Final Project : Movie Graph

COSC 102 - Fall '20

In this final project, you will work with maps, sets and queues so to create a graph data structure. You will design this structure by combining Java collections so as to store the information of a big data set of movies. Once all the movies information is stored in a graph you will design an algorithm to efficiently search through it and find connections between actors according to the movies they played in.

1 Overview

Sepecifically the objective of this project is to determine the shortest path (aka "traversal") between two targets, where each target represents either a *movie* or *actor*. This traversal is performed by navigating between actors and movies; you can jump from an actor to any movie that actor has starred in, and you can jump from a movie to any actor starring in that movie. The goal is to find a path between your two targets in the fewest steps possible.

We will say that each actor or movie "visited" in the traversal counts as a single step (including the start and end targets). For example, imagine we were attempting to determine a path between *Danny DeVito* and *Cate Blanchett*. We could do so in **five** steps:

Danny DeVito starred in **L.A. Confidential** with **Russel Crowe** who starred in **Robin Hood** with **Cate Blanchett**

Ultimately, each "jump" will alternate between an actor and movie, and your start and end targets will be the first and last items in your traversal, respectively.

2 Building and Traversing the Graph

In order to optimally implement the traversal algorithm, you will need to create a *graph* out of the actor/movie data. The data is supplied via a text file and several sample files are provided. The below sections outline the formatting of the dataset file, as well as the construction and traversal of the graph.

2.1 Format of the Dataset

The actor/movie data used to construct your graph is stored in a text file, provided to the `MovieSearch` class via a command-line argument at runtime (more on this below). The formatting for this file is as follows:

- Each line corresponds to a single movie
- Each line contains the following fields, separated by forward slashes (`'/'`):
  - The first field is the movie title, typically followed by the year in parentheses.
  - The rest of the line contains the names of actors who appeared in that movie. Typically they are in the format of last name, a comma, and then first name. You should store the actors' names as-is, it's not necessary to reformat.

Hints

1. Open those text files in jEdit to double-check the data line format described above.
2. Create your own file, a small one, i.e. few lines to write and debug yout code at first, adhering to the format. Possibly it could encode the graph shown on page 3.

2.2 Building the Graph

Your first task is to organize your data into a graph. First, some definitions:

- A **node** is an entity in the graph containing some piece of information (in this case, an actor or movie).
- An **edge** is a connection between two nodes.

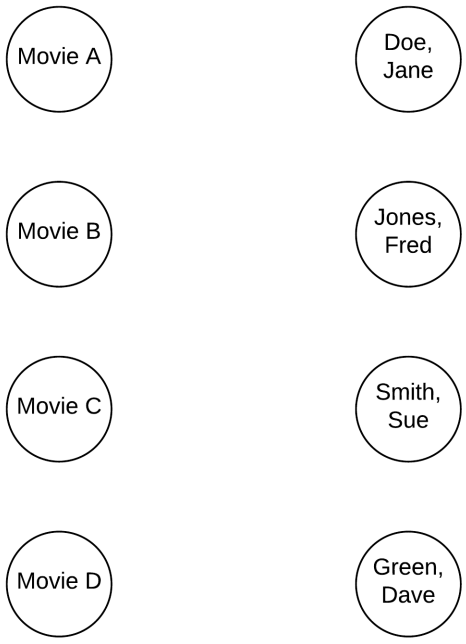
With this in mind, we will construct our graph with the following constraints:

- Every movie and actor in our dataset is a node in our graph. Additionally, all nodes are unique; actors appearing multiple times in the dataset (starring in multiple movies) only have a single node in the graph.
- Every movie node has edges to each of the actor nodes starring in the movie.
- Every actor node has edges to each of the movie nodes that actor stars in.
- Each node can have an indefinite number of edges.
- Due to the nature of our data, all edges between nodes are *bidirectional*. This means that if actor `A` stars in movie `B`, `A` should have an edge to `B` and `B` should have an edge to `A`.

For example, imagine we were building a graph based on the following small dataset:

```
Movie A/Doe, Jane/Jones, Fred
Movie B/Smith, Sue
Movie C/Doe, Jane/Jones, Fred/Smith, Sue
Movie D/Smith, Sue/Green, Dave
```

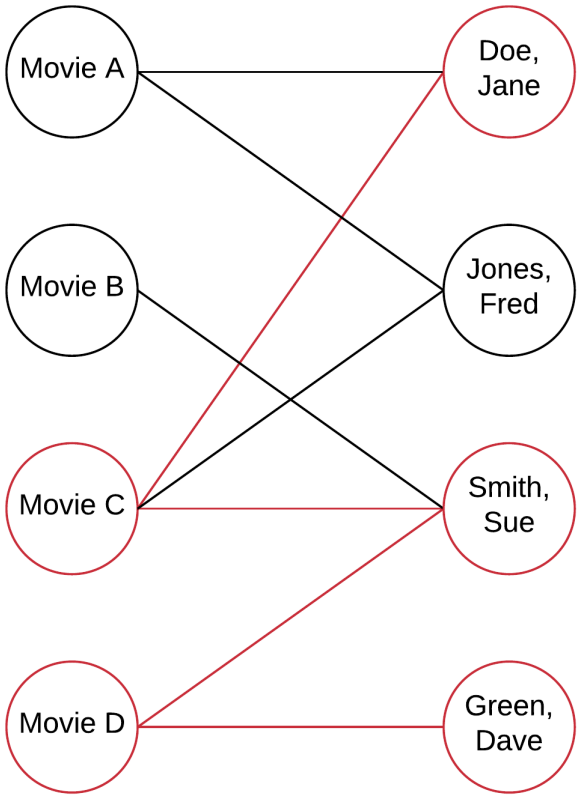
In the space below, **draw the edges** (via lines) based on the data set above. When done, verify your graph by looking at the next page:



In the above image, each circle represents a node, and the lines between circles signify each node's edges. Note that each node can have any number of edges, and we don't need to be able to distinguish between nodes representing a movie and nodes representing an actor.

2.3 Searching Your Graph

With the graph constructed, you will then create an algorithm to efficiently find the shortest path between two nodes. For example, given the graph above, the traversal between `Jane Doe` and `Dave Green` could be completed in **five** steps, as visualized below:



The red outline in the above diagram shows the shortest path traversal. **Fill in the blanks below:**

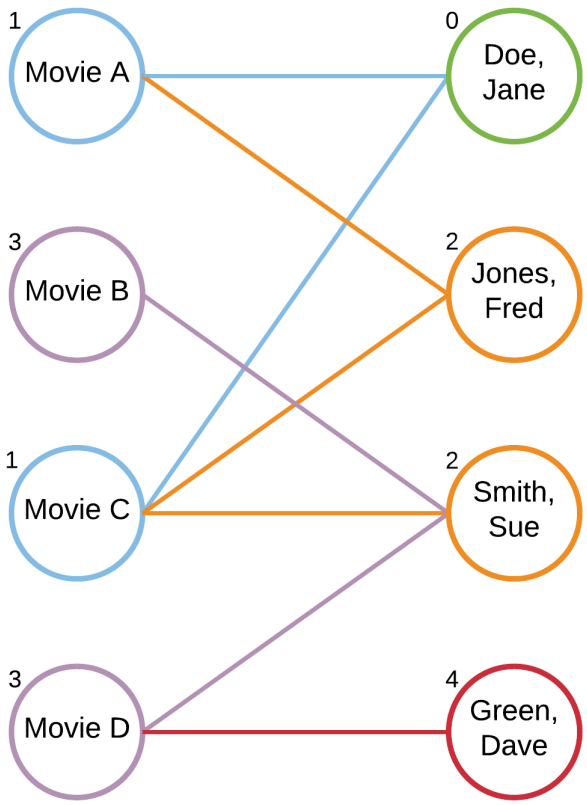
Jane Doe starred in _____ with _____ who starred in _____ with **Dave Green**

To determine this path exists, you will implement a search algorithm called **breadth first search (BFS)**. A BFS, traverses a graph and prioritizes the nodes closest to the starting node before moving on to nodes further away. The BFS also ensures that the target node is reached using a path with the fewest possible jumps.

In other words, we will start at the `target1` node, and look at all the nodes *one edge* away from `target1` to see if they match `target2`. If we don't find a match, we look at all the nodes *two edges* away from `target1`, then *three edges* and so on. The search continues until either we find `target2`, or have exhausted all of the nodes accessible from `target1`.

An ideal way to implement this algorithm is to use a **`Queue`**. The *FIFO* nature of a `Queue` makes it ideal to store nodes in order of distance from the starting point, ensuring the shortest possible path to your target. Think of the `Queue` as a "to-visit" list; it will track all of the nodes you need to visit in the order you need to visit them in.

On the following page is a walkthrough of a `Queue` implemented BFS using the same graph and targets as above. The colored graph shows the path taken, with the numbers to the upper left corner of each node indicating the distance (in edges) from the starting node, `Jane Doe`.



Draw the `Queue` (on the right) as you fill in lines below (solutions in footnote) executing the steps that traverse the graph:

1. Begin the search at one of the target nodes, let's say **`Jane Doe`**. It is our current node.
2. *Enqueue* all of the nodes we can directly visit from **`Jane Doe`**, which are those with which it maintains an edge: _____ and _____.
3. *Dequeue* the front, that is _____ is now our current node.
4. Similar to step 2, from this current node, _____, we can visit a new set of nodes. *Enqueue* all of the nodes with which _____ maintains an edge but **that we have not queued up and/or "visited" previously**. Thus, _____ is enqueued, but _____ is not.
5. Similar to step 3, *dequeue*: **`Movie C`** is our current node. Note that due to the nature of the queue, we will "visit" all of the distance 1 nodes before any distance 2 nodes.
6. Similar to step 4, from this current node, _____, *enqueue* all the nodes that haven't been processed yet: _____ is enqueued, but _____ and _____ are not.

   This process continue until the other target node is found, or there are no more viable nodes to visit.
7.
8.
9. ...

In this example, **`Dave Green`** would be reached after enqueuing and searching the purple colored nodes.[1]

[1] 2. **`Movie A`**, **`Movie C`**; 3: **`Movie A`** 4: **`Movie A`**, **`Movie A`**, **`Fred Jones`**, **`Jane Doe`**; 6. **`Movie C`**, **`Sue Smith`**, **`Jane Doe`**, **`Fred Jones`**

2.4 Reconstituting Your BFS Path

Finding a path between your two target nodes only solves half the problem – you also need to be able to retrace the steps you took to get there. You will do this by maintaining a **history** while performing your BFS.

Your history will contain information regarding your traversal – specifically it will record **each unique node you visit**, as well as **the node you reached that specific node from**. Since you never visit/enqueue the same node multiple times as part of your BFS process, you will only ever reach a given node from one place.

The below outlines the construction of this history utilizing the same graph and search from above:

- Begin at the starting node, `Jane Doe`. Add `Jane Doe` to the history. Since it is the starting node, there is no "previous node", and therefore we can say it was reached from `null`.
  Added to history: `Jane Doe` reached from `null`
- `Jane Doe`'s edges, `Movie A` and `Movie C` get added to the queue. Both of these nodes also get added to the history, and both were reached from `Jane Doe`.
  Added to history: `Movie A` reached from `Jane Doe`, `Movie C` reached from `Jane Doe`
- Dequeue the next node, `Movie A`. From `Movie A`'s edges, only `Fred Jones` gets enqueued (`Jane Doe` already visited). Thus, `Fred Jones` is added to the history.
  Added to history: `Fred Jones` reached from `Movie A`
- This process continues until the search is completed.

In this example, once the BFS is completed, the resulting history could look like the following:

- `Jane Doe` reached from `null`
- `Movie A` reached from `Jane Doe`
- `Movie C` reached from `Jane Doe`
- `Fred Jones` reached from `Movie A`
- `Sue Smith` reached from `Movie C`
- `Movie B` reached from `Sue Smith`
- `Movie D` reached from `Sue Smith`
- `Dave Green` reached from `Movie D`

Once your search is complete, the history is used to reconstruct the path taken through the graph. From the example above, the ending node, `Dave Green`, was reached from `Movie D`, which was reached from `Sue Smith`, which was reached from `Movie C`, and so on. This process continues until we make our way back to the starting node.

3 Your Tasks

Your tasks are to first implement the graph representation of the dataset, and then to implement the algorithm to find the shortest path between two nodes on the graph. You are provided with several sample datasets and two starter `.java` files, described below.

3.1 `MovieSearch`

The **`MovieSearch.java`** file contains all of the code which handles prompting the user, parsing the data file, and displaying the results of the BFS algorithm. **Do not modify any code in this file.**

`MovieSearch` contains the `main` method you will run to start the program. The class expects one command line argument, which is the name of the file you want to use as your dataset to construct your graph. This file must be formatted per the requirements outlined in **Section 2.2**; several sample dataset files are provided for you.

This class handles parsing the provided dataset file, which it stores in an instance of a `MovieGraph` object (more on this below). If an improperly formatted file is provided, the program will terminate with an error message.

When ran, users are prompted repeatedly to enter movie/actor names until they decide to quit by responding to a prompt with `!quit`. After each pair of movie/actor names is provided, the shortest path between the targets is displayed. If no path is available, or if either target doesn't exist in the dataset, an error message is displayed.

3.2 `MovieGraph`

The **`MovieGraph.java`** file is where you will write the code to construct the graph and perform the shortest path search. You will do this by completing the implementation of this class' constructor and `findShortestPath` method. **All of your code will go in this file.**

A `MovieGraph` object represents a graph of movie/actor data as described in **Section 2.2**. At runtime, `MovieSearch` creates a `MovieGraph` object, passing it all of the parsed data out of the supplied data file. Additionally, `MovieSearch` uses the `findShortestPath` method to determine the shortest path between each pair of targets the user enters.

TASK 1: **`MovieGraph`'s constructor** should initiate/execute your code to build the graph of movie/actor data. This constructor accepts a single argument: a `Set` of `String[]` containing all of the movie/actor data `MovieSearch` parsed out of the dataset. More specifically, the data in this `Set` is organized as follows:

- Each `String` array in the `Set` represents one movie (or one line) from the dataset file.
- Each index in a `String` array represents one token of data from the line of the dataset. Within the array, these tokens of data are ordered the same way that they appear in the file.

TASK 2: The **`findShortestPath(String target1, String target2)`** method finds and returns the **shortest path** between the two targets in your graph. This path is returned as an `ArrayList` of `String`s, where each `String` represents one node or "step" in the traversal between the start and end targets.

If a path is found, the first `String` in this `ArrayList` should be `target1` and the final `String` `target2`. If there is no viable path between the two targets, this function will return an **empty `ArrayList`**.

3.3 Sample Output

Below is a sample of what the output from `MovieSearch` should look like once your implementation is complete. This example makes use of the provided `cast.mpaa.txt` file:

```
> java MovieSearch cast.mpaa.txt
Enter an actor or movie name, or type !quit to exit:  Russell, Kurt (I)
Enter another actor or movie name:  McKellen, Ian
Finding shortest link for targets: 'Russell, Kurt (I)' and 'McKellen, Ian':

Link found in 5 steps!
Russell, Kurt (I) --> Vanilla Sky (2001) --> Leung, Ken (I) --> X-Men: The Last
Stand (2006) --> McKellen, Ian
```

This sample represents one test only. That is not where you are debugging should start, it is only our proof of concept. You should debug as you go, using println at the console to display building simple graph, searching paths in your own small dataset file, etc... Make sure you present the process you used in your reflection.

3.4 Additional Tips

Below are a few extra tips to help you get started:

- Start with the the construction of your graph. Think about what data structure(s) are most complimentary to how you need to store and traverse the graph; remember that each node can have any number of edges. Draw diagrams representing the maps, arraylist, specific collections that implements the graph, and BFS traversal.
- If you're having trouble figuring out how to extract the data from the constructor's `Set`, start by looking at the `Set` API page (*hint: in the Java API, `Set` is an interface! Trace `MovieSearch` to find ideas.*)
- You may want to review the Java implemented `HashSet`/`HashMap` data structures – read the API!
- Use debug print statements to validate the contents of your graph before moving on to the BFS algorithm.
- A few dataset files are provided for you, but we recommended three times above to make your own, so do it! Use a small, controlled dataset that is easy for you to develop and to follow as you test your code incrementally.

4 Pairing and Submission

This project replaces your final exam. The below sections detail the expectations regarding (optionally) working in a pair and your submission.

4.1 Working with a Partner

For this final project, you may **optionally work with one additional person of your choosing**. You may also work by yourself if you prefer. If you decide to work in a pair, you are expected to abide by our pair programming principles while being remote: you have to use the navigator and driver roles, regularly swapping responsibilities, working all the time together. It is against the honor code to divide and conquer the implementation of this project if you are working in pair, as it is to use code that is not yours.

If you are working with a partner, you **must inform Professor Fourquet by end of day, Wednesday December 9th**.

4.2 Submission Requirements

In addition to your final code, students are required to submit **either** of the following:

- a short recorded video, or
- a *ReadMe* text file

In this video or *ReadMe*, you must detail:

- Your work process: describe how you designed, implemented, and tested your program.
- Your learning experience: describe what you took away from this final project. For example, detail what you learned about the graphs, maps, and/or the breadth-first search algorithm that will help you in future endeavors.

If working in a pair you must additionally complete the following:

- a peer-evaluation. A document will be shared with you and **both partners** must complete this individually.
- a debrief meeting to discuss the work you collaborated on. You will need to schedule this with Professor Fourquet for sometime on or before **Friday, December 18th**.

Lastly, if working in a pair only one person needs to submit the final code. Additionally, the pair only needs to submit one video or *ReadMe* so long as both members contribute.

4.3 Due Dates

This assignment is due for all students on **Wednesday, December 16th** at **noon (12:00PM) Hamilton time**.

As indicated above, your submission must include your final code as well as your *ReadMe* **or** video file and, if working in a pair, the peer evaluations. Please share your video files with me thanks to google drive.