

EECS 560 Lab 11: Graph 2

Due date

11/15/2020 Sunday, 11:59 pm

Objective

Get familiar with adjacency list representation of graph with C++. Find the shortest path from given source vertex using Dijkstra's algorithm.

Requirements to complete the template

1. Extend the graph class "**myGraph**" from lab 10.
2. Extend your backend data structure to store the adjacency list and **distance of edges**. Note that you must choose the backend data structure wisely so that the actual running time and memory usage doesn't exceed the allowed limit. See **Testing and Grading** for more details.
3. Extend the constructor **myGraph(char* filename)** that create the graph object using the provided instance file with name **filename**. The instance file has format:
s₁,t₁₁,c₁₁,s₁,t₁₂,c₁₂,...,s₂,t₂₁,c₂₁,s₂,t₂₂,c₂₂,... (separated by comma), where each tuple (**s,t,c**) stands for one directed edge from source vertex **s** to target vertex **t** with cost/distance **c**.
4. Extend the destructor **~myGraph()** if needed.
5. Implement public member functions **void print2()** that print the distance of all edges using **cout** in following format:
<s₁>: <c₁₁> <c₁₂> ...
<s₂>: <c₂₁> <c₂₂> ...
...
where **<c_{ij}>** is the cost/distance of the edge from vertex **i** to its **j**-th target, and are separated by space ' ', don't leave space at the end of each line. Note that there is a string **:** after the source.
6. Implement public member functions **void dijkstra(int source)** that finds the shortest paths from vertex **source** to all reachable vertices. Refer to Figure 9.31 and Figure 9.29 to implement your function.
7. Implement public member functions **void printPath(int i)** that print the shortest path from vertex **source** to vertex **i**. Refer to Figure 9.30 to implement your function.
8. Implement public member functions **int computePathCost(int i)** that returns the cost of the shortest path from vertex **source** to vertex **i**. Use -1 to indicate that there is no path from vertex **source** to vertex **i**.
9. Any other internal (private) functions you need to implement the previous functions.

Testing and Grading

We will test your implementation using a tester main function, on a number of instances that are randomly generated. We will release the tester main function, several instances (will be different from the grading instances but will have the same format), and expected output together with the lab instruction via Blackboard. Your code will be compiled under Ubuntu 20.04 LTS using g++ version 9.3.0 (default) with C++11 standard.

The command line we are going to use for compiling your code is: “g++ -std=c++11 main.cpp” (note that main.cpp will try to include the .hpp file you submit, and your .hpp file needs to be properly implemented to compile successfully).

Your final score will be the percentage your program passes the grading instances. **Note that if your code does not compile (together with our tester main function), you will receive 0.** Therefore, it is very important that you ensure your implementation can be successfully compiled before submission.

Your program will be considered “fail” for the cases if any of the running time or maximum resident set size exceeds 5 times of the provided benchmark file.

Submission and Deadline

Please submit your implementation as one .hpp file, with file name “myGraph_[YourKUID].hpp”. For example, if my KU ID is c123z456, my submission will be a single file named “**myGraph_c124z456.hpp**”. Submissions that do not comply with the naming specification will not be graded. Please submit through Blackboard.