Project #06: Graph Algorithms

Complete By: Wednesday, November 27th @ 11:59pm
Assignment: BFS, DFS, and Dijkstra's algorithm
Policy: Individual work only, late work *is* accepted (see "Policy" section for more details)
Submission: via Gradescope

Background

In lab and HW you've been building and working with graphs. Our graph class is limited to 26 vertices 'A' – 'Z', but it's enough to allow the investigation of various graph algorithms. In this assignment you're going to write a program to perform the following three algorithms: BFS, DFS, and Dijkstra's shortest weighted path. Much of the code has already been developed, and you are free to use any code from lab and HW.

Program Overview

The program will input a graph from a text file, exactly as we have been doing in the lab and HW. The graph is then output, and the user is prompted for a starting vertex. Based on that starting vertex, the program outputs the neighbors, and then the results from traversing via BFS, DFS, and Dijkstra's algorithm. This is repeated until the user enters '#' for the starting vertex. Screenshot:

```
Enter filename containing graph data> graph1.txt
**Vertices: A B C D E F
**Edges: (A,B,100) (A,D,115) (B,A,80) (B,C,20) (B,D,12) (C,E,29) (D,C,7) (E,C,29)

Enter a starting vertex or #> A
Neighbors: B D
BFS: A B D C E
DFS: A D C E B
Dijkstra: A B D C E
 A: 0 via A
 B: 100 via A B
 D: 112 via A B D
 C: 119 via A B D C
 E: 148 via A B D C E

Enter a starting vertex or #>
```

File input

The input to the program will come from a text file that contains N > 0 lines. The file format is exactly the same as from earlier lab and HW: one vertex per line, following by #, then one edge per line, followed by #. Assume all weights are positive, and that multi-edges will not occur (i.e. there is at most one edge between any two vertices). Here's the "graph1.txt" input file:

```
A
B
D
C
F
E
#
A B 100
B A 80
A D 115
B D 12
B C 20
C E 29
E C 29
D C 7
#
```

Four sample input files are provided on the course web site under Projects, project06-files: "graph1.txt", "graph2.txt", "graph3.txt", and "graph4.txt". Visual drawings of graphs 1 – 3 are given in this PDF.

The code to read the text file and build the graph has been provided in lab and HW; see the function **buildGraph()**. This function should be in the same C++ file (util.cpp?) as your BFS and DFS solutions. Feel free to reuse this function exactly as provided. If you decide to write your own function, note that since we are not inputting strings that may contain spaces, you can use the **>>** input operator instead of getline().

Provided files and implementation details

Besides sample input files, we are providing a **graph** class (similar to what has been used in lab and HW), and a **minqueue** class needed by Dijkstra's algorithm. Note that a main.cpp file is *not* provided; that is intentional. If you prefer to work on Codio, the project "**cs251-project06-graphs**" is provided with these files + a makefile. If you prefer to work outside Codio, the files are provided in both Linux and Mac-Windows formats: download from course web page under Projects, project06-files.

When it comes time to implement Dijkstra's algorithm, take a look at the provided **minqueue** class. This provides the "PopMin" function shown in the zybooks algorithm. Note that clarifications to Dijkstra's algorithm will be given in class, so you should check the lecture notes starting from Friday Nov 22 (Day 37), and beyond. Your implementation of Dijkstra's algorithm should compute the set of vertices that are visited, in the order they are visited, must like BFS and DFS. In addition, your implementation also needs to compute the minimum distance to each of these vertices, and the predecessors back to the starting vertex.

Program behavior

The program starts by inputting a filename. The program should check to make sure the file exists, and halt with an error message if not. If the file exists, open the file, build the graph, and then output to the screen; note that the provided **graph** class now contains an **output()** function to output the graph for you. The program then prompts for a starting vertex, and outputs the results of neighbors, BFS, DFS, and Dijkstra's algorithm. This is repeated until the user enters the sentinel '#'. Here's another screenshot:

```
Enter filename containing graph data> graph3.txt
**Vertices: A B C D E X
**Edges: (A,B,100) (A,D,115) (B,A,80) (B,C,20) (B,D,12) (C,B,19) (C,E,29) (C,X,2000) (D,C,7) (E,A,999)
(E,C,29) (X,D,2900)

Enter a starting vertex or #> A
Neighbors: B D
BFS: A B D C E X
DFS: A D C X E B
Dijkstra: A B D C E X
 A: 0 via A
 B: 100 via A B
 D: 112 via A B D
 C: 119 via A B D C
 E: 148 via A B D C E
 X: 2119 via A B D C X

Enter a starting vertex or #> F
Not a vertex in graph, ignored...

Enter a starting vertex or #> X
Neighbors: D
BFS: X D C B E A
DFS: X D C E A B
Dijkstra: X D C B E A
 X: 0 via X
 D: 2900 via X D
 C: 2907 via X D C
 B: 2926 via X D C B
 E: 2936 via X D C E
 A: 3006 via X D C B A

Enter a starting vertex or #> #
```

Note that if the user enters a vertex that doesn't exist, e.g. 'F' as shown above or illegal chars such as '7' or 'a', then the program should output an error message. With regards to the output shown for Dijkstra's algorithm, the first output is the set of visited vertices, in order they are visited --- much like BFS and DFS. Then, for each visited vertex, the minimum distance is output, along with the path that achieves this minimum distance.

Requirements

1. You must use the provided "graph.h" class as your graph.
2. You may not change the provided "graph.h" class --- we will grade your submission using our graph.h.

3. You must have functions that implement BFS, DFS, and Dijkstra's alg. Those functions, and any functions they call, may *not* output to a file or the console. You must pass all data in via parameters, and return all results via parameters / return value. If your function breaks this requirement, we will mark it as incorrect and score this aspect of the assignment as 0.

4. The max nesting of loops is 2 --- i.e. any function that contains more than two levels of loop nesting will be deleted. Any functionality that depends on the deleted function will not be graded and scored as 0. Use functions. Here's an example of code that would violate this requirement:

```
while (…)
{
  .
  .
  for (…)
  {
      for (…)
      { }
  }
  .
  .
}
```

The code has 3 levels of explicit loop nesting. The solution is to turn one of the loops into a function, and call it, that's fine. The goal is to encourage more use of functions.

Program submission

Submit all program files (.cpp, .h, .hpp) on Gradescope to "**Project 06 – Graphs**". Unlike previous projects, we will not auto-grade this one. Your submission will be checked to ensure that it compiles, but that's it. Pay close attention to the requirements, e.g. do *not* modify the provided **graph** class, otherwise your code will not compile. And code that doesn't compile won't be graded.

Your program should match the output shown in the screenshot as closely as possible, and provide all required output. The only error checking required is the file name, and the vertex input by the user. For grading, 80% will be based on correctness, and 20% based on commenting, readability, and approach. In particular, we are going to be looking for good use of functions. If you break one of the requirements, we may very well score your submission as a 0.

Policy

Late work *is* accepted. Since the late period falls over Thanksgiving break, you may submit as late as Sunday, December 1st @ 11:59pm for a penalty of 10%. After Sunday December 1st, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for

grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .