CSE310 Project 2: Finding a Shortest Path
Posted: Thursday, 04/02/2020, Due: Sunday, 04/26/2020

This is a programming project. You should use the C++ programming language, not any other programming language. **All programs will be compiled and tested on gradescope**. You will need to submit it electronically at gradescope, in one zip file, named CSE310-P02-Lname-Fname, where Lname is your last name and Fname is your first name. The zip file should contain a set of files that are absolutely necessary to compile and execute your program. **Refer to the announcement on April 12 for submission instructions**.

In your first programming project, you have implemented the `max-heap` data structure and its associated functions. In this programming project, you have to implement the min-heap data structure and its associated functions. You have also to expand the fields of `ELEMENT` to contain other fields as needed. Then you will add a module to implement Dijkstra's shortest path algorithm. Your program should be able to read in a graph, and build the `edge adjacency list` of the graph. Note that the edge adjacency list is an array (indexed by the vertices) of singularly linked lists, where the list according to a node v denotes the outgoing neighbors of node v. When given a source-destination pair, your program should compute the shortest path from the source to the destination, and output the result as required. Dijkstra's shortest path algorithm uses a min-heap of the vertices of the graph, where the key value at a node is the currently known distance from the source to the given node. You have to define the data types as needed.

You should implement a module that takes the following commands from the key-board and feed to the main program:

- **S**
- **R**
- **W**
- **F s t iFlag**

On reading **S**, the program stops.
On reading **R**, the program reads in an edge weighted directed graph from file `Ginput.txt`, builds the adjacency lists, and waits for the next command.

The file `Ginput.txt` is a text file. The first line of the file contains two integers n and m, which indicate the number of vertices and the number of edges of the graph, respectively. It is followed by m lines, where each line contains three integers: u, v, and w. These three integers indicate the information of an edge: there is an edge pointing from u to v, with weight w. Please note that the

vertices of the graph are indexed from 1 to n (not from 0 to $n-1$).
On reading **W**, the program writes the graph information to the screen, and waits for the next command.

The screen output format of **W** is as follows: The first line should contain two integers, n and m, where n is the number of vertices and m is the number of edges. It should be followed by n lines, where each of these n lines has the following format:

$$u : (v_1, w_1); (v_2, w_2); \ldots ; (v_k, w_k)$$

On reading **F s t 1**, the program runs Dijkstra's shortest path algorithm to compute a shortest s-t path, and prints out the length of this shortest path to the screen, and waits for the next command. The information printed to the screen should be of the format: `LENGTH: l`, where `l` is the path length.

On reading **F s t 0**, the program runs Dijkstra's shortest path algorithm to compute a shortest s-t path, and prints out the path of this shortest path to the screen, and waits for the next command. The information printed to the screen should be of the format: `PATH:` $s, v_1, v_2, \ldots, t$, where the vertices listed gives out the path computed.

Please note that your program should read in only one graph, but may be asked to compute s-t shortest paths for different pairs of s and t. Therefore, during the computation of the s-t shortest path, your program should not modify the given graph.

You should use modular design. At the minimum, you should have

- the main program as `main.cpp` and the corresponding `main.h`;
- the heap functions `heap.cpp` and the corresponding `heap.h`;
- the graph functions `graph.cpp` and the corresponding `graph.h`;
- various utility functions `util.cpp` and the corresponding `util.h`.

You should also provide a Makefile which compile the files into an executable file named `proj2`.
Grading policies: We will manually check your submission to see whether you have provided a functioning Makefile (10 points), implemented modular design (10 points), provided documentation (10 points), and performed graph I/O (10 points) (40 points total). The remaining 60 points come from the posted + unposted test cases. There are 30 posted test cases (which can you view yourself on Canvas) and 30 unposted test cases (which are not visible to you). When you submit your project to gradescope, you will see immediately how many of these test cases you pass, and thus how many points you get for this grading criteria.

(10 pts) You should provide a Makefile that can be used to compile your project on gradescope. The executable file should be named proj2. If your program does not pass this step, you will receive 0 on this project.

(10 pts) Modular design: You should have a pair of implementation and header files for each of the following: utility, heap, graph.

(10 pts) Documentation: You should provide sufficient comment about the variables and algorithms. You also need to provide a README file describing what you are trying to achieve in this project.

(10 pts) Graph I/O: The program reads the graph from a file into memory and outputs the necessary information.

(30 pts) Your program should produce the correct output for a posted set of test cases.

(30 pts) Your program should produce the correct output for an unposted set of test cases.