CS 2233 Data Structures and Algorithms
Final Exam Programming Assignment
Graphs

Preliminary: Graph definition

Definition: a *directed weighted graph G* consists of a set *V* of vertices, a set *E* of edges such that each edge *e ϵ E* is associated with an ordered pair of vertices, and a weighting function *w* which maps *E* onto the set of real numbers, where *e = (a, b)* denotes an edge *e* from vertex *a* to vertex *b*, and *w(e)* is the weight of edge *e*.

Assignment: Given any connected, weighted graph *G*, use Dijkstra's algorithm to compute the shortest (or smallest weight) path from any vertex *a* to any other vertex *b* in the graph *G*. You may print the results of this algorithm to the screen or to a log file. Your solution should be complete in that it shows the shortest path from all starting vertices to all other vertices. Complete the exercises below, then submit your code files and your exercise worksheet, and all supporting documents to Canvas before beginning the written portion of the final exam. You may use an external source as a reference for Dijkstra's shortest path algorithms, please cite accordingly.

Notes:
This is the programming component of the final examination for this class, and so you will be required to do all the work for this problem from the ground up working by yourself. In class we have been reviewing some ways to implement portions of this program. While you are not required to follow those suggestions exactly, you will be required to develop an object-oriented solution in C++ which uses ADTs (of your own design) to arrive at a working solution. A list of action items you will need to follow are:

1. Develop a Graph ADT which is possibly composed of ADTs for edges, vertices, and weights
2. Create a parser to read in a graph specification file and populate the graph ADT
3. Develop an "ADT checker" to ensure that a graph ADT is not malformed (edges which connect non-existent vertices, vertices which are orphans, etc.)
4. Refine the methods of the ADTs you create to support Dijkstra's shortest path algorithm (what queries to a graph ADT are necessary or preferred for this algorithm?)
5. Implement Dijkstra's shortest path algorithm against the graph ADT you have created
6. Develop a method to print to the screen the shortest paths from all vertices to all other vertices in the graph
7. Export a well-formed graph to a file which can be read and displayed by an online GraphViz tool such as https://dreampuf.github.io/GraphvizOnline

The specification of a graph will conform to the following format:

```
# Comment lines start with a # character.
GRAPH: name=GraphName
VERTEX: name=SrcNodeName
VERTEX: name=DstNodeName
EDGE: (SrcNodeName, DstNodeName, Weight)
```

Grading:

Program compiles and produces well-formatted output	40%
Program properly implements a graph ADT	20%
Program gives correct output for given test cases	20%
Program gives correct output using my tests:	10%
Code is formatted consistently and coding standards are followed:	10%

Exercise 1:
Given this simple graph, use your program to compute the shortest path and complete the table below

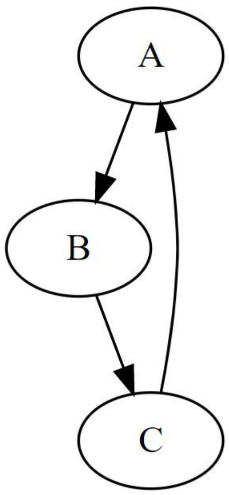
```
# Comment lines start with a '#' character.
# Blank lines are also ignored.

GRAPH: name=Exercise1
VERTEX: name=A
VERTEX: name=B
VERTEX: name=C

EDGE:  (A, B, 1)
EDGE:  (B, C, 3)
EDGE:  (C, A, 4)
```

Destination

		A		B		C	
		Path	Weight	Path	Weight	Path	Weight
Source	A						
	B						
	C						

Draw the graph in the space provided below:

	A		B		C	
	PATH	WEIGHT	PATH	WEIGHT	PATH	WEIGHT
A	no path	0	B	1	BC	4
B	CA	7	no path	0	C	3
C	A	4	AB	5	no path	0



Exercise 2:

Given a more complex graph, use your program to compute the shortest paths and complete the table below

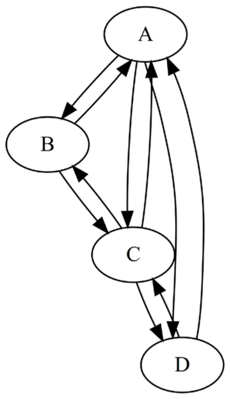
```
GRAPH: name=Exercise2
VERTEX:  name=A
VERTEX:  name=B
VERTEX:  name=C
VERTEX: name=D

# Edges are directional, we can make an undirected graph by reversing edges in a directed graph.
EDGE: (A, B, 6)
EDGE:  (B, A, 6)
EDGE:  (A, D, 5)
EDGE:  (D, A, 5)
EDGE:  (A, C, 2)
EDGE:  (C, A, 2)
EDGE:  (B, C, 3)
EDGE:  (C, B, 3)
EDGE:  (D, C, 4)
EDGE:  (C, D, 4)
```

Destination

		A		B		C		D	
		Path	Weight	Path	Weight	Path	Weight	Path	Weight
Source	A								
	B								
	C								
	D								

Draw this graph in the space provided below:

	A		B		D		C	
	PATH	WEIGHT	PATH	WEIGHT	PATH	WEIGHT	PATH	WEIGHT
A	no path	0	CB	5	D	5	C	2
B	CA	5	no path	0	CD	7	C	3
D	A	5	CB	7	no path	0	C	4
C	A	2	B	3	D	4	no path	0



Questions:

Q1: Explain how you would use the try/catch exception mechanism to verify that a constructed graph ADT was formed correctly.

In my Graph.Edge() and Graph.Insert() function I throw exceptions. If the user inserts a duplicate vertex name it throws an exception and if the user tries to define an edge with a vertex that has not been inserted the function throws an exception. The declarations of both of these functions are surrounded by try/catch blocks.

Q2: How does your program handle the fact that some vertices of a defined graph may not be reachable?

If a node has no edges going to or from it then it is not included in the graph. It is still available for use but until it has a defined edge reaching it or leaving it is not used in displaying the graph or calculating shortest path.

After all, if a vertex is not connected to the main graph it is its own graph and should not be included in display or any path calculations.

Q3: What would your program do if I defined an edge from a vertex to itself with a weight other than zero? What *should* the program do? (note: I will not actually test this, but explain to me what will happen, and what you think is the correct behavior regardless of what your program might do)

It actually throws an error and displays to the console that the node to node edge is not a zero and doesn't put the edge into the graph. Even though it doesn't put the edge into the graph, if the program had done so it would actually not use the greater than zero edge as the shortest route from a vertex to itself. It would maintain that the shortest route is still zero. For instance, I could leave lipscomb go around Nashville for five miles without making any stops then come back to LU. In this case the route from lipscomb to LU is five miles but it is still not the fastest route since I could have just stayed on the campus and maintained a route of zero. I think the correct behavior would be inserting it anyway and letting the algorithm take care of it.