Bechdel Graph Final Project

Aileen Du, Gwen-Zoe Yang, Audrey Wang, Suzanne Xu
Wellesley College
CS 230: Data Structures
Smaranda Sandu
April 28, 2024

Introduction:

In this project, we investigated real-life data of various movies, including their cast, which were represented in our implementation by many types of data structures. We used ADTs that we learned throughout this course in order to read and analyze the given data sets on certain Hollywood movies and their actors. Firstly, we created an implementation of a bipartite, undirected graph that had vertices, which represented movies and actors, and edges, which represented the relationship between the actor and movie. It was a bipartite graph because it is not possible to have a movie play a movie and an actor play an actor. Therefore, there cannot be edges connecting vertices on the left side and there cannot be edges connecting vertices on the right side. Secondly, we created a visualization of the given data sets, through yEd, by processing them in a method we created. Next, we created two methods that return a list of actors given a movie and a list of movies given an actor. We used these two methods in our implementation of a method that returns the number of movies that separate two given actors. Finally, we had a method that creates two lists, one which passes the test and one that fails the test, which tells us whether or not the movie has some percentage, inputted by the user, of women in its cast.

Methods:

For our HollywoodGraph, we implemented the Graph interface. We considered implementing the AdjListsGraph rather than Graph, but decided we wanted to make our own instead to better understand the Graph interface. We defined the inherited methods according to the instructions, and made 6 new instance methods. We made an ArrayList of type T for the vertices, and an ArrayList of LinkedLists of type T for the arcs. We chose to use ArrayLists of type T and LinkedLists because those are the implementations we discussed in class. In addition, ArrayLists are easily indexed, which is necessary for finding the corresponding arcs to the vertices, and LinkedLists are easy to connect to each other. Type T also makes the HollywoodGraph flexible and able to take on different data types. In addition, we made an integer instance variable called count, which holds the total number of vertices. Finally, we had two private Hashtables with a String and integer pair to help us in another method called findNewBechdelValue.

In terms of methods that we added ourselves, we wrote readFile, findActors and findMovies, findSeparation with a helper method called createNewPath, and findNewBechdelValue. readFile takes a String containing the file name that we want to convert into a HollywoodGraph and read all the actors into the graph. In addition, it reads all of the movies into the femaleMovies and totalMovies Hashtable, with the number of female actors or total actors associated in the String integer pair. findActors and findMovies finds the arcs of a given vertex (or, the movies/actors of a given actor/movie). These two methods did the same thing, but we made two separate methods since, although the program considers actors and movies to be the same type, the user would not. Thus, having two separate methods makes the

code more accessible and easily understandable to the user. findSeparation employs BFS (breadth first search) in finding the number of degrees of separation of two given actors working together. The findSeparation helper method, createNewPath, is used to create new paths to queue into the BFS ArrayQueue. findSeparation increments the instance variable, separation, and returns it.

Finally, for findNewBechdelValue, we use the scanner to take user input about what percentage would be considered a pass for the bechdel test and iterate through all the movies in femaleMovies. For every movie in femaleMovies, we divide the number of total female actors (found in the String, integer pairs in the femaleMovies Hashtable) by the number of total actors (found in the String, integer pairs in the totalMovies Hashtable). Then, we check if the percentage of female actors in the given movie is greater than or equal to the inputted percentage. If it is, we add it to a LinkedList of all the passed movies, and if it isn't, we add it to a LinkedList of all the failed movies. This Bechdel Test is significant because it allows the user to decide what percentage of female actors they want to consider a pass. If they enter a relatively high percentage, like 50%, and get only a handful of movies, they can then run the program again and readjust their percentage to get more movies to pass the test. Our testing method allows the user to manipulate the test themselves to suit their own needs, which is useful in itself.

Results:

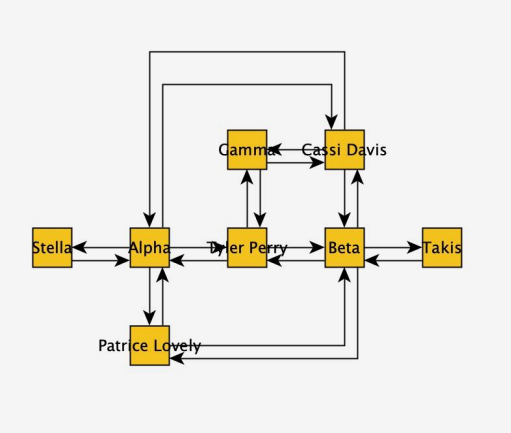

Figure1. Tgf file for small_castGender.txt

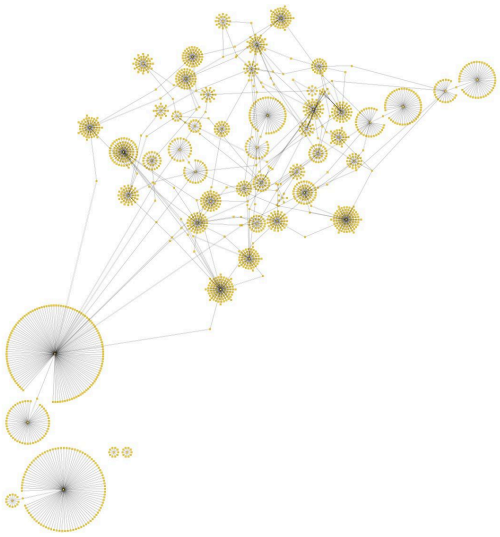

Figure2. Tgf file for nextBechdel_castGender.txt

In the main method for testing, we created three "dummy" Hollywood Graph objects that had certain testing txt files as their arguments. For the first dummy, we printed the original adjacency list graph, what the graph looked like after another vertex and edge was added, and what the graph looked like after an edge and vertex were removed. The getters and setters were also tested with the first dummy. For the second dummy, it creates a TGF graph and prints out the vertices and arcs. It also prints the separation numbers of various actors and a LinkedList of movies that pass the new Bechdel Test that we created. For the third dummy, it creates another TGF graph, given a different testing file. This dummy object tests findMovies() with Jennifer Lawrence, a method which returns all the movies a given actor is in, and findActors() with Jungle Book, a method which returns the cast of a given movie. It also prints the separation numbers between

Megan Fox and Tyler Perry, and Nick Arapoglou and Tyler Perry, and a LinkedList of movies that pass the new Bechdel Test.

Conclusions:

By doing this project, we were able to learn a real life implementation of the Graph data structure, which led us to apply OOP paradigms seen in the course. More specifically, we saw abstraction in our HollywoodGraph class that had multiple methods, along with their helper methods, that performed our intended behaviors. All of our instance variables and methods were encapsulated using the private modifier. We were able to meet at least once weekly to design, implement, and test computer programs from the start of our project, which resulted in efficient work. In applying our experience from assignments and labs, reading in a file and converting into a graph on yEd used different data structures: we gained a deeper understanding of the relationship between ArrayLists and LinkedLists by utilizing the two in our implementation of the Graph interface. We honed our problem solving skills in our attempt to code every task. As we thought through how to implement the HollywoodGraph class, we considered making two separate ArrayLists for the vertices of movies and actors, but ultimately decided that it made more sense for us to keep them in the same list given the bipartite nature of the problem. Also, in coding task 2.3 where we were tasked with trying to find the separation number between two actors, we learned how to keep track of which vertices had been visited by using breadth-first search that we looked at in the course. We were able to make decisions when thinking about whether or not multiple nested for loops would be beneficial for runtime efficiency. Everyone used the style we learned throughout the course and documentation of methods as we coded. Aside from technical learnings, we also got a sense of what it is like to work on a coding project in a larger group. We had to coordinate schedules, divide work equally, and efficiently discuss, as well as implement ideas accordingly.

Collaboration:

Our work was divided amongst team members evenly throughout each task. In Task 1.1, Audrey and Aileen began by individually drawing diagrams to show the vertices and edge relationships. Then, we convened and decided that we could have vertices of both movies and actors. Suzy and Gwen coded the methods once we deemed the relationship an edge if an arc existed both ways, movie to actor and actor to movie. In Task 1.2, we all looked back on our lab that used yEd as a refresher on our to create a graph from a text file. Everyone coded the actual constructor of HollywoodGraph to call on our readFile method that we wrote together at our April 10th meeting. In Task 2.1 and 2.2, we discussed as a group and collectively thought it would be easiest to have the movie and actor as a parameter and return a linked list. We coded this together at our April 23rd meeting. Task 2.3 took a lot of thinking on everyone's part. We spent two meetings thinking about whether or not recursion would be a good idea since it made sense to iterate through all the neighbors of one vertex and the neighbors of those again and again until the wanted vertex was found. Then, we realized that the Breadth First Search(BFS) is a better approach. We read through the class notes on BFS and found that we can keep track of the path by storing paths to a queue, and from there we can find the degree of separation. Suzy worked on Task 2.4 by creating a method that uses two separate LinkedLists to store the passed movies and failed movies, using a for loop to iterate femaleMovies.