- CPSC 316 PROJECT 3 : NOT QUITE MAPQUEST -

The purpose of this assignment is to use Dijkstra's algorithm to implement the MapQuest©-like program *Not-Quite-Map-Quest*© (*NQMQ*). You will be provided with a list of cities and the roads that connect them, as well as the distances along these roads between pairs of cities. Your program will read a file containing this information, accept starting and ending points, and display the path between the two named cities having shortest distance.

Cities and connecting roads are represented as a weighted graph with no multiple edges or loops. Input regarding cities and roads is read directly from the file `nqmq.dat`; do *not* read this file by redirecting standard input. A sample data file can be downloaded from the web page. A subset of a similar file appears at right. Note the following:

```
3              /* number of cities */
Atlanta_GA          /* city names */
Boston_MA
Chicago_IL
1 3 606   /* dist city1 to city3 */
2 3 860   /* dist city2 to city3 */
```

1. The distances (assumed to be in miles) appear only once; in other words the file gives the distance from *city1* to *city2* but not from *city2* to *city1*. Your program will have to account for this by storing the data twice, once for the road as given and once for the reverse direction.

2. Two cities that are not connected by a road are assumed to be infinite distance apart. Use whatever large integer constant you think adequate.

3. To get around problems with reading spaces while still having something resembling a nice format, we use the underscore character for a space in city names.

After reading the data file and computing the path lengths, prompt the user for two cities to travel between. I *strongly* suggest simplifying the interface by numbering the cities on the screen in menu form and accepting city numbers as input. Remember that such a menu will change when the `nqmq.dat` file changes. Output consists of the path between the two cities and the distance associated with that path. It's probably simplest to just list the cities in the order visited and then the distance. If done this way the output screen could look like the one at right. Note that this is only an example of how you could represent the output. You may organize the screen in any manner you wish that is clear.

```
Atlanta_GA to Boston_MA

Path:
Atlanta_GA
Chicago_IL
Boston_MA

Distance: 1466 miles
```

As discussed in discrete math, since this graph has no loops or multiple edges, it may be (and *should* be) represented as an adjacency matrix. Further information can be found in most discrete math or data structures texts. Finally there are a number of ways to search for paths. Our method of choice is Dijkstra's algorithm. Remember that, in addition to finding the length of the shortest path, you must be able to list the cities along this path in order so that you can communicate this information to the user. The best options appear to be either building and marking the path as you go, or marking the path and then retracing it to print out at the end. Any way you can make this work is okay.

Files

You are supplied with these completed files:

- `makefile` - use `make` on the linux system to compile the project using this makefile. The binary file will be named `digraph.out`.
- `Digraph.hpp` - This contains the header file for a directed graph class using an adjacency matrix.
- `test_dijkstra.cpp` – This file contains all of the test cases and will be what you must compile and run to test your code.
- `nqmq.dat` and `nqmqBig.dat` – Data files with city names and road distances between cities.

You must complete the project by completing/writing the following:

- `Digraph.cpp` - There are seven functions to be implemented in this file.

General Information

- **Make any changes to the `.cpp` files and not the header files.**
- You ARE allowed to use `std::vector` and `std::map`. You are **NOT** allowed to use `std::priority_queue` or other structures from the STL library.

Rubric

- **[30%]** Your program must compile and run on our knuth linux server
- **[50%]** Your program must pass all test cases
- **[10%]** You must document all functions in your code (within reason)
- **[10%]** You must properly handle memory, memory leaks will cost you points

Submission Instructions

When done, zip all relevant files (`makefile`, `Digraph.hpp`, `nqmq.dat`, `test_dijkstra.cpp`, your newly-constructed `Digraph.cpp` plus any additional stack/list/heap classes you used) into one archive and submit it to Brightspace.

Last updated 11.22.2017 by T. O'Neil.