CS-472: Design Technologies for Integrated Systems

Programming Assignment 1 Due date: 13/10/2022, 24:00

1 Introduction

In this homework, you are asked to implement the Bellman-Ford shortest path algorithm. A similar algorithm, Dijkstra's Algorithm, is implemented for you as a reference. The pseudocode can be found in the textbook on page 55.

2 Input Format

The first line of the input file contains a number n specifying the number of nodes in the graph. A node is represented with an integer i, where $0 \leq i \leq n-1$. Then, each of the remaining lines specifies a weighted (directed) edge with three numbers. The first number is the node where the edge goes out, and the second number is the node where the edge goes in. The third number is the weight of the edge, which can be negative.

If there are repeated edges, the weight in its last appearance should be stored. Self-loops should be removed. These are already taken care of with the provided code.

3 Coding Interface

A template code is provided. You should implement your data structure in `graph.hpp`, where you should have a class `Graph` with the following interfaces:

- `Graph( uint32_t n )`
  The constructor. n is the number of nodes.

- `void add_edge( uint32_t from, uint32_t to, int32_t weight )`
  This function will be called by `main.cpp` when an edge from `from` to `to` with weight `weight` is read in.

- `vector<uint32_t> bellman_ford_shortest_path( uint32_t source, uint32_t sink )`
  This is where you should implement Bellman-Ford Algorithm. If there is no path from `source` to `sink` or if there are negative-weight loops in the graph that is reachable from the source (even though it is not on the path from source to sink), the return value should be an empty vector. If `source` is the same as `sink`, the return value should be a vector with only one element, which is this node. Otherwise, the first element of the returned vector should be `source` and the last element should be `sink`, and there should be an edge between each pair of neighboring elements. The *length* of the path is defined as the sum of the weights of these edges. The algorithm should return a path of the minimum possible length. If there are multiple minimum-length paths, any one of them is an acceptable solution.

A graph data structure is already implemented for you, but you are free to change anything in `graph.hpp` (including the `Node` class), as long as you keep the interfaces of the three functions above the same. Search for `TODO` to find where you should do your work. You are NOT allowed to modify other files, nor to add additional files, as only `graph.hpp` will be copied to the grading directory.

4 Compiling, Testing and Submission

With a command line interface, type `make` to compile the code. An executable named `hw1` will be produced if compilation is successful.

Type `./hw1 <path_to_test_file> <source> <sink> <d|b>` to test the code, where the last argument indicates which algorithm to use (`d` for Dijkstra's and `b` for Bellman-Ford). For example, type `./hw1 testcases/test01.txt 3 5 d` to test on the first test case to find the shortest path from node `3` to node `5` with Dijkstra's Algorithm.

Some expected results are provided in the file `ref_output.txt`.

Please submit `graph.hpp`, compressed in one ZIP, to Moodle. Please do not change the name(s) of the `hpp` file(s).

5 Questions for Yourself

Think about these questions (no need to submit; we will discuss in the exercise session).

- What does it mean to have negative-weight loops?
- How do negative-weighted edges affect the shortest-path algorithms?
- Why do the two shortest-path algorithms have different expected results in some test cases?
- What are the differences between the two shortest-path algorithms? (Complexity? Efficiency? Limitations?)