CS241
Dijkstra's Shortest Path Algorithm Project

Overview

In this project, you will implement a graph ADT as well as Dijkstra's algorithm, commonly used for finding the shortest path in a graph between two vertices. Your program will read in a text file containing a description of a graph, run Dijkstra's algorithm, and print the final shortest path at the end, along with debugging information that will verify your algorithm is working correctly.

Part A: Graph ADT Implementation

The graph data structure you will implement will be a directed graph, where vertices are represented by ***strings*** and edges have integer weights. These weights may be used to represent costs, distances, or times (the meaning is immaterial for implementing the data structure). You will use two C++ built-in structures to implement the graph. First, you will use a `set<string>` to store the collection of vertices. Internally, the build-in C++ set data type uses binary search trees, though you (and your code) don't need to worry about that.

More information about the set structure is here: http://en.cppreference.com/w/cpp/container/set

Edges in the graph will be stored using an adjacency list. Recall that an adjacency list stores all the edges (and their weights) that emanate from a vertex as a linked list. To enable fast access to the linked list for each vertex, we will use a map that associates strings (representing vertices) with these linked lists. The complete data type for edges is: `map<string, list<pair<string, int>>>`. Let's break that down from the inside out:

The inner-most part is `pair<string, int>`. These pairs represent edges; or at least, the ending vertex of each edge [the `string` part] and its associated weight [the `int` part].

Moving outward, we have a `list<pair<string, int>>`. Think of this as a list of edges. So each of these lists is an adjacency list.

At the outermost level, we have `map<string, list<pair<string, int>>`. This represents the entire adjacency list structure: a mapping between vertices (strings), and their corresponding adjacency lists (`list<pair...>>`).

How to use C++ maps:

C++ maps are incredibly useful; they maintain an association between any data type (the key type) and any other data type (the value type). Maps enable very fast lookup by key. C++ maps use BSTs behind the scenes, so almost all operations are logarithmic time. Note that you don't have to write any BST code to use these maps; all that is handled behind the scenes.

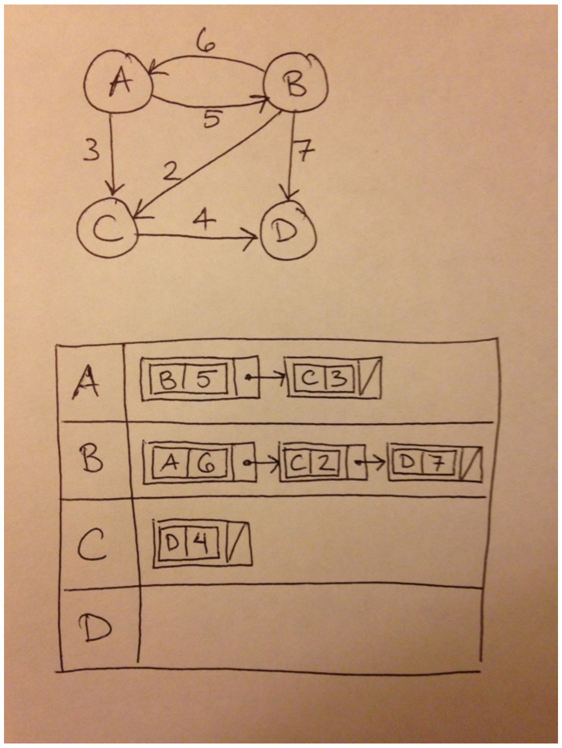
See the documentation here: http://en.cppreference.com/w/cpp/container/map

Here's a quick example:

```
map<string, int> ages;          // Make a map associating people's names (strings)
                                // with their ages (ints).
ages["Alice"] = 20;             // Alice is 20 years old (this inserts Alice: 20) into the map.
ages.insert( {"Bob", 19} );     // Another way to add a new item into the map
                                // (same effect as ages["Bob"] = 19

ages["Alice"] = 21;             // Increase Alice's age (this is how you modify a map entry).

cout << ages.at("Alice");       // Safest way to retrieve an entry from the map, though you can
                                // also do cout << ages["Alice"]; in most situations.

edges.count("Alice")            // returns 1 if Alice is in the map at all, 0 otherwise
                                // useful to check if an entry is in the map or not.

edges.size()                    // returns the number of entries in the map.
```

```
for (pair<string, int> p : ages) {
  cout << p.first << p.second;         // iterate over the entries in a map:
                                       // p.first is the name, p.second is the age.
}
```

So you will be using a `map<string, list<pair<string, int>>>`. Here's an example of how that might look:



Notice the triply-nested structure of the map, which stores lists, and each list node is a pair of a `string` and an `int`.

What you need to do:

- Implement all methods in the `Graph` class. See `graph.h` and `graph.cpp` for what they should do. Because you are using sets and maps that are built into C++, these methods are very short (some of them are one line of code).

- Take a look at `graph1.txt` through `graph4.txt` to examine how the file structure works. There are four possible commands you'll see in these files:

  The first line of the file will always be the word "`directed`" or "`undirected`" which specifies the type of graph. To make life easy, I suggest that your Graph class represent adjacency lists as directed edges all the time, and for undirected graphs, just store each edge twice (once in each direction, with the same weight).

  `vertex name`: this command should add a vertex called "`name`" to the graph. You may assume "`name`" is a string that hasn't already been used as the name of a vertex.

  `edge name1 name2 weight`: this command should add an edge between the vertices `name1` and `name2` with the given integer weight. You may assume `name1` and `name2` have already been added to the graph, and there is not an existing edge between them already.

  `dijkstra name1 name2`: this command will always be the last one in the file. When your program sees this command, first call `print()` on your graph to print out the entire state of the graph. Then call your Dijkstra function (see part B below) to run Dijkstra's algorithm to find the shortest path from vertex `name1` to vertex `name2`.

- Write some tests in `main.cpp` to test that your Graph class is working.

Part B: Dijkstra's algorithm implementation

At this point, the only command you haven't implemented yet is the one that runs Dijkstra's algorithm. Write a function to do this. (The code from class is included at the end of this document.)

Your Dijkstra function MUST do the following (refer to the sample output for how it should look):

- Print out the name of each vertex as it is removed from the priority queue (your order should match mine).
- Print out every update to the `dist[]` map that is made (again, see my output for how this should look).
- After the algorithm finishes, print the final path (in the correct order from start node to finish node), and the total path distance.
- Print the final state of the `dist[]` and `prev[]` tables.

- Your output should match mine. The only exception is when visiting a node, you may look at its neighbors in any order (you don't have to do it alphabetically). Also, for the priority queue, if there are ties in distances, let the priority queue break them for you --- in other words, just call `extractMin()` and let it handle ties for you.

Hints and suggestions for Dijkstra:

- Use a `map<string, int>` for `dist`.
- Use a `map<string, string>` for `prev`.
- Use the `PriorityQueue` class provided. (It should be easy to figure out the code, which you don't need to modify.)

Dijkstra's Algorithm

```
void dijkstra(Graph g, Vertex start, Vertex finish)
{
        dist[start] = 0

        create min-priority queue Q

        for each vertex v in the graph:
                if v != start:
                        dist[v] = infinity
                        prev[v] = undefined

                Q.add(v, dist[v])

        while Q is not empty:
                u = Q.extract_minimum()

                if u == finish: break

                for each neighbor v of u:
                        alt = dist[u] + length(u, v)
                        if alt < dist[v]
                                dist[v] = alt
                                prev[v] = u
                                Q.decrease_priority(v, alt)

        Traverse prev[] array starting from prev[finish] in reverse order back
        to start vertex to get final shortest path.
}
```