {"ECS 20", "ECS 36A"});
    prefs.at(1).insert(prefs[1].begin(), {"ECS 20", "ECS 36B", "ECS 36C"});
    prefs.at(2).insert(prefs[2].begin(), {"ECS 36B", "ECS 36C"});
    prefs.at(3).insert(prefs[3].begin(), {"ECS 20", "ECS 36C"});
    prefs.at(4).insert(prefs[4].begin(), {"ECS 36C"});
    auto assignment = assignTutoring(prefs,
        {"ECS 20", "ECS 36A", "ECS 36B", "ECS 36C"});
    std::cout << "=== Tutor assignments ===\n";
    printAssignment(assignment);
}
$ g++ -Wall -Werror -std=c++14 demo_netflow.cpp netflow.cpp
$ ./a.out
=== Adjacency list of the flow ===
0: 1 (5) 2 (7)
1: 3 (8)
2: 1 (3) 3 (3) 4 (1)
3: 4 (6) 5 (5)
4: 5 (7)
5:
=== Tutor assignments ===
Tutor #0 assigned to: ECS 36A
Tutor #1 assigned to: ECS 36B
Tutor #2 assigned to: ECS 36C
Tutor #3 assigned to: ECS 20
Tutor #4 assigned to:
```

UCDAVIS
COMPUTER SCIENCE