Programming Assignment Five: Graph Algorithms

Out: Nov. 28, 2017; Due: Dec. 16, 2017.

I. Motivation

1. To give you experience in implementing a graph data structure using the adjacency list representation.
2. To give you experience in implementing a few graph algorithms.

II. Programming Assignment

You will read from the **standard input** a description of a directed graph and then report three things on that graph:

1. The shortest path between a source node and a destination node specified in the input.

2. Whether the graph is a directed acyclic graph (DAG).

3. Determine the total weight of a minimum spanning tree (MST) of the graph when it is treated as an undirected graph.

1. Input Format

The first line in the input specifies the number of nodes in the graph, N. The nodes in the graph are indexed from 0 to $N-1$. The second line specifies a source node $0 \leq s \leq N-1$ and the third line specifies a destination node $0 \leq d \leq N-1$. You should report the shortest path from s to d. Each subsequent line represents a **directed** edge by 3 numbers in the form:

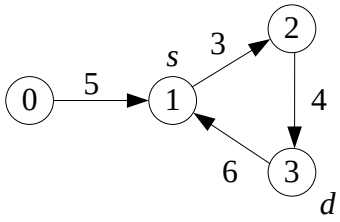
```
<start_node> <end_node> <weight>
```

where both `<start_node>` and `<end_node>` are integers in the range $[0, N-1]$, representing the start node and the end node of the edge, respectively, and `<weight>` is a **non-negative integer** representing the edge weight. Thus, the graph specified in this format is a directed graph with non-negative edge weights.

An example of input is

```
4
1
3
0 1 5
1 2 3
2 3 4
3 1 6
```

It represents the following directed graph with the source node as Node 1 and the destination node as Node 3:



Typically, we will describe the graph in a file. However, since your program takes input from the standard input, you need to use the Linux input redirection "<" on the command line to read the graph from the file.

2. Output Specification

Your program writes to the **standard output**. It will first show the shortest path. Then, it will tell whether the graph is a DAG or not. Finally, it will calculate the total weight of a MST of the original graph when treated as an undirected graph.

a) Showing the shortest path

If there exists a path from the source node to the destination node, your program should print the length of the **shortest** path. Specifically, it should print the following line:

```
Shortest path length is <length>
```

where <length> specifies the length of the shortest path. In the special case where the source node is the same as the destination node, the shortest path length is 0.

If there exists no path from the source node to the destination node, your program should print:

```
No path exists!
```

b) Telling whether the graph is a DAG or not

If the graph is a DAG, your program should print:

```
The graph is a DAG
```

Otherwise, print:

```
The graph is not a DAG
```

c) Calculating the total weight of an MST

You should treat all the directed edges in the original graph as undirected edges, and obtain the total edge weight of a minimum spanning tree for the undirected graph. If there exists a spanning tree, your program should print the total edge weight of an MST. Specifically, it should print the following line:

```
The total weight of MST is <total_weight>
```

where <total_weight> specifies the total edge weight of an MST.

If there exists no spanning tree, your program should print:

```
No MST exists!
```

For the example graph shown above, one valid output should be:

```
Shortest path length is 7
The graph is not a DAG
The total weight of MST is 12
```

III. Program Arguments and Error Checking

Your program takes no arguments. You do not need to do any error checking. You can assume that all the inputs are syntactically correct.

IV. Implementation Requirements and Restrictions

- You must implement the graph data structure using the **adjacency list representation**.
- You must make sure that your code compiles successfully on a Linux operating system. You are required to write your own `Makefile` and submit it together with your source code files. **Your compiled program should be named as `main` exactly.**
- We highly encourage the use of the STL, such as `vector`, `deque`, `map`, `algorithm`, etc., for this project, with the exception of two prohibited features: The C++11 regular expressions library (whose implementation in gcc 4.7 is unreliable) and the thread/atomics libraries (which spoil runtime measurements). Do not use other libraries (e.g., boost, pthreads, etc).
- Output should only be done where it is specified.

V. Hints on Data Structure

You may want to define the following three abstract data types: node, edge, and graph.

VI. Testing

We have supplied three input files `g1.in`, `g2.in`, and `g3.in` for you to test your program. The outputs of our program for these three inputs are also provided. They are `g1.out`, `g2.out`, and `g3.out`. All these files are located in the Programming-Assignment-Five-Related-Files.zip. To do the test, type a similar command to the following into the Linux terminal once your program has been compiled:

```
./main < g1.in > test.out
diff test.out g1.out
```

If the `diff` program reports any differences at all, you have a bug.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

VII. Submitting and Due Date

You should submit all the related `.h` files, `.cpp` files, and the `Makefile` via the online judgement system. The `Makefile` compiles a program named `main`. Please also include a pdf file containing all of your source code. See announcement from the TAs for the details about how to submit these files. The submission deadline is 11:59 pm on Dec. 16, 2017.

VIII. Grading

Your program will be graded along four criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style
4. Performance

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. For example, we will check whether you implement the graph data structure using the adjacency list representation. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, if it is hard for the TAs to check whether your graph data structure is implemented using adjacency list, it will lead to General Style deductions. Part of your grade will also be determined by the performance of your algorithm. We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, you will lose the performance score for those test cases.