CS 15 Project 2: Six Degrees

Six degrees of what?

Ever heard of the game Six Degrees of Kevin Bacon? (Play the game online; it's fun and counts as serious academic work.) Your program will play the game, too, except instead of actors we will use musical artists and songs that they have collaborated on with other artists!

Your program will interactively accept queries in the form of an input command and the names of artists. It will then output a path to get from the first artist to the second via musical collaborations (with some restrictions depending on the input command).

```
dfs
Stefflon Don
SZA
"Stefflon Don" collaborated with "French Montana" in "Don't Sleep".
"French Montana" collaborated with "will.i.am" in "Feelin' Myself".
"will.i.am" collaborated with "Usher" in "OMG".
"Usher" collaborated with "Pitbull" in "DJ Got Us Fallin'" in "Love".
"Pitbull" collaborated with "Shakira" in "Rabiosa".
"Shakira" collaborated with "Rihanna" in "Can't Remember to Forget You".
"Rihanna" collaborated with "SZA" in "Consideration".
***
```

NOTE: Three asterisks followed by a new line denote the end of a path.

Getting Started

You can get the starter files from

```
/comp/15m1/files/proj2
```

Note that the reference implemetation files are in the `binary/` folder, so you will want to make a recursive copy with `cp -r binary/ desination_path`. Use the reference implementation compiled for your system.

Useful Advice

We have compiled a short list of things you can do to maximize your success with this project:

- Read the entire assignment specification thoroughly.
- Carefully review all of the documentation within the provided code (see the Provided Code section).
- Understand the purpose of every public function for the `CollabGraph` class and the `Artist` class.
- If your program is not working as intended, execute your program with valgrind. This will help you identify the source of your errors.
- If your program is working, execute your program with valgrind. This will help you identify errors you may have missed.
- Test your implementation with small datasets of your own design.
- Create datasets that test edge cases.
- Draw pictures of graphs if you are stuck.

Program Details

You will write a program called `SixDegrees` that accepts either 1, 2, or 3 command-line arguments (in addition to the program name), like this:

```
./SixDegrees dataFile [commandsFile] [outputFile]
```

Where:

- `dataFile` is an input file containing information about artists. You will populate your graph with the information stored in this file. See the **Populating the CollabGraph** section for an explanation of `dataFile` format.
- `commandsFile` (optional) is a second input file containing commands to execute. If this parameter is not given, then you must read input from `std::cin`.

- `outputFile` (optional) is an output file. If provided, `SixDegrees` will send search results to this file. If this parameter is not given, then you must send output to the standard output stream (`std::cout`)

Note: it's a common convention to denote optional parameters by putting them in square brackets.

Input and Output Behavior

The input and output behavior of the `SixDegrees` program is determined by the number of command line arguments. The `SixDegrees` program must receive either 1, 2, or 3 command line arguments.

- `./SixDegrees artists.txt`
  - `SixDegrees` will populate the graph with the information stored in the provided `artists.txt`
  - `SixDegrees` will read commands from the standard input stream (`std::cin`)
  - `SixDegrees` will send output to the standard output stream (`std::cout`)
- `./SixDegrees artists.txt commands.txt`
  - `SixDegrees` will populate the graph with the information stored in the provided `artists.txt`
  - `SixDegrees` will read commands from the provided `commands.txt`)
  - `SixDegrees` will send output to the standard output stream (`std::cout`)
- `./SixDegrees artists.txt commands.txt output.txt`
  - `SixDegrees` will populate the graph with the information stored in the provided `artists.txt`
  - `SixDegrees` will read commands from the provided `commands.txt`
  - `SixDegrees` will send output to provided `output.txt`

  The files can have any names; these are just examples.

If number of commands line arguments is not 1, 2, or 3, then the following must be sent to the standard error stream (`std::cerr`)

```
Usage: ./SixDegrees dataFile [commandFile] [outputFile]
```

When a file cannot be opened, the following must be sent to the standard error stream (`std::cerr`)

```
filename cannot be opened.
```

Where *filename* is replaced with the name of the file which could not be opened.

Artist Class

The SixDegrees program uses the `Artist` class to represent artists and their discography. The full `Artist` implementation is provided (see the Provided Code section). **Do not modify the contents of `Artist.h` or `Artist.cpp`; there are no exceptions to this rule**. When you submit your finished project, we will compile your code with our original `Artist` implementation.

CollabGraph Class

The SixDegrees program relies on an undirected, unweighted graph whose vertices are `Artist` objects and whose edges are labeled with strings. A partial `CollabGraph` implementation is provided (see the Provided Code section). **Do not modify the provided contents of `CollabGraph.h` or `CollabGraph.cpp`; you are only to implement the functions indicated below with the title "To implement"**.

To Implement: report_path

The `CollabGraph` interface contains a `report_path` function which you are to implement. It returns a `std::stack` containing a path from the provided `source Artist` to the provided `dest Artist`. Invoking `report_path` will do one of two things:

- Return an empty `std::stack`, which indicates that there **is not** a valid path between the provided `source` and `dest Artist`s.
- Return a non-empty `std::stack` with at least two elements, which indicates that there is a valid path between the provided `source` and `dest Artist`s. The topmost element is the `source Artist`, and the bottommost element is the `dest Artist`.

Note: The `report_path` function will only behave if your traversal algorithm correctly set the predecessor of each vertex between `source` and `dest`.

Moreover, the traversal algorithms that you implement in your `SixDegrees` class will only behave if you clear the modified metadata resulting from a previous traversal of the `CollabGraph`. Thus, it is vital that you thoroughly unit test each function you implement.

Assuming that your traversal algorithms are well-behaved, you now have a `std::stack` that represents the path from `source` to `dest`. It will be your responsibilty to implement a function in your `SixDegrees` class that takes this `std::stack` and sends a properly formatted summary of the path to the output stream (see the **Path Format** section).

To Implement: get_vertex_neighbors

The `CollabGraph` interface also contains a `get_vertex_neighbors` function for you to implement. It takes in an `Artist` and returns a vector of all `Artist`s connected to the input by an edge. If the `Artist` does not have neighbors, it will return an empty vector. This function will be vital for implementing the traversal algorithms later on.

Using enforce_valid_vertex

As you implement these above functions, you may note the edge case where a provided artist is not a vertex in the graph. **You are to utilize the `enforce_valid_vertex` helper function to address these cases**.

Populating the CollabGraph

The `CollabGraph` will be populated with information stored in a `dataFile`. A `dataFile` contains a sequence of artist entries, where each entry must follow these formatting rules:

- the name of an artist, terminated by a newline
- a list of songs in which that artist collaborates, where each song title is terminated by a newline.
- A sentinel line, which contains a `*` and a newline character

See the given artists file for a detailed example.
Note: the `CollabGraph` class will throw an exception if you violate any of its representation invariants when you populate your collaboration graph. See the **Runtime Exceptions** section for additional information.

Runtime Exceptions

The `CollabGraph` class will throw a `std::runtime_error` if you attempt to use a vertex that has not been inserted into the collaboration graph.

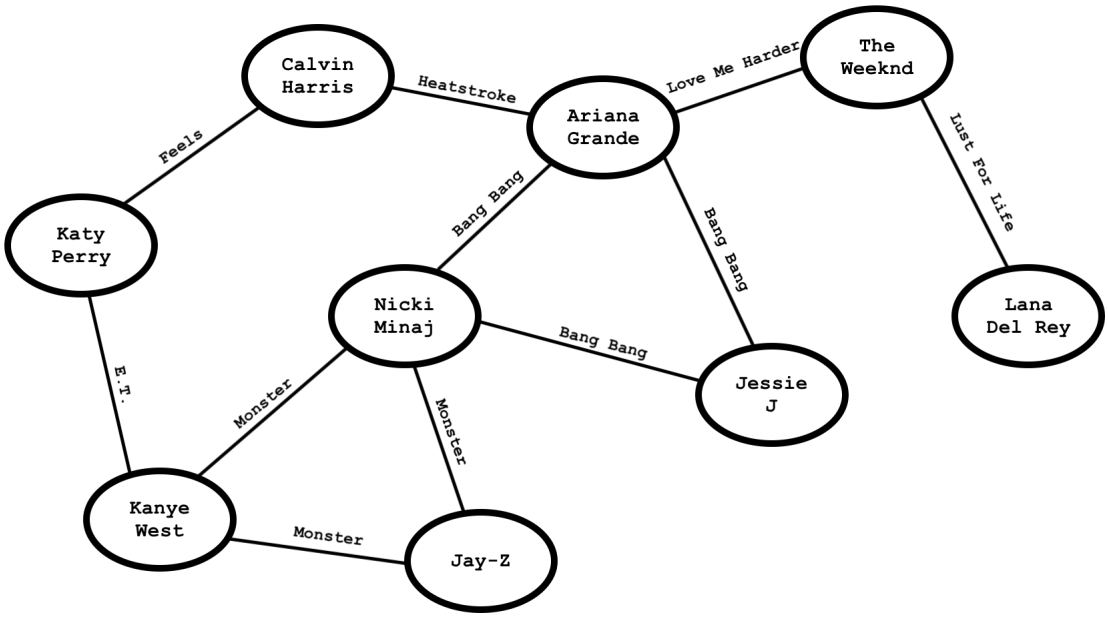
To help you correctly populate your collaboration graph, the `CollabGraph` class will throw a `std::runtime_error` if you violate any of its representation invariants while populating the graph. The `CollabGraph` class will throw a `std::runtime_error` if you do any of the following:

- attempt to insert an artist/vertex that is improperly initialized. A properly initialized artist must have a name that is not the empty string. attempt to insert an edge named "". The empty string is **not** a valid edge name.
- attempt to insert an edge between a vertex and itself. Looped edges are forbidden.
- **DO NOT** catch a `std::runtime_error` thrown by the `CollabGraph`. These exceptions exist to help you identify faults in your implementation. If your implementation prompts the `CollabGraph` to throw an exception, you have violated an invariant of the `CollabGraph` class and you must rethink your solution.

As always, `valgrind` is an invaluable tool. Don't forget to use it.

Data Representation

In a `CollabGraph`, the name of the edge between two artists is the name of a song in which they collaborated.



Every artist from the input file `dataFile` becomes a vertex of the graph. If two artists have collaborated on a song together, they are connected by an edge whose value is the name of the song. There is an edge between two artists if they've performed together in at least one song. If the artists have appeared in multiple songs together, don't worry about it! The `CollabGraph` implementation will handle which edge is used automatically.

> The predecessor of a vertex in the `CollabGraph` can only be set once during a single traversal. If you attempt to set the predecessor for a vertex that already has one, the `set_predecessor` function will do nothing.

Six Degrees Class

You are required to design a `SixDegrees` class that implements the following functionalities:

- Populate a `CollabGraph`, given a file containing a list of artists and songs

- Execute a sequence of commands, and output the results
    - Your implementation must be capable of handling input and output as specified in the **Input and Output Behavior** section.
    - If an unknown command is provided, then an appropriate message must be sent to the specified output stream (see the **Printing** section) and the command loop must continue to run. The program must not crash.
    - You may need to use auxiliary data structures in your implementation. You are permitted to use the `std::stack`, `std::queue`, and `std::vector` from the Standard Template Library.
    - Choose your auxiliary data structures deliberately. If it makes more sense to use a `std::queue` or `std::stack`, don't use a `std::vector`.

The interface for your `SixDegrees` class **must** be defined in `SixDegrees.h`. The implementation for your `SixDegrees` class **must** be defined in `SixDegrees.cpp`.

main.cpp

You will implement a `main.cpp` that is responsible for handling command line arguments and interacting with the rest of your implementation. Your main.cpp must implement all behavior specified in the **Input and Output Behavior** section and communicate with your `SixDegrees` class.

The Commands

You must implement four commands, which are specified below. You may optionally implement a fifth command, but this is left as a JFFE. All commands are entered in the following format:

```
bfs
Meghan Trainor
Megan Thee Stallion
```

Each line is read in its entirety. First, a command is entered. Each argument to that command is provided **on a new line**. Only one command/argument may be provided per line. The following is an example of an *improperly* formatted command:

```
bfs Meghan Trainor Megan Thee Stallion
```

The same format is used for commands from both the standard input stream (`std::cin`) AND from files. (Hint: this is a good opportunity to implement a modular solution that can read input from any input source).

You will implement the following commands:

Breadth First Search (bfs)

```
bfs
artistA
artistB
```

- Takes in two artists (each on a separate line) and prints the **shortest** path from `artistA` to `artistB` found with a breadth first search. There may be multiple shortest paths, so printing any one of them is okay.
- If a provided artist is not found in the graph, then an appropriate message (see the **Printing** section) is sent to the output stream and the command loop will continue running. The program must not terminate.
- If a path does not exist between `artistA` and `artistB`, then an appropriate message (see the **Printing** section) is sent to the output stream and the command loop will continue running.

Depth First Search (dfs)

```
bfs
artistA
artistB
```

- Takes in two artists (each on a separate line) and prints **any** path from `artistA` to `artistB` found with a depth first search. Be careful about looping back to a vertex that has already been visited!
- If either or both artists are not found in the graph, then an appropriate message (see the **Handling Unknown Artists** section) is sent to the output stream for each unknown artist and the command loop will continue running.
- If a path does not exist between `artistA` and `artistB`, then an appropriate message (see the **Printing** section) is sent to the output stream and the command loop will continue running.

Exclusive Search (not)

```
not
artistA
artistB
[artistC]
[artistD]
...
*
```

- Takes in two artists (each on a line by itself) and also a list of artists to exclude (each on a line by itself) and prints out the shortest path from `artistA` to `artistB`, excluding all artists from the list of artists to exclude.
- Note: Exclusion means that the printed path should **not** include that `Artist`.
- The list of `Artist`s to exclude can be empty. This just finds the shortest path from A to B with no exclusions.
- If any of the provided artists are not found in the graph, then an appropriate message (see the **Handling Unknown Artists** section) is sent to the output stream for each unknown artist.

  Note: artists will be read from the input stream until an asterisk is received, even if an unknown artist was already encountered.
- If a path does not exist between `artistA` and `artistB`, then an appropriate message (see the **Printing** section) is sent to the output stream and the command loop will continue running.
- NOTE: The asterisk denotes the end of the not command.

Terminate program (quit)

```
quit
```

- Terminates the program and prints nothing.
- Don't forget to free all heap allocated memory!

JFFE: Inclusive Search (incl)

```
incl
ArtistA
ArtistB
ArtistC
```

- Takes in three artists (each on a line by itself) and prints any path from `artistA` to `artistB` that passes through `artistC`.
- If any of the provided artists are not found in the graph, then an appropriate message (see the **Handling Unknown Artists** section) is sent to the output stream for each unknown artist.
- The following cases are important to consider:
  - When finding a path from from `artistA` to `artistB`, the path between `artistA` and `artistC` must **not** contain artists from `artistC` to `artistB` and vise versa. That is, the path from `artistA` to `artistB` must not contain duplicate artists.
  - If there is a path from `artistA` to `artistC` which passes through some `artistX`, but all paths from `artistC` to `artistB` also pass through `artistX`, what should your algorithm do next? Give up? Or, try to find another path from `artistA` to `artistC`?
  - What if the path from `artistA` to `artistB` does not contain a path from from `artistA` to `artistC` or from `artistC` to `artistB`?

  We have come up with two distinct solutions for this problem. Any functional solution that discovers a valid path in a reasonable amount of time is acceptable.

Printing

All output goes to the designated output stream, as specified in the **Input and Output Behavior** section. Results from your `SixDegrees` class are never sent to the standard error stream (`std::cerr`).
You can send logging and debug output to `std::cerr` while your are developing your program, but you **must** remove all extraneous instances of `std::cerr` before you submit.

Path Format

If a path is found between two artists, you must print each intermediate connection. Consider this general example, where there is a path from `artistA` to `artistD`, through `artistB` and `artistC`:

```
"[Artist A]" collaborated with "[Artist B]" in "[songName]".
"[Artist B]" collaborated with "[Artist C]" in "[songName]".
"[Artist C]" collaborated with "[Artist D]" in "[songName]".
```

Example:

```
"Zedd" collaborated with "Ariana Grande" in "Break Free".
"Ariana Grande" collaborated with "Kehlani" in "The Way".
"Kehlani" collaborated with "ZAYN" in "wRoNg".
```

If a path is not found between two artists, you must print an appropriate message. Consider this general example, where there is no path from `artistA` to `artistB`:

```
A path does not exist between "[artistA]" and "[artistB]".
```

Example:

```
 A path does not exist between "Kesha" and "ZAYN".
```

Handling Unknown Artists

If a provided artist is not present in the graph, then you must print an appropriate message. Consider this general example, where `artistA` is not present in the graph:

```
"[Artist A]" was not found in the dataset :(
```

Example:

```
"Nickelback" was not found in the dataset :(
```

There is no space between the colon and the parenthesis of the frowny face

Handling Invalid Commands

If an invalid command is entered, your `SixDegrees` program will send the following message to the output stream:

```
[invalid] is not a command. Please try again.
```

Where [invalid] is the invalid command. For example:

```
ILoveMyTAs is not a command. Please try again.
```

Provided Code

The purpose of this assignment (NOTE: not the purpose of the program!) is to introduce you to designing algorithms and utilizing provided data structures, which are skills you will build upon in 160 and 40 (should you choose to take them). Thus, we will be providing you with the following:

- A partially implemented `CollabGraph` class (both the `CollabGraph.h` and the `CollabGraph.cpp`). You will implement two functions, `report_path` and `get_vertex_neighbors`, adding private helper functions if you wish. Do not edit the signatures of any existing `CollabGraph` functions.

  `CollabGraph.cpp` contains extensive documentation and detailed function contracts. You are not required to fully understand the implementation, but we recommend that you review the documentation to better understand how to use the `CollabGraph` class.

- A fully implemented `Artist` class (both the `Artist.h` and the `Artist.cpp`)

  `Artist.cpp` contains extensive documentation and detailed function contracts. You are not required to understand the implementation, but we recommend that you review the documentation to better understand how to use the `Artist` class.

- An example `dataFile` called `artists.txt` which contains a dataset with which to populate the graph.

  The format of this file is described in the **Populating the CollabGraph** section.

- An executable for the reference implementation, `./the_SixDegrees`

- A `Makefile` that you will need to modify to include any aditional files needed for the program to run.

  Your `SixDegrees` program **must** build when we run make `SixDegrees` and produce an executable named `SixDegrees` which can be run by typing `./SixDegrees`

Your Assignment

Phase 0: Design Checkoff

You **must** upload physical drawings and pseudocode to your Design Checkoff to the Gradescope. While pseudocode is required, TAs will not look at any

implemented code before you complete Phase 0.

You should upload the following items::

- A description of your approach for each of the four required commands. In particular, you should consider if you'll use/modify DFS and/or BFS for your implementation of the `not` command.
- A short description of how the graph is represented in the `CollabGraph` class.
- A drawing of a small graph with sample input and output, including edge cases.
- A list of public and private functions included in your `SixDegrees` class, along with short descriptions of each.
- A plan of what you will do each day, accounting for unforeseeable delays.

Phase 1: Getting Acquainted With the Graph

This phase will help you dive deeper into the graph's representation. You will be in charge of implementing the following three components:

1. `CollabGraph::report_path`

   Implement `CollabGraph::report_path` as described in its respective section.

2. `CollabGraph::get_vertex_neighbors`

   Implement `CollabGraph::get_vertex_neighbors` as described in its respective section.

3. Populate the `CollabGraph`

   You will write code that reads in artist information from a dataFile formatted as described in the **Populating the CollabGraph** section. This code must also insert a vertex for each artist and insert edges between artists that have collaborated.

To complete Phase 1, you will have to implement `main.cpp` (see the **main.cpp** section) and begin your `SixDegrees` class implementation. The due date for Phase 1 is listed in the course calendar.

IMPORTANT: You must also submit a partially completed `README`, which must include an estimate of how long it took you to complete Phase 1. You should also submit unit testing `.cpp` files for the `CollabGraph` functions you implemented.

Phase 2: The Final Chapter

It's time to create the meat of your program! Implement the remainder of the program as specified above – command loop, traversal algorithm, printing, the works. As usual, you must also submit evidence of thorough testing.

- Testing includes provided command files which run your program and tests the breadth of your code's functionality as well as the depth of it.
    - This includes both basic cases that test each command, as well as edge cases that test the limits of your program.
    - Your input files should be well documented as well, explaining each test and the purpose of it.
    - Please make sure you give any testing input files helpful names, and describe them thoroughly (what are they testing for?) in both the Files and Testing sections of the `README`.
- A `README` file which outlines the usual things (for which we have supplied an outline):
    1. The title of the homework
    2. Author's name (you)
    3. How much time the assignment took you in hours
    4. The purpose of the program
    5. Acknowledgements for any help you received
    6. The files that you provided and a short description of what each file is and its purpose
    7. How to compile and run your program
    8. Describe the data structures and the algorithms used
    9. Details and an explanation of how you test the various parts of the assignment and program as a whole

- Additionally, answer the following questions in your README:
    1. What is the difference between BFS and DFS? What are their pros/cons?
    2. How did you choose to tackle the incl (if implemented) and not commands? How do they compare to the BFS and DFS algorithms?
    3. State what you think your runtime is for each of the commands (excluding `quit`) commands and why.

Submitting Your Work

Be sure your files have header comments, and that those header comments include your name, the assignment, the date, and acknowledgements for any help you received (if not already credited in the `README` file). Submit your work as discussed in class.