CS 130: Algorithms and Data Structures I
Fall 2019
Professor Reeves
Assignment 4: MyTunes Artist Recommendations
Released: 12/09/2019
Due: 12/19/2019, 11:59 PM

1 Introduction

Now that you have built an artist search feature for your MyTunes service. You would like to help users discover new artists to whom they may not have been exposed to previously. You realize that one means by which artists show interest in other artists is through collaborations. As part of a recommendation system for new artists of interest, you would like to incorporate collaboration data. In particular, collaborations will be represented as weighted edges in a graph with artists being represented as nodes. Recommendations will be built by calculating the shortest paths between a particular artists and all other artists in this artist graph.

2 Instructions

The high-level goal of this project is to construct a graph using the adjacency list representation of a graph as discussed in class. The graph is defined using the following data structures.

```
typedef struct NODE *LIST;
struct NODE {
    int key; // index of node in GRAPH
    double weight; // edge weight from source node
    LIST next = nullptr; // next pointer in predecessor list
};

/* A node in the collaboration graph */

struct GRAPH_NODE {
    std::string artist_name = "";
    double dist;
    LIST successors = nullptr; //adjacency list
    int pred_ptr; // points to previous node in shortest path (index in G)
    int to_pq; // points at corresponding priority queue node
};
```

Edge weights between artists are stored in each NODE. Each artist (a node in the graph) maintains a list of artists with whom the artist has collaborated (successors). Each of the successors is a linked list. The edge weight stored in

each NODE in the successors list is the inverse of the number of times the artists have collaborated, $\frac{1}{n}$, where n is the number of collaborations. This means that artists whom have collaborated more will have smaller edge weights. A smaller edge weight means that the artists are closer together on the graph.

Each GRAPH_NODE will hold an artist's name, a distance (the shortest distance from a source NODE in the graph), a list of successors (for representing the adjacent NODEs), a predecessor pointer (an integer) which references the index of a NODE in the graph, and a pointer (using an integer) to the node's location (index) in a priority queue.

This priority queue will be used to implement *Dijkstra*'s algorithm. Recall that *Dijkstra*'s algorithm is used to efficiently calculate the shortest distance between a source node and each destination node that is reachable through a path in the graph. The priority queue is defined by the data structures below:

```
/* A heap structure used to maintain a min-heap for Dijkstra's algorithm */
struct HEAP{
    int arr[MAX + 1]; // values represent node index in graph
    int length = MAX;
};

/* A pointer to a heap structure used for min-heap */
typedef struct HEAP* PRIORITY_QUEUE;
```

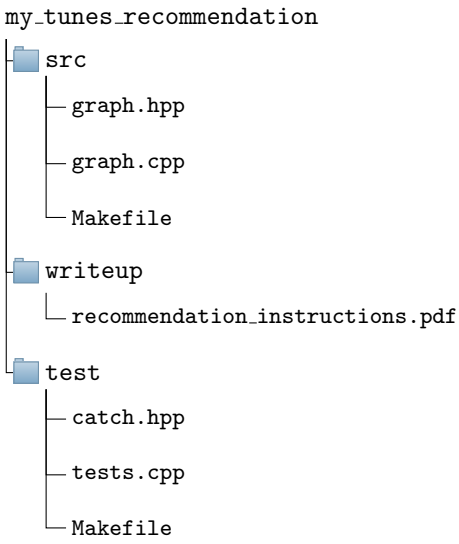
This priority queue is a pointer to a min-heap. The heap holds an integer array and a length indicating the number of nodes *currently* in the priority queue. This priority queue along with the graph will be used for the implementation of *Dijkstra*'s algorithm. These structures will be used calculate a type of strength of the connection between nodes (artists) in the graph. While you are not responsible for implementing this functionality, the MyTunes software will use this code to recommend connected artists for an artist in which a user has shown interest (by following the artist or listening to the artist's music within MyTunes). One recommendation algorithm might display to a user all artists that have a shortest path within a value of X of the source artist.

You are responsible for implementing the following functions for this program. The following is just a list of the functions. Detailed descriptions of what each function should do are specified in the *graph.hpp* file.

Swap
Priority
MoveUp
MoveDown
Initialize
Dijkstra
ReleaseMemory (an overloaded function)
textitBuildGraph

3 Files

You are being provided with a set of files for this assignment that are available on Blackboard. The files are located in a ZIP file named *my_tunes_recommendation.zip*. The directory structure provided when you uncompress *my_tunes_recommendation.zip* is as follows:



Starter files for your implementation have been provided for you. These files are *graph.hpp* and *graph.cpp*. Implement your solution in *graph.cpp*. Th-GRAPH_NODEe *test* directory stores files for testing that your code works as expected. You should make sure that all tests are passed before you submit your assignment. More details on testing are located in the *Testing* section of this document.

4 Testing

Your implementation is being tested using a unit-testing framework called Catch2. The functionality of this library is provided in the file *test/catch.hpp*. Feel free to view this file but **do not** modify it. A Makefile has been provided for you in the *test* directory. To build the tests, type *make* inside the *test* directory from your computer's command line. To run, the tests after they have been successfully built, type *./runtests* in the *test* directory from your computers command line. You can build and run your tests with a single line by typing *make && ./runtests*. When you initially run your tests (before implementing your solution), you will see the following at the end of the output from the *./runtests* command:

```
===============================================================
test cases: 10 | 2 passed | 8 failed
assertions: 11 | 3 passed | 8 failed
```

After successfully implementing your solutions, you will see a message such as the following in the console after executing the *./runtests* command:

```
===============================================================================
All tests passed (11 assertions in 5 test cases)
```

You may need to install the GNU g++ and GNU make on your computers in order to properly compile your code and run the tests. Instructions for downloading and installing these programs are included below for Windows:

4.1 g++

`http://www.codebind.com/cprogramming/install-mingw-windows-10-gcc/`

4.2 Make

`http://gnuwin32.sourceforge.net/packages/make.htm` (Windows)

Things are a bit simpler for Mac OS X users. You need to install Xcode and the command line tools (both are free). Instructions can be found here:

`https://www.embarcadero.com/starthere/xe5/mobdevsetup/ios/en/installing_the_commandline_tools.html`

If you are having trouble setting up these programs, please come talk to me about it as soon as possible.

5 Evaluation

The only file that you need to submit is your version of *graph.cpp* on Blackboard for this assignment. If your code passes all of the tests for this assignment **and** satisfies the requirements for the functionality of each function above, you will receive all available points.