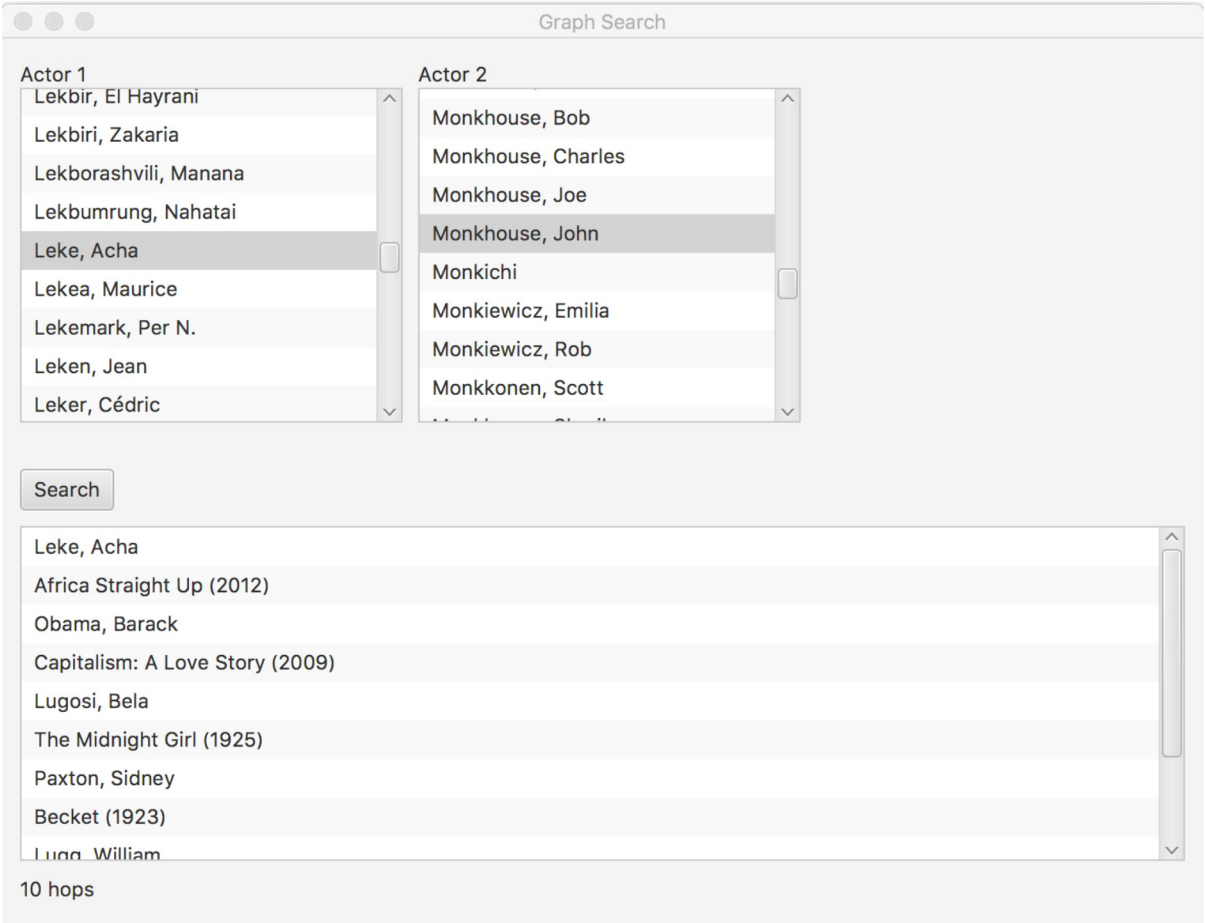
CS2103 2018 B-Term -- Project 3 -- Graph Search

Prof. Jacob Whitehill

Introduction

In this project you will develop software instructure to model large **graphs** along with a search engine to find **shortest paths** between any pair of nodes in a graph. As an example application illustrating how to use this infrastructure, you will develop a search engine for the Internet Movie Database (IMDB) that allows users to find a shortest path of actor nodes and movie nodes that connect any two specified actors (if such a path exists). The code that your team will write to load data and find paths will interface with code (written by me) that implements the graphical user interface (GUI -- see figure below) of the application.



Parsing the IMDB data

One of the tedious but crucial aspects of real-world computer programming is converting data from one format to another. It's important to become efficient and very reliable at doing this, and this project provides a practice opportunity. In particular, you will create a class `IMDBGraphImpl` that implements the `IMDBGraph` interface. Implementing this class will require you to parse data files that contain information on actresses/actors and the movies in which they starred.

While you are welcome to use more sophisticated parsing methods such as regular expressions, **these are not required for this assignment**. In fact, my own implementation simply uses a `Scanner` (just its `hasNextLine` and `nextLine` methods -- nothing fancier) as well as some standard `String` methods: `indexOf`, `substring`, and `lastIndexOf`.

Data format

IMDB uses a non-standard and rather unfortunate data format...as sometimes happens in the real world. The dataset consists of two data files: `actors.list` and `actresses.list`. Each file starts with a preamble that describes the file structure. The `actors.list` data file begins in earnest with the lines:

```
THE ACTORS LIST
===============

Name                    Titles
----                    ------
```

(The `actresses.list` begins analogously.) In each file, actresses/actors are listed alphabetically. For each actress/actor, the list of movies in which she/he starred is listed, followed by an empty line to separate the current actress/actor from the next actress/actor, e.g.:

```
$hutter                 Battle of the Sexes (2017)  (as $hutter Boy)  [Bobby Riggs Fan]  <10>
                        NVTION: The Star Nation Rapumentary (2016)  (as $hutter Boy)  [Himself]  <1>
                        Secret in Their Eyes (2015)  (uncredited)  [2002 Dodger Fan]
                        Steve Jobs (2015)  (uncredited)  [1988 Opera House Patron]
                        Straight Outta Compton (2015)  (uncredited)  [Club Patron/Dopeman]
```

In the example above, an actor named `$hutter` (for some reason he spells his name with a $ sign) has starred in 5 movies and/or television shows, one movie/show shown on each line. At least one tab character (`\t`) separates the actor from the first movie/show, and at least one tab character precedes each movie/show given on subsequent lines. Note that the actress'/actor's name is **not** repeated for each movie/show in which she/he starred -- it is shown only once for the **first** movie/show. Along with the name of the movie/show and the year, additional information may be given including the name of the character and the billing position of the actor in the credits. **In this programming project, you should only parse the title and year of each movie.** In the example above, your parsing code should thus produce the following five strings for the actor `$hutter`:

- `Battle of the Sexes (2017)`
- `NVTION: The Star Nation Rapumentary (2016)`
- `Secret in Their Eyes (2015)`
- `Steve Jobs (2015)`
- `Straight Outta Compton (2015)`

In particular, **the string representing the title of each movie that your code parses should include the year in parentheses as part of the title string**.

Parsing the data

The `actors.list` and `actresses.list` data files are **not** in ASCII format; rather, they are in **ISO-8859-1** format. This file format allows the use of non-ASCII characters that are common in many languages (but not English). It is essential that you use the proper character decoder when parsing the files. We recommend that you use the `java.util.Scanner` class, e.g.:

```
final Scanner scanner = new Scanner(new File(actorsFilename), "ISO-8859-1");
```

The line of code above instantiates a new `Scanner` object that reads from a `java.io.File` (on disk) and decodes it using a `ISO-8859-1` decoder. From then on, you can use all the standard methods available through the `Scanner` class; some of the more useful ones include: `hasNextLine` `nextLine`, and `findInLine`.

Keeping the memory usage tractable

The IMDB contains about 30,000,000 lines of data that your code must parse and analyze in order to build the graph. In order to reduce the space and time costs involved in this project, you should **exclude** any **TV series** as well as any **TV movie**. As described in the data files, TV movies contain `(TV)` in the line describing them, e.g.:

```
                        Sweet Talkin' Guys (1991) (TV)  (archive footage)  [Himself]
```

TV series have quotation marks around the titles, e.g.:

```
                        "All You Need Is Love" (1977) {Introduction (#1.1)}  [Himself]
```

In addition, you should also **exclude** any actress or actor who starred **only** in TV series/TV movies; an example of such an

actress/actor is `Aaltio, Mikko`:

```
Aaltio, Mikko           "Bitwisards" (2016) {Bitwisardin saari (#1.10)}  [Jorma]  <6>
                        "Bitwisards" (2016) {Pitkästä ilosta ja tyhjännauramisesta (#1.11)}  [Jorma]  <10>
                        "Bitwisards" (2016) {Uusi aika (#1.8)}  [Jorma]  <10>
```

because he starred only in shows in a TV series.

Note: do **not** worry about trying to correct any typos that may be present in IMDB, or trying to "merge" two actors whose names differ in trivial ways (e.g., the presence/absence of a middle initial). You should assume that each actress/actor and each movie title in the `actors.list` and `actresses.list` datasets are fully correct (even though this is not actually the case).

Implementing the `IMDBGraph` interface

As mentioned above, one of your tasks in this project is to implement the `IMDBGraph` interface. This class should contain a public constructor that takes two parameters (in order):

1. The absolute path (e.g., `"C:\MyDirectory\actors.list"`) of the `actors.list` file.
2. The absolute path (e.g., `"C:\MyDirectory\actresses.list"`) of the `actresses.list` file.

Your code **must** load the IMDB data from the files whose absolute paths are specified in the parameters passed to the constructors. In addition, these constructors **must** throw a **java.io.IOException** -- they should **not** attempt to catch this exception themselves. Both of these criteria are essential to enable us to grade your code automatically.

Specifically, your code should contain the following constructor:

```
public IMDBGraphImpl (String actorsFilename, String actressesFilename) throws IOException {
  // Load data from the specified actorsFilename and actressesFilename ...
}
```

Correspondingly, our test code will invoke your constructors as:

```
graph = new IMDBGraphImpl("IMDB/actors.list", "IMDB/actresses.list");
```

Finding shortest paths between nodes in a graph

To find a shortest path between node `s` and `t` in a graph, you should implement a **breadth-first search (BFS)**, as described during class. Once you have found a shortest path -- if one exists -- you then need to **backtrack** from `t` back to `s` and record the sequence of nodes that were traversed. The result should then be returned back to the caller (the `GraphSearchGUI`, in this case). Even for large social network graphs consisting of millions of nodes, BFS will be very fast, as it operates in $O(n)$ time (average case), where n is the number of nodes in the graph. For this part of the assignment, you should create a class called `GraphSearchEngineImpl` that implements the `GraphSearchEngine` interface. The `findShortestPath` method of your `GraphSearchEngineImpl` should return an instance of type `List<Node>` in which the **first** element of the list is `s`, the **last** element of the list is `t`, and the **intermediate** nodes constitute a shortest path (alternating between movies and actresses/actors) that connect nodes `s` and `t`. If no shortest path exists between the pair of nodes, then your method should return `null`.

Make sure that your `GraphSearchEngineImpl` is not "tied" to the IMDB data in any way -- the search engine should be useful for **any** graph of `Node` objects.

Requirements

1. **R1**: Create a class called `IMDBGraphImpl`that implements the `Graph` interface:
   - The class should load data from the two files whose filenames are specified in the constructor. Your parsing code should exclude TV series, TV movies, and any actresses/actors who starred only in TV series/movies.
   - The class should have a public constructor, which can throw an `IOException`, that take two parameters specifying the absolute paths of the actors and actresses data files, in that order.
   - The class should construct a graph based on the data parsed from these files.
   - To make sure that everyone is parsing the same data files, you **must** use the IMDB data contained in the Zip file at the following link: IMDB.zip.
2. **R2**: Implement BFS within a class `GraphSearchEngineImpl` that implements `GraphSearchEngine`. Your search engine

should be able to find shortest paths between **any** pair of `Node` objects (or return `null` if no such path exists) -- whether they are IMDB nodes, CiteSeer (scientific publication) nodes, or anything else.

Make sure that the files you submit are named exactly as described above so that our automatic test code can give you credit for your work. Note that you may **not** change any of the interfaces that we provide in any way.

Teamwork

You may work as a team on this project; the maximum team size is 2.

Design and Style

Your code must adhere to reasonable Java style. In particular, please adhere to the following guidelines:

- Each class name should be a singular noun that can be easily pluralized.
- Class names should be in `CamelCase`; variables should be in `mixedCase`.
- Avoid "magic numbers" in your code (e.g., `for (int i = 0; i < 999 /*magic number*/; i++)`). Instead, use **constants**, e.g., `private static final int NUM_ELEPHANTS_IN_THE_ROOM = 999;`, defined at the top of your class file.
- Use whitespace consistently.
- No method should exceed 50 lines of code (for a "reasonable" maximum line length, e.g., 100 characters). If your method is larger than that, it's probably a sign it should be decomposed into a few helper methods.
- Use comments to explain non-trivial aspects of code.
- Use a Javadoc comment to explain what each method does, what parameters it takes, and what it returns. Use the `/**...*/` syntax along with `@param` and `@return` tags, as appropriate.
- Use the `final` keyword whenever possible.
- Use the **most restrictive** access modifiers (e.g., `private`, default, `protected`>, `public`), for both variables and methods, that you can. Note that this does not mean you can never use non-`private` access; it just means you should have a good reason for doing so.
- Declare variables using the **weakest type** (e.g., an interface rather than a specific class implementation) you can; ithen instantiate new objects according to the actual class you need. This will help to ensure **maximum flexibility** of your code. For example, instead of
  ```
  final ArrayList<String> list = new ArrayList();
  ```
  use
  ```
  final List<String> list = new ArrayList<String>();
  ```
  If, on the other hand, you have a good reason for using the actual type of the object you instantiate (e.g., you need to access specific methods of `ArrayList` that are not part of the `List` interface), then it's fine to declare the variable with a stronger type.

Getting started

Download the Project3 starter file.
Download the IMDB data.

Note: the `GraphSearchGUI` class is the motivating application of this project, but you are not required to use it or extend it in any way. It's mainly just for fun.

Testing

In this project, we will not grade your test code. However, **we highly recommend that you create a toy dataset** to help you develop and debug your BFS code. Creating toy datasets is a very useful technique in a variety of software engineering (and computer science) domains. This will greatly reduce the amount of computation that is required to make progress on the project. It will also make it easier to understand and eliminate any bugs that your code might contain.

What to Submit

Create a Zip file containing `IMDBGraphImpl.java` and `GraphSearchEngineImpl.java`, along with any other files that you wish. Submit the Zip file you created to Canvas. Your code will be graded by automatic scripts. **Submission deadline**: Wednesday, November 14, at 11:59pm EDT.