Title: Methods and Tools for Software Engineering
Course ID: ECE 650
Lectures: Thursday, 05:30 – 08:20
Instructor: Prof. Alireza Sharifi, `first.last@uwaterloo.ca`
TA: Hari Govind Vediramana Krishnan, `hgvedira@uwaterloo.ca`

Office hours by appointment. Begin all email subjects with `[ECE650]`.

Assignment 2 - Due Friday, October 18, 2017

The GitHub repository will be announced.

- For this assignment, you need to write a program that takes input till it sees an `EOF`.
- One of the kinds of input contributes towards the specification of an undirected graph.
- Another kind of input asks you to print out a shortest-path from one vertex to another in the current graph.
- Write your code in C++. Ensure that it compiles with the GCC version that is on `ecelinux[1-3]`, and runs on `ecelinux[1-3]`.

Resources

The book, "Introduction to Algorithms", by Cormen, Leiserson, Rivest & Stein, may be useful to you. The 2^{nd} ed. is available at the library. It is also accessible online via `lib.uwaterloo.ca`. Section 22.1 discusses how a graph may be represented. For determining a shortest-path, you can use a simple algorithm, such as Breadth-First Search (22.2), or Bellman-Ford (24.1).

Sample Run

Assume that your executable is called `a2-ece650`. In the following, "`$`" is the command-prompt.

```
$ ./a2-ece650
V 15
E {<2,6>,<2,8>,<2,5>,<6,5>,<5,8>,<6,10>,<10,8>}
s 2 10
2-8-10
V 5
E {<0,2>,<2,1>,<2,3>,<3,4>,<4,1>}
s 4 0
4-1-2-0
```

Input, Output, and Error

Your program should take input from standard input, and output to standard output. Errors can also be output to standard output, but should always start with "`Error:`" followed by a brief description. Your program should terminate gracefully (and quietly) once it sees `EOF`. Your program should not generate any extraneous output; for example, do not print out prompt strings such as "`please enter input`" and things like that.

As the example above indicates, one kind of input is the specification of a set of vertices `V`, and set of edges `E` of the undirected graph. The specification of a set of vertices starts with '`V`', followed by a space, followed by a positive integer, all in one single line. If the integer that follows the `V` is i, then we assume that the vertices are identified by $0, \ldots, i-1$.

The specification for a set of edges starts with '`E`'. It then has a space, followed by the set of edges in a single line delimited by '{' and '}'. The two vertices of an edge are delimited by '`<`' and '`>`' and separated by a comma. The edges in the set are also separated by a comma. There are no whitespace characters within the { }.

The only other kind of input starts with an 's'. It asks for a shortest path from the first vertex to the second that is specified after the `s`. The `s` is followed by a space, a vertex ID, another space, and a second vertex ID. The lines `2-8-10` and `4-1-2-0` above are outputs of the `s` commands that immediately precede them. The output comprises vertex IDs separated by `-`, with no whitespace within. If a path does not exist between the vertices, you should output an error.

The graph is specified by the specification of the set of vertices `V` followed by the set of edges `E`, in that order. `V` and `E` always occur together. There is no relationship between subsequent graph specifications; when a new `V` specification starts, all previous information can be forgotten. After the specification of the graph there can be zero or more shortest-path queries `s`. For each `s` query, only one shortest path should be output; multiple shortest paths might exist and an arbitrary choice can be made.

We will not test your program for format errors in the input. That is, our input will be perfectly formed. However, we will test your program for other kinds of errors. For example, if we have `V 5`, we may try to specify an edge `<2,10>`, for a vertex `10` that does not exist. Similarly, we may ask for a shortest path to a vertex that does not exist. We may also ask for a shortest path when the vertices exist, but a path does not exist between them. Errors should be output in such cases.

Marking

Your output has to perfectly match what is expected. You should also follow the submission instructions carefully. It discusses how to name your files, how to submit, etc. The reason is that our marking is automated.

- Does not compile/make/crashes: automatic 0
- Your program runs, awaits input and does not crash on input: + 20
- Passes Test Case 1: + 20
- Passes Test Case 2: + 20
- Passes Test Case 3: + 15
- Correctly detects errors: + 20
- Programming style: + 5

CMake

As discussed below under "Submission Instructions", you should create a `CMakeLists.txt` file to build your project. We will build your project using the following sequence:

```
cd PROJECT && mkdir build && cd build && cmake ../ && make
```

where `PROJECT` is the top level directory of your submission. If your code is not compiled from scratch (i.e., from the C++ sources), you get an automatic 0.

Submission Instructions

You should place all your files at the top of a github repository. The repository should contain:

- All your C++ source-code files; you may use further subdirectories if you wish;
- A `CMakeLists.txt`, that builds a final executable named `a2ece650`;
- A file `user.yml` that includes your name, WatIAM, and student number.

See `README.md` for any additional information.

The submitted files should be at the top of the `master` branch of your repository.