Programming Assignment 5 - PART-II:

A Dijkstra-based Application

DUE: Sunday, May 1 by 11:59PM (last day before finals week starts!).

[COMMENTING TEMPORARILY ENABLED IF YOU HAVE THIS LINK TO HELP CLARIFY MATTERS SOONER RATHER THAN LATER]

Big Picture

You will implement a binary heap based implementation of Dijkstra's algorithm to support a toy application in which a "traveler" walks around a graph starting from a user-specified start vertex and with a target destination (also user-specified) in mind.

From the user's point of view, vertices are identified by strings (internally, your code will also have an integer vertex-id for each vertex).

Edges between vertices are **undirected** (an edge (u,v) can be traversed from u to v or from v to u).

Edges are also weighted with non-negative floating point numbers indicating some measure of distance of time it takes to traverse the edge.

The application takes a single command line argument: a file containing the graph specification (file format has its own section below).

After reading the graph (assuming there are no errors), the application does the following (an example is given in a later section which might be the best place to start):

> (Setup)
>
> Prints a list of all of the vertices in the graph by name.
>
> Asks the user for their current location which the user enters as a vertex name.
>
> Asks the user for their destination (also specified by vertex name).

Next, the program runs Dijkstra's shortest paths algorithm on the graph. Exactly how you should run Dijkstra is discussed below (to enable the features/behavior I am about to describe).

If the destination vertex is not reachable from the start vertex, then the program prints an appropriate message and terminate.

Otherwise (destination reachable), it then reports the shortest path-length from the given start vertex to the given destination.

It then prints the (technically "an") optimal path as a sequence of vertices (starting from the given start vertex to the given destination of course).

Then...

(interactive loop)

As long as the user has not arrived at their destination, the program repeatedly does the following:

Prints the user's current location (vertex name)

Prints the user-specified destination (vertex name)

Prints the minimum distance from the user's current location to the specified destination.

Lists user options for their next move: give up or select one of the vertices directly connected to the current location.

Option 0 is always "I give up". This just terminates the program.

Options 1, 2, 3 and so on correspond to the neighboring vertices of the user's current location. For each of these options, the vertex

```
            name and the length of that edge is displayed.

        Gives a recommended move (i.e., vertex on a shortest
        path to the destination).

        Reads the user selection (as an integer).

        Assuming it is a valid selection, the state is updated:

            Total distance traveled is updated.

            User's current location is updated.
```

File Format

Recall that we will be working with undirected, edge-weighted graphs.

Input files follow the general format below where vertex-names are strings and edge lengths are non-negative floating point numbers.

```
<number-of-vertices>
<vertex-name> <vertex-name> <edge-length>
<vertex-name> <vertex-name> <edge-length>
.
.
.
<vertex-name> <vertex-name> <edge-length>
```

Thus, the file first tells you the number of vertices in the graph and then lists all of the edges.

An actual example (suppose this is the contents of a file `graph1.txt`):

```
7
locA locB 7.0
locB locC 10.1
locA locD 9.2
locD locC 5.0
locB locD 2.0
locB locE 19.9
locA locE 18.0
locF locE 12.2
locC locF 8.0
```

```
locG locF 11.7
locD locF 15.4
```

Another example:

```
3
Chi Mil 90.0
Mil Min 120
Min Chi 180
```

Detail: vertex names are simply non-empty sequences of non-white-space characters. Thus, the following are technically allowable vertex names:

```
123
--
&
!!abc
```

This makes reading vertex names as simple as possible.

Constraint (actually "assumption"): in all test files we use to grade your program, vertex names will never exceed 10 characters (thus a buffer of size 11 is always sufficient -- one extra slot for the end of string character).

This assumption is in the interest of letting you get to the "good stuff" as quickly as possible.

ASIDE: However, in general, a really robust graph reader might allow arbitrarily long vertex names (the above assumption would likely enable buffer overflows which you know about from 261). Some tools/ideas to make achieving this behavior not *too* difficult:

getline: reads an entire line of input and can be called in a way in which it allocates a sufficiently large buffer for the line (you are still responsible for de-allocating it). Using getline lets us eliminate arbitrary limits on line length. Link: https://www.gnu.org/software/libc/manual/html_node/Line-Input.html

> Once you have the line, of course the length of the line is an upper-bound on the length on the individual vertex names (thereby letting you allocate sufficiently large buffers for them). To allocate buffers for vertex names that are no longer than necessary, you could use strtok to break out the names before allocating buffers and copying.

Example Program Behavior

Below illustrated the expected behavior of the application program through an example.

```
$ ./travel graph1.txt
  Welcome to travel planner.

  Vertices:

    locA locB locC locD locE locF locG

  SELECT YOUR CURRENT LOCATION:   locG
  SELECT YOUR DESTINATION     :   locA

  You can reach your destination in 33.7 units.

  SHORTEST PATH:  locG ->
                  locF ->
                  locC ->
                  locD ->
                  locB ->
                  locA

  TIME TO TRAVEL!

     CURRENT LOCATION:  locG
     DESTINATION:       locA
     MINIMUM DISTANCE TO DESTINATION:  _____

     POSSIBLE MOVES:

         0. I give up!
         1. locF  (11.7 units)

     RECOMMENDED MOVE: _____
```

```
    SELECT A MOVE (ENTER A NUMBER):  1
    TOTAL DISTANCE TRAVELED:         11.7 units

    ------------------------------

    CURRENT LOCATION:  locF
    DESTINATION:       locA
    MINIMUM DISTANCE TO DESTINATION:  _____

    POSSIBLE MOVES:

        0. I give up!
        1. locE  (12.2 units)
        2. locG  (11.7 units)
        3. locC  (8 units)

    RECOMMENDED MOVE: _____

    SELECT A MOVE (ENTER A NUMBER):  1

    TOTAL DISTANCE TRAVELED:        23.9 units

    ------------------------------

    CURRENT LOCATION:  locE
    DESTINATION:       locA
    MINIMUM DISTANCE TO DESTINATION:  _____

    POSSIBLE MOVES:

        0. I give up!
        1. locF (12.2 units)
        2. locB (19.9 units)
        3. locA (8.0 units)

    RECOMMENDED MOVE: 3

    SELECT A MOVE (ENTER A NUMBER):  3

    TOTAL DISTANCE TRAVELED:         41.9 units

    YOU MADE IT.
    (OPTIMAL DISTANCE: ______)
    GOODBYE.

$
```

Data Structures

The table below lists the primary tasks that your data structures must support to achieve the expected behavior. For each task, a recommended data structure enabling the task is recommended along with organizational comments.

Task	Data Structure	Comments
Mapping from vertex name to vertex id	hmap (enables O(1) expected lookup of vertex id from name).	Component of graph data structure
Mapping from vertex-id to vertex name	Array indexed by vertex-id. This is the inverse of the hmap above.	
Mapping from vertex-id to list of neighbors	Array indexed by vertex-id. Can (should?) be same array as for above task. Entries in the list contain vertex-id and edge length.	
Priority Queue	Heap-Based Priority Queue (i.e., part-I of this assignment!)	Enables Fast Implementation of Dijkstra. Conceptually, does not “belong” to the graph data structure.
Data Structure Encapsulating Result	Captures:	Allows post-facto extraction of

of Dijkstra -- a report of sorts.	● source vertex s ● Shortest distances to each reachable vertex from s (d-values) ● Shortest paths tree (implicitly or explicitly). ● pointer to the graph on which Dijkstra was run to generate the report.	shortest paths, etc. Conceptually, is “associated” with a graph, but not “owned” by that graph. Recall how the sample bfs code achieved this.

Code Organization

Modularize!!! You have not been given explicit decomposition of tasks into files, but now and going forward, you should be able to make sensible choices in this regard.

Things to consider:

> I am implementing Dijkstra for this particular application. In the future is it conceivable that I might want to use it for some other application?
>
> Who “owns” what? For example, ideally, everything associated with a graph data structure is truly intrinsic to the graph itself -- not an artifact of some application or operation on the graph.

Makefile: you will submit a makefile supporting the target “travel”:

```
$ make travel
```

should produce your executable.

Miscellaneous / Rules

- Remember, that your implementation of Dijkstra's algorithm must be heap-based; your submission for part-I of this assignment should fit seamlessly into this role.
- You ***are*** allowed to share code from part-I with your classmates. If you didn't quite complete part-I or are concerned it is buggy, you can borrow a solution from a friend (of course, it could be buggy itself, but at least between the two (or more) of you, you have a good chance of getting a correctly working priority queue.