Data Structures and Algorithms II
Fall 2015
Programming Assignment #3

You are going implement Dijkstra's algorithm to solve the single-source shortest-path problem. The program will determine the shortest path in a specified graph from a specified starting vertex to each other vertex in the graph. In order to do this efficiently, your program should use the binary heap class that you created for the previous assignment.

Your program should start by asking the user to enter the name of a file specifying the graph. Every row in the input file represents an edge in the graph. Each row consists of two string ids representing the source vertex and destination vertex of the edge (in that order) followed by an integer representing the cost (a.k.a. distance or weight) of the edge. The rows will contain no leading or trailing whitespace, single spaces will separate fields, and all rows will end with a single Unix-style newline character. All vertex ids will consist only of lowercase and capital letters and digits. All edge costs will be positive integers less than one million. A vertex exists if it is the source or the destination of any edge. The source vertex of an edge will never be the same as the destination vertex, but it is possible that multiple edges might connect the same vertices. Your program may assume that the file, if it can be opened, is valid. You are not required to include error checks for invalid file formats; you may if you wish, but I will not check for this.

Once the program is finished reading in the graph, the user should be prompted to enter the id of a starting vertex. The user should be re-prompted until they enter a valid index (i.e., a string id representing a vertex that exists in the graph). The program should then apply Dijkstra's algorithm to determine the shortest path to each node from the specified starting vertex. The implementation should rely on the binary heap class that you created for the previous assignment. (The heap class, of course, relies on the hash class you created for the first assignment, and you will also likely rely on the hash class for a couple of other purposes as well.) When the algorithm has finished determining the shortest path to each node, your program should output the CPU time, in seconds, that was spent executing the algorithm.

The program should then ask the user for the name of an output file. The output file should contain one row for every vertex that exists in the graph, with vertices listed in the same order that they first appear in the input file. Each row in the output file should contain a vertex id followed by a colon, a single space, and then the shortest distance from the specified starting vertex to the given vertex. All of these distances are guaranteed to be less than one billion. After the distance, the row should contain one space, a left bracket, the path from the starting vertex to the current vertex, a right bracket, and finally a single Unix-style newline character. Vertices in the path should be separated by a comma followed by a single space. There should not be any space or comma before the first vertex in the path (the specified starting vertex) or after the last vertex in the path. If there is no path from the specified starting vertex to any existing vertex in the graph, the corresponding output row should contain the vertex id followed by a colon, a single space, and then the text "NO PATH" followed by a single Unix-style newline character. You must follow these instructions *exactly*.

In class, we stepped through Dijkstra's algorithm for the following graph, which came from Figure 9.20 in the textbook:

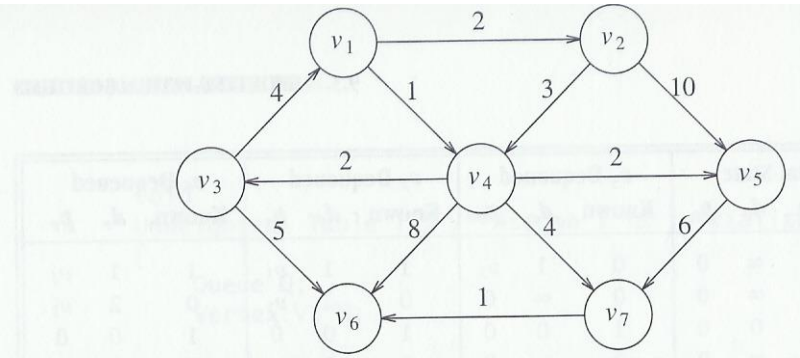

Figure 9.20 The directed graph G (again)

The file representing this graph might look like this:

```
v1 v2 2
v1 v4 1
v2 v4 3
v2 v5 10
v3 v1 4
v3 v6 5
v4 v3 2
v4 v5 2
v4 v6 8
v4 v7 4
v5 v7 6
v7 v6 1
```

Any permutation of the rows representing edges in this file would designate the same graph (but the order of rows in the output file might be different). Assume a file called `graph.txt` exists, containing the data shown above. Then a sample run of your program might look like this:

```
Enter name of graph file: graph.txt
Enter a valid vertex id for the staring vertex: v1
Total time (in seconds) to apply Dijkstra's algorithm: 0.000
Enter name of output file: out.txt
```

The prompts to the user may vary, but the file `out.txt` should look *exactly* like this:

```
v1: 0 [v1]
v2: 2 [v1, v2]
v4: 1 [v1, v4]
v5: 3 [v1, v4, v5]
v3: 3 [v1, v4, v3]
v6: 6 [v1, v4, v7, v6]
v7: 5 [v1, v4, v7]
```

If the user specifies the same graph file but enters v5 as the id of the starting vertex, then the output file should look exactly like this:

```
v1: NO PATH
v2: NO PATH
v4: NO PATH
v5: 0 [v5]
v3: NO PATH
v6: 7 [v5, v7, v6]
v7: 6 [v5, v7]
```

As already stated, you should rely on your heap implementation (which in turn relies on your hash table implementation) to implement Dijkstra's algorithm efficiently. If you have implemented these classes correctly and completely, you should not need to modify any of your heap or hash files for this assignment. You should also create a graph class that is constructed with Dijkstra's algorithm in mind, so the implementation of the algorithm can be handled by a member function of this class. I suggest including a private nested class to store nodes in the graph. The graph can also contain a linked list of pointers to nodes. Whenever a new node is encountered, you can allocate memory for the node and add a pointer to the new node to the end of the linked list. (Alternatively, you may decide to use a linked list of nodes directly, instead of a linked list of pointers. Either way, you may use the provided C++ list class for this purpose.) One field of each node must store an adjacency list for the node. This can also use the provided linked list class. Each node in an adjacency list represents an edge, and each edge must at least specify the destination vertex and the cost of the edge. (You do not necessarily have to specify the source node in each edge, since it is the same for every node in an adjacency list.)

As you are reading the edges of the graph from the input file, you will need some way to efficiently determine whether or not you have encountered each vertex id already. If not, you need to create a new vertex node. If the source vertex of the edge has been previously encountered, you have to locate the corresponding node efficiently in order to update its adjacency list. I suggest using your own hash table implementation for these tasks. Whenever a new vertex is encountered, add an entry with the new vertex id to the hash table, and use the void pointer to point to the new node. To locate a node corresponding to a source vertex, use the getPointer member function of the hash table class. I also suggest using your hash table class to determine whether or not a starting vertex entered by the user is valid.

When you are implementing Dijkstra's algorithm, I suggest using the void pointer of each heap node to point to the node corresponding to each vertex. You can then use the optional parameter of deleteMin to obtain this pointer and access the node immediately. Although you could also locate the node by just obtaining the vertex id from deleteMin and then using your hash table to obtain the pointer to the node, this is a bit less efficient, and I consider this less elegant, so I may take off a few points for this solution.

After you have completed the assignment, e-mail me (*CarlSable.Cooper@gmail.com*) all of your code, including a `Makefile`. I should be able to run "make" and then test your executable. The program is due before midnight on the night of Tuesday, November 24.