NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

GRAPHS

INTRODUCTION

The purpose of this assignment is to work with directed graphs. You'll implement a class that allows a user to query various info about a graph, which also implements several of the graph algorithms discussed in class.

YOUR TASK

You are responsible for implementing all of the following functions:

- In `graph.h`, you will implement all of the methods of the `Graph` class:
  - `nvertices()`
  - `nedges()`
  - `v_label(vertex)`
  - `v_index(label)`
  - `assign(vertices, edges)`
  - `edge_exists(origin, destination)`
  - `edge_weight(origin, destination)`
  - `undirected_adjacency_list(vertex)`
  - `in_adjacency_list(vertex)`
  - `out_adjacency_list(vertex)`
  - `weighted_adjacency_matrix()`
  - `unweighted_adjacency_matrix()`
  - `depth_first(start, visitfn, connected_only)`
  - `breadth_first(start, visitfn, connected_only)`
  - `toposort()`
  - `dijkstra(start)`

SOME PROVIDED TYPES

In `graph.decl.h` The `GraphEdge` data structure is used to store the info needed for the edges that are used to construct your `Graph`.

```
struct GraphEdge {
  size_t origin;      // index of vertex the edge starts from
  size_t destination; // index of vertex the edge goes to
  double weight;      // the weight/cost associated with this edge
};
```

The `AdjListEdge` is used to store data on edges in your adjacency lists.

```
struct AdjListEdge {
  size_t vertex; // the index of the vertex on the other side of the edge
  double weight; // the weight/cost associated with this edge
};
```

Your implementation of Dijkstra's shortest path algorithm will return a `std::vector` of `dijkstra_row`, which will contain the info on shortest paths to every vertex from the source vertex you passed to the algorithm.

```
struct dijkstra_row {
  size_t  vertex;         // the index of the vertex for this row
  bool    visited;        // whether the vertex has been visited
  double  shortest_known; // distance of shortest known path to this vertex
  ssize_t came_from;      // the index of the vertex we came from for shortest known path, or -1 if N/A
};
```

In `config.h` If you want to store the graph data in a more convenient/efficient format than the edge list that is used to construct it, you are free to do so. Any additional data members that you'd like your `Graph` to contain can be added to the `StudentExtra` struct in `config.h`. It is currently empty, and its use is not required, but it is available if you want it. Any members you add to the `StudentExtra` can be accessed through the `student` member of the `Graph`.

THE METHODS OF `Graph`

`nvertices()` Returns the number of vertices in the graph stored.

`nedges()` Returns the number of edges in the graph stored.

`v_label(vertex)`

- `vertex` - a `size_t` representing the index of the vertex to look up.

Retuns the string label for the vertex index passed in.

`v_index(label)`

- `label` - the `string` label of the vertex whose index you would like.

Returns the `size_t` index of the vertex whose label matches `label`.

`assign(vertices, edges)`

- `vertices` - a `std::vector` of strings containing labels for the vertices in the graph.
- `edges` - a `std::vector` containing a `GraphEdge` for each edge in the graph.

Removes any data currently stored in the `Graph`, and replaces it with the graph described by the vertex vector, `vertices`, and edge list vector, `edges`, passed in.

If you are using the `StudentExtra` structure to store the graph in another format, this is where you would handle that conversion.

Returns nothing.

`edge_exists(origin, destination)`

- `origin` - the index of the vertex the edge starts from
- `destination` - the index of the vertex the edge goes to

Returns `true` if an edge exists going from `origin` to `destination`. Returns `false` if not.

`edge_weight(origin, destination)`

- `origin` - the index of the vertex the edge starts from
- `destination` - the index of the vertex the edge goes to

If an edge exists going from `origin` to `destination`, return its weight/cost. Otherwise return `INFINITY`.

`undirected_adjacency_list(vertex)`

- `vertex` - the index of the vertex whose adjacency list we want.

Returns a `std::vector` containing an `AdjListEdge` for each edge that starts or ends at the vertex with the index passed in.

`in_adjacency_list(vertex)`

- `vertex` - the index of the vertex whose adjacency list we want.

Returns a `std::vector` containing an `AdjListEdge` for each edge that ends at the vertex with the index passed in.

`out_adjacency_list(vertex)`

- `vertex` - the index of the vertex whose adjacency list we want.

Returns a `std::vector` containing an `AdjListEdge` for each edge that starts at the vertex with the index passed in.

`weighted_adjacency_matrix()` Returns a `std::vector` that contains the weighted adjacency list in row major order. Each row represents the corresponding starting vertex, and each column represents the corresponding ending vertex.

If there is an edge going from the starting vertex to the ending vertex, the value of that entry will be the weight/cost of that edge. If not, the value should be `INFINITY` (**NOT** zero).

`unweighted_adjacency_matrix()` Returns a `std::vector` that contains the unweighted adjacency list in row major order. Each row represents the corresponding starting vertex, and each column represents the corresponding ending vertex.

If there is an edge going from the starting vertex to the ending vertex, the value of that entry will be `true`. If not, the value should be `false`.

`depth_first(start, visitfn, connected_only)`

- `start` - index of the vertex to start the traversal from
- `visitfn` - the function (or function-like object) to call when visiting
- `connected_only` - controls whether vertices to which not path from the `start` vertex exists are traversed separately.

Performs a depth first traversal of the graph starting from the vertex whose index is passed in as `start`. When visiting, the `visitfn` will be called on the index of the vertex visited.

When choosing which of the unvisited adjacency vertices to do next, choose them in ascending order of the vertex index numbers.

If `connected_only` is `true`, only start the traversal from the starting vertex.

If `connected_only` is `false`, also traverse the graph from any remaining non-visited vertices, that may have been missed because there was no path to them from the original starting vertex.

`breadth_first(start, visitfn, connected_only)`

- `start` - index of the vertex to start the traversal from
- `visitfn` - the function (or function-like object) to call when visiting
- `connected_only` - controls whether vertices to which not path from the `start` vertex exists are traversed separately.

Performs a breadth first traversal of the graph starting from the vertex whose index is passed in as `start`. When visiting, the `visitfn` will be called on the index of the vertex visited.

When choosing which of the unvisited adjacency vertices to do next, choose them in ascending order of the vertex index numbers.

If `connected_only` is `true`, only start the traversal from the starting vertex.

If `connected_only` is `false`, also traverse the graph from any remaining non-visited vertices, that may have been missed because there was no path to them from the original starting vertex.

`toposort()` Performs the Topological Sort algorithm on the graph. Returns a `std::vector` containing the index of each vertex in the order determined by the algorithm.

If a cycle is found. Stop early and return only the vertices whose order was determined before the cycle. The caller can determine whether a cycle stopped the algorithm early by comparing the length of the vector returned to the number of vertices in the graph.

When ther are multiple choices for the next vertex, choose the one with the lowest vertex index first.

`dijkstra(start)`

- `start` - the index of the source vertex, from which all of the shortest paths will be found.

Use Dijkstra's Shortest Path algorithm to find the shortest path to every vertex from the source vertex whose index is passed in as `start`.

When there are multiple unvisited vertices to choose as the next one (same minimal shortest known path distance), choose the one with the lowest index first.

This will return the table used by the algorithm in its final state, as a `std::vector` containing a `dijkstra_row` for every vertex in the graph.

Remember that Dijkstra's algorithm is not guaranteed to (and likely will not) work if any of the edges have a negative weight. If any of the edges in your graph have a negative weight, print...

```
"Error: Dijkstra's algorithm does not support graphs with negative edge weights.\n"
```

... and return an empty vector.

NOTES

Your work should be done in `graph.h` and `config.h`. Do not alter the other files that were provided with the assignment.

You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission.

TESTING

There are only two testing programs included. These will be used to evaluate the functionality of your implementations of the required functions. Typing `make` will attempt to compile them all, and will succeed to the degree it can with whatever you have implemented at that point.

The table below has a list of the tests available, and they are shown in order from least complex to most complex.

Order	Test	Purpose
1	`01-simple`	Tests the most basic graph functions with known data.
2	`01-loadgraph`	Allows you to test the rest of your functions with data loaded in from a file. Loads from a default file if no filename is passed as a command line argument.

The expected output is contained in the `*.refout` files in the `output/` directory.

HOW TO SUBMIT

In the autograder, submit the following files (and only these files): - `graph.h` - `config.h`

GRADING CONSIDERATIONS

- Does it compile? Does it run? All of the tests should compile and run on turing/hopper with the `Makefile` provided, and points will be deducted for each test that will not compile.
- Does the output match for all of the tests? The autograder will check this, but I have provided reference output for each of the test programs where such a thing is relevant, and you can compare your program's output to what is expected. This output can be found in the files ending in `.refout`.
- Did you indent your code?
  - Indentation aids in the readability of source code, and if you're not indenting your code blocks, the grader will legitimately dislike you for it. I'm authorizing them to mark you off if you subject them to reading that.
- Did you document your code?
  - You need a docbox at the top of every one of the files you're required to change including:
    - Your name
    - Your zid
    - Your course section
    - A description of what the program does
  - You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
  - Add other comments inside your code blocks describing what you're doing and why.
  - The use of `doxygen` style comments is encouraged, but not required.