VE281 — Data Structures and Algorithms

Programming Assignment 4

Instructor: Hongyi Xin
— UM-SJTU-JI (Fall 2020)

Notes

- Due Date: 12/17(tentative)
- Submission: on JOJ

1 Introduction

In this project, You will implement a graph data structure using the adjacency list representation, and then implement some graph algorithms based on the data structure:

- Dijkstra's shortest path algorithm
- Topological sort
- Prim's algorithm or Kruskal's algorithm

2 Programming Assignment

2.1 Input and Output

2.1.1 Input Specification

The first line in the input specifies the number of nodes $|V|$ in the graph $G = (V, E)$, where the nodes in the graph are indexed from 0 to $|V| - 1$. The second line specifies a source node $0 \leqslant s \leqslant |V| - 1$ and the third line specifies a destination node $0 \leqslant d \leqslant |V| - 1$. 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 $[0, |V|1]$, 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:

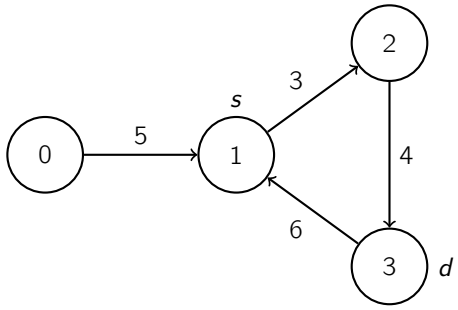

Figure 1: Sample Input

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.

2.1.2 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!
```

Sample Output

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
```

2.2 Design of Data Structures

You are encouraged to design your own object-oriented programming data structure in this project. In general, you need to have at least two Abstract Data Types: `Graph` and `Node` (or `Vertex`).

The `Graph` class should have a constructor which is able to load the graph from the input edges. You can further implement different algorithms, each as a method of the `Graph` class.

The `Node` (or `Vertex`) class should save the intermediate attributes used in the algorithms, such as `distance`, `visited`, and `predecessor`.

You may also need a simple `Edge` ADT with only a few attributes. You can define a class for it, or you can use `std::pair` or `std::tuple` to represent an edge for simplicity.

2.3 Algorithms

2.3.1 Dijkstra's Algorithm

You should use Dijkstra's Algorithm to find the shortest path between a source node and a destination node specified in the input.

You can use the STL container `std::priority_queue` to implement the algorithm. When you're going to push a node into the `std::priority_queue`, you are not able to detect whether the node is already in the queue. In this case, you should push the node anyway, and remove duplication when popping the queue.

2.3.2 Topological Sort

You should use topological sort to detect whether a graph is a DAG, we'll omit the details here. Check the lecture slides for reference.

2.3.3 Minimum Spanning Tree

You can use either Prim's algorithm or Kruskal's algorithm to find the MST. Both of them should be able to pass all the cases on the online judge.

Optional: However, the performance of these two algorithms are different on dense and sparse graphs. If you want to achieve a balanced time complexity, you may need to implement both of them and decide which one is better for a specific graph.

3 Implementation Requirements and Restrictions

3.1 Program

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

You should write all of your code in one file `main.cpp`.

3.2 Requirements

- You must make sure that your code compiles successfully on a Linux operating system with `g++` and the options `-std=c++1z -Wconversion -Wall -Werror -Wextra -pedantic`.
- You must implement the graph data structure using the **adjacency list representation**.
- You should only hand in one file `main.cpp`.
- We highly encourage the use of the STL! You can use any header file defined in the `C++17` standard. You can use cppreference as a reference.
- Output should only be done where it is specified.

3.3 Memory Leak

You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call valgrind, which can tell whether you have any memory leaks. (You need to install valgrind first if your system does not have this program.) The command to check memory leak is:

```
valgrind --leak-check=full <COMMAND>
```

You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

```
./main < input.txt
```

causes memory leak, then `<COMMAND>` should be "`./main < input.txt`". Thus, the command will be

```
valgrind --leak-check=full ./main < input.txt
```

4 Grading

Your program will be graded along five criteria:

4.1 Functional Correctness

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program.

4.2 Implementation Constraints

We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points.

4.3 General Style

General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm.

4.4 Performance

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.

4.5 Report on the performance study

Finally, we will also read your report and grade it based on the quality of your performance study.

5 Acknowledgement

This programming assignment is designed based on Weikang Qian's VE281 programming assignment 5.