Noelle Davis, Saniya Dawn, and Sarah Goldman

CS 230

5/3/23

Final Project Report: Gender Imbalances in Hollywood Movies

1) Title

The title of our project is Gender Imbalances in Hollywood Movies. The names of our team members are as follows: Noelle Davis, Saniya Dawn, and Sarah Goldman. We would like to thank the professors and TAs who helped us: Takis Metaxas, Stella Kakavouli, Smaranda Sandu, Jada Onwuta, Becky Chen, Carrie Wang, and Maria Ordal.

2) Introduction

In this project, we were tasked with implementing a HollywoodApp that had numerous functionalities. The app had to be able to *create a graph* linking actors with the movies they acted in. We were also tasked with *generating a value* for each inputted movie which would inform the user whether or not that movie had over 48% women in its cast; a value of 1 indicates the movie **passed.** Additionally, the app had to be able to *return the list of movies* a given actor had acted in, as well as *return the list of actors* in a given movie. Finally, the app had to be able to *find the **degree of separation*** between a given actor a1 and another given actor a2, meaning the number of movies that separated those two actors, and return that number.

We would like to cite the many helpful references we utilized while completing this project: the CS 230 slides by the Wellesley CS 230 team, reading materials about ways to measure gender imbalances in Hollywood films from statistical analysis website FiveThirtyEight, descriptions of the data set files from the FiveThirtyEight Github Repository, the Java Foundations textbook by John Lewis, Peter DePasquale, and Joseph Chase, reading materials from MIT 6.101 about searching, a helpful article about breadth-first versus depth-first searches in Java from Codecademy, and the Java API, specifically the API for Iterator, Queue, Hashtable, LinkedList, Vector, Scanner, Set, String, IOException, and File.

3) Method

1.1

We wrote **classes Movie and Actor** to implement the Hollywood App class. They absolve the need for repetitive code in Hollywood App.

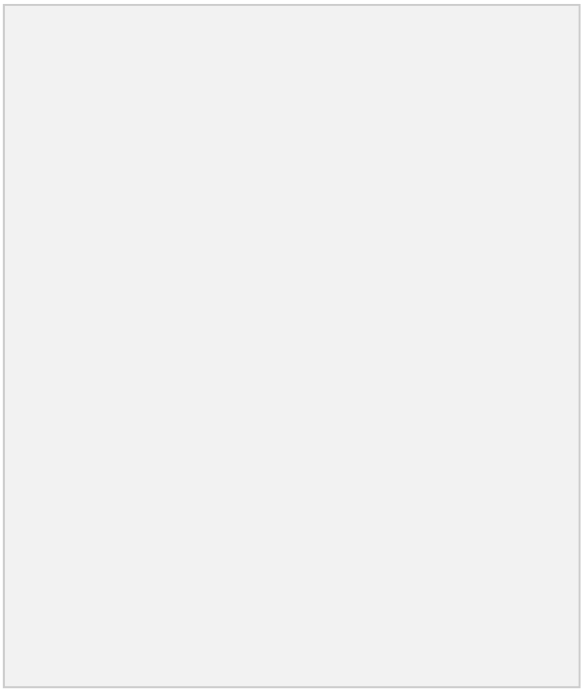
Class Movie creates a **HashTable** in which the keys are the movie names and the values are its actors. To implement this, we created a Hashtable<String, Vector<String>> called movieTable. In our constructor movieTable is initialized to be used in the method fillMovieTable, which fills the hashtable as described above. It does this by utilizing a Scanner to **read an input File** if it can be found. A new Vector<String> is created for each line in the input file. Then each line is split by [","], with the result put into a String [ ] wordArray. **The zeroth element of wordArray is the movie name and the first is the actor name.** The method then **checks if the hashtable already contains the movie name.** If not, the movie and actor are added to the HashTable. Otherwise, the actor's name is added to the existing Vector of actors via a holder Vector<String> currentActors and the use of .add(actor). We then replace the movie's value with the updated Vector.

Class **Actor** is similar to **Movie** as it also creates a **Hashtable <**String, Vector<String>>**;** however, in this case, the keys are the actor names and the values are the movies they've been in.

Both classes define getters of both the keys and key values. The method getKeys() uses a Vector<String> called movieNames as the return value. The Vector is filled via an Iterator<String> that iterates through the **Set** of key values returned by calling .keySet() on movieTable. The method then returns Vector<String> movieNames. To get the key values, the method getKeyValues() uses a specific movie name key to return the values stored at that element. It does this by calling .get on movieTable. The implementation is similar in Actor, but there is no iterator used in getKeys for the Actor class. There is a Set<String> called actorNames that gets the values of actorTable by calling .keySet(), and then it returns actorNames. The method getKeyValues() is the same except it takes an actor's name to return the value stored in actorTable.

Both classes are utilized in the Hollywood App class. In the constructor of the HollywoodApp class, we initialize Vectors, HashTables, and Graphs. The string passingMov gets the method of fortyEightPassers, and the actedWithTable gets the method actedWithT().

1.2



We created this graph with the **saveTGF method from AdjListsGraph,** and used organic layout in yEd. In the method makeGraph in our HollyWoodApp class, we initialized our Vector<String> keys by setting it to the keys of the Movie table mTable, using the .getKeys() method. A nested for loop was then used to set the elements of keys into a String movie, and movie was then added to the AdjListsGraph moviesAndActors using addVertex() and elementAt(). Then a Vector<String> actors was made and set to the keyValues of the mTable using getKeyValues(). A nested for loop went through actors to add the actor as a vertex and to add an edge between the actor and movie. We noticed that we obtained a bipartite graph. Actors are connected to movies and movies are connected to actors, but movies and movies are not connected, and actors and actors are not connected. We wondered how this would affect traversals/paths.

1.3: fortyEightPassers

To answer this question, we created the fortyEightPassers method in HollywoodApp, which returns a string denoting which films have over 48% women in their casts. As noted previously, our Actor class creates a hashtable in which the keys are the actor names and the values are the actor's gender followed by all the movies they have acted in. Our Vector<String> keys holds all the movie titles with the actors as values. We set our Vector<String> moviePassers, and then using a for loop we **go through each value in keys and make two double variables called totalCount and femaleCount**. TotalCount is set to the size of the Vector<String> actors. With our first for loop, we set the value of the String movie to the elementAt() keys, and the Vector<String> actors that gets the keyValues from a movie and setting that value into actors by calling getKeyValue() on mTable.

As the **second for loop goes through actors**, it sets each value of actor to a String actor using ,get(), and another Vector<String> called genderV is made that gets the first element of the key Value of aTable, which is found by using the .getKeyValue() method, and then using the .firstElement() method on that. Within the nested for loop there is an if statement, saying that for each String "female" in genderV, add 1 to femaleCount. Going back into the original for loop, another if statement is made that if femaleCount decided by the total count is greater than the threshold value, we add that movie key to moviePassers. There is a final for loop that goes through moviePassers, and it adds each element of moviePassers onto the String passingMovies, and in the end this method returns a string of all the movies that have 48% women in their cast.

2.1: allMovies

To find the list of movies an actor has played in, given an actor, **we used our previously implemented Actor class.** We created an instance of Actor named aTable that was initialized in our constructor using the file we wanted. Moving down to the contents of the method itself, we created a String named movieString to return our answer. We then created a Vector<Vector<String>> named actorValues to hold all the values in the aTable (actor table) for the given actor. We then **extracted just the Vector<String> of movies** by taking the last element of actorValues (since the gender of the actor was irrelevant for this problem) and storing that under the variable name actorMovies. We then **looped through the movies in actorMovies** and added them to the movieString, returning the movieString at the end.

2.2: allActors

To find the list of all actors who have played in a given movie, **we used our previously implemented Movie class.** We created an instance of Movie named mTable that was initialized in our constructor using the file we wanted. Moving down to the contents of the method itself, we created a String named actorString to return our answer. We then created a Vector<String> named movieValues to hold all the values in the mTable (movie table) for the given movie. We then **looped through the actors in movieValues** and added them to the actorString, returning the actorString at the end.

2.3: degreeOfSeparation

In order to solve the degree of separation problem, we first created a method called actedWithT. This method takes no inputs. We utilize several data structures in this method of type Hashtable<String, Vector<String>> and Vector<String>. The variable actedWithTable is a Hashtable and the variables movieNames and actedWith are initialized in this method. We then use a series of nested loops. The **first loop iterates through each movie name.** For each index, i, in the Vector, **allActors** gets the return of .getKeyValues() (the actors who acted in the movie); we also store the String **currentMovie**, the current index location in the movieNames Vector.

With the Vector<String> allActors having all the actors for each movie, **the second loop iterates through each element stored.** At each index, k, the **String current Actor is updated** to that value and a Vector<Vector<String>> named **otherActorsAndMovies** is initialized. **In the third loop, we iterate through all the elements again to avoid putting the actor in its own value.** This third loop contains an if statement that considers if other_ele is equal to the current actor. If so, the element is skipped. If not, otherActors adds the current value of other_ele and the current movie. The holder, otherActorsandMovies adds the otherActors Vector.

We then **continue in the second loop** to an if statement that states if actedWithTable does not contain the key of our currentActor, we will **input (via .put()) our current actor as the key and otherActorsAndMovies as the values.** The else statement states, if our actedWithTable already contains the key of our currentActor name, we create another Vector<Vector<String>> named heldActors, which gets the current elements of our actedWithTable at the currentActor. We then iterate through otherActorsAndMovies. If heldActors does not contain the otherActorsAndMovies at the index, h, of the for loop, it will add otherActorsAndMovies to heldActors, which updates the co stars of our current actor. We then **update our actedWithTable to hold our currentActor and heldActors, via .replace().**

When all of the actors are in the HashTable with all of the costars, the method returns actedWithTable.

The getter getActedWith returns the values held at the actor name key in actedWithT.

As requested, degreeOfSeparation finds the path between two given actors and returns the integer of the path length minus one (only if the path length is greater than 0). To implement this, we made a LinkedQueue<Vector<Vector<String>>> **bfsPaths**, a LinkedList<Vector<String>> **visited**, which is used to ensure that the same vertex is not searched twice, and initialized the Vector<Vector<String>> **moviePath**. We also made a Vector<Vector<String>> actorAndMovieName and a Vector<String> a1AndMovie.

There are two base cases in our implementation. The first is an if statement that returns a 0 if actor 1 (**a1**) and actor 2 (**a2**) are the same. The second utilizes another Vector<Vector<String>> called vals that gets who a1 acted with. A for loop is used to go through the values. If any of the elements contain the other actor's name, the method will return 0. Following the two base cases we use the .add() method to add actor 1 and a null to a1AndMovie. This is so we don't return a value for the connection between actor 1 and actor 1. Then a1AndMovie is added to actorAndMovieName. In bfsPaths, we enqueue actorAndMovieName, and add a1AndMovie to visited.

From there, we created a **while loop that ends when our queue is empty,** as that would indicate that we have gone through all options and there is no path between a1 and a2. While the queue is not empty, we iterate through each path in the queue. For each path, we **dequeue**, **add the last element in the path to the visited LinkedList, and search the last element's neighbors for a2,** storing those neighbors in the variable *neighbors*. If the last element's neighbors contain a2, we have found our path, so we **return the length of the path** (minus one to account for our base case being 0.) Otherwise, we **go through each neighbor** in neighbors to **make sure that we haven't visited** them yet, **merge them with the dequeued path,** and **add that new path to the queue.** We repeat this process until we either find the path or run out of paths to search, in which case the method returns -1.

4) Collaboration

Our group met 8 times over the course of the project. Although two of these meetings were for planning purposes, we spent a total time of **30+ hours coding, brainstorming, debugging, and refactoring, not including times when members went to office hours alone.**

We completed the work as follows:

Noelle brainstormed and wrote pseudocode for all three of the classes. In the Actor and Movie classes she worked on the creation of the HashTables of actor and movie values, the toString()s, and the getters and setters. In the Hollywood App she worked on makeGraph(), fortyEightPassers(), allMovies(), allActors(), actedWithT(), getActedWith(), and degreeOfSeparation() . In addition she worked on JavaDoc and test cases for all of the classes as well as in-line comments. Outside of the code, she went to office hours with Stella, Takis, Jada, Carrie, Becky, and Maria. Finally, she also wrote part of the final project report.

Saniya helped come up with ideas and pseudocode for HollywoodApp class. She worked on the creation of degreeOfSeparation(), actedWithT(), allMovies(), allActors(), makeGraph(), and the toString() methods. She also worked on javadoc, in-line comments, testing cases, and writing the final report. She also attended office hours with Carrie, Becky, and Stella.

Sarah brainstormed and wrote pseudocode for all three classes. She also worked on the creation of the Actor and Movie classes; specifically, she worked on HashTable creation and the toString() methods of those classes. In the Hollywood App she worked on makeGraph(), fortyEightPassers(), actedWithT(), getActedWith(), and degreeOfSeparation(). She also worked on JavaDoc, wrote in-line comments, and helped with brainstorming testing cases. Outside of coding, she went to office hours with Stella, Smaranda, Jada, Carrie, Becky, and Maria, and wrote part of the final project report.

We executed the work as described above.